DOCA Platform Framework (DPF) Documentation v25.10.0

Table of contents

Advanced Configuration 7

DPUServices 7

DOCA Argus Service 8

DOCA Blueman Service 16

DOCA Firefly Service 19

DOCA Telemetry Service 36

DOCA VPC OVN Service 41

OVN High Availability (HA) guide 85

VPC OVN upgrade guide 88

Host Trusted Multi-DPU support OVN-Kubernetes and HBN Services 90

Host Trusted with Non Kubernetes Workers 102

Enable Custom Certificate Authority for DMS 102

Host Installation for non-Kubernetes Environments 104

Observability 112

Secondary CNI support for HBN-OVNK use case 115

Using Private Registries 119

Zero Trust Advanced Configuration 122

Developer Guides 127

API 127

API reference 128

DPFOperatorConfig 286

DPUCluster 297

DPUDeployment 301

DPUDevice 336

DPUDiscovery 342
DPUFlavor 352
DPUNode 362
DPUService 371
DOCA Argus Service 8
DOCA Blueman Service 16
DOCA Firefly Service 19
DOCA Telemetry Service 36
DOCA VPC OVN Service 41
OVN High Availability (HA) guide 85
VPC OVN upgrade guide 88
DPUServiceChain 475
DPUServiceCredentialRequest 482
DPUServiceIPAM 488
DPUServiceNAD 496
DPUSet 500
Architecture 513
Component Description 513
Overview 524
Services 527
DPUService Development Guide 528
Storage Development Guide 538
System 579
Conditions 579
Developing a Conformant DPF System 584
End-To-End Testing 589
Local Development 590
Scale Testing 592

Getting Started 597
DPF Host Trusted 598
DPF Zero Trust 602
Helm Prerequisites 606
Lifecycle Management 624
DPF Operator Upgrade Guide 624
Platform Support 629
Release Notes 632
DOCA Platform Framework v24.10.0 632
DOCA Platform Framework v25.1.0 644
DOCA Platform Framework v25.1.1 652
DOCA Platform Framework v25.4.0 660
DOCA Platform Framework v25.7.0 671
DOCA Platform Framework v25.7.1 681
DOCA Platform Framework v25.10.0 682
Troubleshooting 701
DPF CLI - dpfctl 702
Reporting Issues with sosreport 715
User Guides 723
Host Trusted Deployment 724
DPF Prerequisites for Host Trusted 727
DPF System Prerequisites for Host Trusted 728
Host Network Configuration Prerequisites 730
Host Trusted Use Cases 736

DPU Passthrough in DPF Host Trusted 736
Host Based Networking 757
OVN Kubernetes 791
OVN Kubernetes with Host Based Networking 827
OVN Kubernetes with Host Based Networking and SNAP Block Storage 870
SNAP Storage with Host Based Networking 933
Zero Trust Deployment 1021
DPF Prerequisites for Zero Trust 1024
DPF System Prerequisites for Zero Trust 1024
Zero Trust Use Cases 1026
DPU Passthrough in DPF Zero Trust 1027
HBN and SNAP Storage in DPF Zero Trust 1054
HBN in DPF Zero Trust 1146
OVN VPC Service Deployment Guide 1216

- Introduction
  - Key Features
  - Hardware Support

> **Note**
>
> The source for this documentation is the GitHub repo at github.com/NVIDIA/doca-platform under `docs/public/`.

Introduction

DOCA Platform Framework (DPF) is a system that provisions and orchestrates NVIDIA BlueField DPUs in cloud environments.

Key Features

- **Kubernetes Integration**: Streamlines DPU provisioning and management via Kubernetes APIs
- **DPU Service Orchestration**: Enables efficient deployment and orchestration of services on DPUs
- **Cluster Management**: Simplifies the management of DPUs within Kubernetes clusters

Hardware Support

DPF supports dual port NVIDIA BlueField-3 DPUs. BlueField DPUs are installed as PCI devices in servers to manage various types of network traffic, such as data center communication and storage traffic, through dedicated ports. They feature arm64 CPUs running Linux OS and NVIDIA DOCA Software components.

The DOCA Platform Framework (DPF) documentation is organized into the following sections:

- Getting Started - Quick start guide and initial setup instructions
- Lifecycle Management - Documentation for upgrade and maintenance of DPF
- API - API reference documentation
- Architecture - System architecture and design documentation
- Storage Developer Guide - Documentation for Storage development and configuration
- DPUServices - Documentation for DPF DPU services
- DPUService Developer Guide - Documentation for DPUService development and configuration
- Troubleshooting - Documentation for troubleshooting and debugging DPF
- User Guides - Documentation for DPF users
- Release Notes - Release notes and version history
- Platform Support - Hardware requirements and supported platforms

Advanced Configuration

This page contains advanced configuration documentation for the DOCA Platform Framework (DPF). These guides cover advanced topics and configurations for power users and administrators.

Documentation Sections

- DPU Services - Documentation for DPF DPU services including Argus, Blueman, Firefly, and DOCA Telemetry Service
- Host Trusted with Non Kubernetes Workers - Installation guides for DPF Host Trusted with workers that are not part of a Kubernetes cluster
- Using Private Registries - Configuration for using private container registries
- Zero Trust Advanced Configuration - Advanced configuration for the DPF Zero Trust
- Observability Guide - Configuration to enable observability in DPF
- Secondary CNI support for HBN-OVNK use case - Documentation for enabling secondary CNI support for Host Based Networking and OVN Kubernetes
- Host Trusted Multi-DPU support OVN-Kubernetes and HBN Services - Guide that describes how to target particular DPUs for provisioning and service orchestration of OVN-Kubernetes and HBN Services.

DPUServices

This section contains documentation for the DPU services that can be deployed and managed by the DOCA Platform Framework (DPF). Each service has its own documentation, deployment guides, and configuration details:

- **Firefly**: DOCA Firefly service for secure PTP synchronization
    - Firefly User Guide
- **BlueMan**: DOCA BlueMan service for comprehensive DPU management

    - BlueMan User Guide

- **Telemetry Service (DTS)**: DOCA Telemetry Service for monitoring and metrics collection
    - DTS User Guide

- **Host-Based Networking (HBN)**: DOCA Host-Based Networking service for advanced networking capabilities
    - HBN User Guide (Host Trusted)
    - HBN User Guide (Zero Trust)

- **Argus**: DOCA Argus Service for Container Threat Detection
    - Argus User Guide

- **Storage Services**: Storage-related services including SNAP and SPDK CSI for storage management and virtualization

- **OVN VPC Service**: DOCA VPC OVN service provides accelerated Virtual Private Cloud (VPC) networking with network isolation and advanced SDN capabilities using Open Virtual Network (OVN)
    - OVN VPC Deployment Guide
    - OVN VPC Service Guide

- **Example Services**: Reference implementations and test services for development and validation

For implementation details of these services, refer to the `dpuservices` directory in the repository root.

DOCA Argus Service

This documentation explains configuration and deployment of DOCA Argus service as DPUService in DPF.

Main Argus concepts are explained in the official DOCA Argus documentation.

The official documentation provides a more comprehensive overview, DPUService users should consult it for detailed explanation of service configuration.

The DOCA Argus usecase in DPF is container threat detection in AI workloads and microservices, utilizing a Bluefield DPU to perform live machine introspection at the hardware level.

Service Components

Argus component runs on the DPU and analyzes specific snippets of volatile memory directly, providing attested insights into the operation of various workloads, whether they are bare-metal, virtualized, or containerized. By default, Argus scans all systems of the host, so for scanning specific systems only refer to official DOCA Argus documentation.

IOMMU Kernel Parameters Requirements

Virtualized environments (running inside a VM)

You must set **both** of the following kernel parameters on the **host**:

- `intel_iommu=on` **or** `amd_iommu=on`
- `iommu=pt`

  Example: `intel_iommu=on iommu=pt`

Bare-metal environments (running directly on hardware)

You have **two valid options**:

1. **Disable IOMMU completely**

   - `intel_iommu=off` **or** `amd_iommu=off`

2. **Enable IOMMU with passthrough**

   - `intel_iommu=on` **or** `amd_iommu=on`

   - `iommu=pt`

For more details, refer to the official NVIDIA DOCA Argus documentation.

Configuration

Configuration files:

DPUServiceConfiguration

```
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceConfiguration
metadata:
  name: argus
  namespace: dpf-operator-system
spec:
  deploymentServiceName: argus
  serviceConfiguration:
    helmChart:
      values:
        config:
          isLocalPath: false
        containerImage: nvcr.io/nvidia/doca/doca_argus:1.1.1-
doca3.2.1
```

DPUServiceTemplate

```yaml
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceTemplate
metadata:
  name: argus
  namespace: dpf-operator-system
spec:
  deploymentServiceName: argus
  helmChart:
    source:
      chart: doca-argus
      repoURL: https://helm.ngc.nvidia.com/nvidia/doca
      version: 1.1.1
```

The general resources are:

DPUFlavor

Defines the DPU flavor for the Argus service.

DPUFlavor

```
apiVersion: provisioning.dpu.nvidia.com/v1alpha1
kind: DPUFlavor
metadata:
  name: dpf-provisioning-argus
  namespace: dpf-operator-system
spec:
  bfcfgParameters:
    - UPDATE_ATF_UEFI=yes
    - UPDATE_DPU_OS=yes
    - WITH_NIC_FW_UPDATE=yes
  configFiles:
    - operation: override
      path: /etc/mellanox/mlnx-bf.conf
      permissions: "0644"
      raw: |
        ALLOW_SHARED_RQ="no"
        IPSEC_FULL_OFFLOAD="no"
        ENABLE_ESWITCH_MULTIPORT="yes"
    - operation: override
      path: /etc/mellanox/mlnx-ovs.conf
      permissions: "0644"
      raw: |
        CREATE_OVS_BRIDGES="no"
        OVS_DOCA="yes"
    - operation: override
      path: /etc/mellanox/mlnx-sf.conf
      permissions: "0644"
      raw: ""
  grub:
    kernelParameters:
      - console=hvc0
      - console=ttyAMA0
      - earlycon=pl011,0x13010000
      - fixrttc
      - net.ifnames=0
```

```
      - biosdevname=0
      - iommu.passthrough=1
      - cgroup_no_v1=net_prio,net_cls
      - hugepagesz=2048kB
      - hugepages=3072
  nvconfig:
    - device: '*'
      parameters:
        - PF_BAR2_ENABLE=0
        - PER_PF_NUM_SF=1
        - PF_TOTAL_SF=20
        - PF_SF_BAR_SIZE=10
        - NUM_PF_MSIX_VALID=0
        - PF_NUM_PF_MSIX_VALID=1
        - PF_NUM_PF_MSIX=228
        - INTERNAL_CPU_MODEL=1
        - INTERNAL_CPU_OFFLOAD_ENGINE=0
        - SRIOV_EN=1
        - NUM_OF_VFS=46
        - LAG_RESOURCE_ALLOCATION=1
        - LINK_TYPE_P1=ETH
        - LINK_TYPE_P2=ETH
  ovs:
    rawConfigScript: |
      _ovs-vsctl() {
        ovs-vsctl --no-wait --timeout 15 "$@"
      }

      _ovs-vsctl set Open_vSwitch . other_config:doca-init=true
      _ovs-vsctl set Open_vSwitch . other_config:dpdk-max-
memzones=50000
      _ovs-vsctl set Open_vSwitch . other_config:hw-offload=true
      _ovs-vsctl set Open_vSwitch . other_config:pmd-quiet-
idle=true
      _ovs-vsctl set Open_vSwitch . other_config:max-idle=20000
```

```
      _ovs-vsctl set Open_vSwitch . other_config:max-
revalidator=5000
      _ovs-vsctl set Open_vSwitch . other_config:ctl-pipe-
size=1024
      _ovs-vsctl --if-exists del-br ovsbr1
      _ovs-vsctl --if-exists del-br ovsbr2
      _ovs-vsctl --may-exist add-br br-sfc
      _ovs-vsctl set bridge br-sfc datapath_type=netdev
      _ovs-vsctl set bridge br-sfc fail_mode=secure
      _ovs-vsctl --may-exist add-port br-sfc p0
      _ovs-vsctl set Interface p0 type=dpdk
      _ovs-vsctl set Port p0 external_ids:dpf-type=physical

      _ovs-vsctl set Open_vSwitch . external-ids:ovn-bridge-
datapath-type=netdev
      _ovs-vsctl --may-exist add-br br-ovn
      _ovs-vsctl set bridge br-ovn datapath_type=netdev
      _ovs-vsctl br-set-external-id br-ovn bridge-id br-ovn
      _ovs-vsctl br-set-external-id br-ovn bridge-uplink
puplinkbrovntobrsfc
      _ovs-vsctl --may-exist add-port br-ovn pf0hpf
      _ovs-vsctl set Interface pf0hpf type=dpdk
```

DPUDeployment

Defines the DPUDeployment for the Argus service.

DPUDeployment

```yaml
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUDeployment
metadata:
  name: argus
  namespace: dpf-operator-system
spec:
  dpus:
    bfb: bf-bundle
    dpuSets:
    - nameSuffix: dpuset-argus
      nodeSelector:
        matchLabels:
          feature.node.kubernetes.io/dpu-enabled: "true"
    flavor: dpf-provisioning-argus
  serviceChains:
    switches:
    - ports:
      - serviceInterface:
          matchLabels:
            uplink: p0
    upgradePolicy:
      applyNodeEffect: true
  services:
    argus:
      serviceConfiguration: argus
      serviceTemplate: argus
```

Configuration

Official Argus documentation explains configuration options.

DPUDeployment

The complete `DPUDeployment` configuration is in DPUDeployment.yaml.

Output

Argus offers multiple ways to get events, that includes logs to stdout, log files and telemtry records in json or syslog formats.

DOCA Blueman Service

Overview

DOCA BlueMan is a standalone web dashboard that runs on the DPU, providing consolidated information about system health, basic information, and telemetry counters through a single interface. All data is gathered from the DOCA Telemetry Service (DTS).

Prerequisites

- DTS service must be running
- DTS Guide

Documentation

- Official Blueman Documentation

Deployment

Starting the Service

To start the BlueMan service, you need to create a `DPUService` resource.

This resource will define the service configuration and deployment details.

Configuration file:

DPUService

```yaml
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUService
metadata:
  name: doca-blueman-service
  namespace: dpf-operator-system
spec:
  helmChart:
    source:
      repoURL: https://helm.ngc.nvidia.com/nvidia/doca
      version: 1.0.8
      chart: doca-blueman
  serviceDaemonSet:
    updateStrategy:
      type: RollingUpdate
      rollingUpdate:
        maxUnavailable: 2
    labels:
      dpuservice.dpu.nvidia.com/name: doca-blueman-service
    annotations:
      dpuservice.dpu.nvidia.com/name: doca-blueman-service
```

```
kubectl apply -f DPUService.yaml
```

Verification

```
kubectl -n dpf-operator-system exec deploy/dpf-operator-
controller-manager -- /dpfctl describe all --show-
resources=dpuservice/doca-blueman-service --show-
conditions=dpuservices
NAME                                                NAMESPACE
STATUS       REASON   SINCE  MESSAGE
DPFOperatorConfig/dpfoperatorconfig                 dpf-operator-
system  Ready: True  Success  85m
  DPUServices
    DPUService/doca-blueman-service                 dpf-operator-
system
                  Ready
True         Success  15m
                  ApplicationPrereqsReconciled
True         Success  17m
                  ApplicationsReady
True         Success  15m
                  ApplicationsReconciled
True         Success  17m
                  ConfigPortsReconciled
True         Success  17m
                  DPUServiceInterfaceReconciled
True         Success  17m
```

Network Configuration

Accessing the UI

By default, the Blueman UI is not accessible from outside the DPF host. To enable external access, configure IP prerouting using the following commands:

```
iptables -t nat -A PREROUTING -p tcp --dport 10000 -j DNAT --to-destination $DPF_BF_IP:10000
iptables -t nat -A PREROUTING -p tcp --dport 443 -j DNAT --to-destination $DPF_BF_IP:443
```

Accessing the Dashboard

Open your browser and navigate to:

```
https://$DPF_HOST_IP/login
```

DOCA Firefly Service

This documentation explains configuration and deployment of DOCA Firefly service as DPUService in DPF.

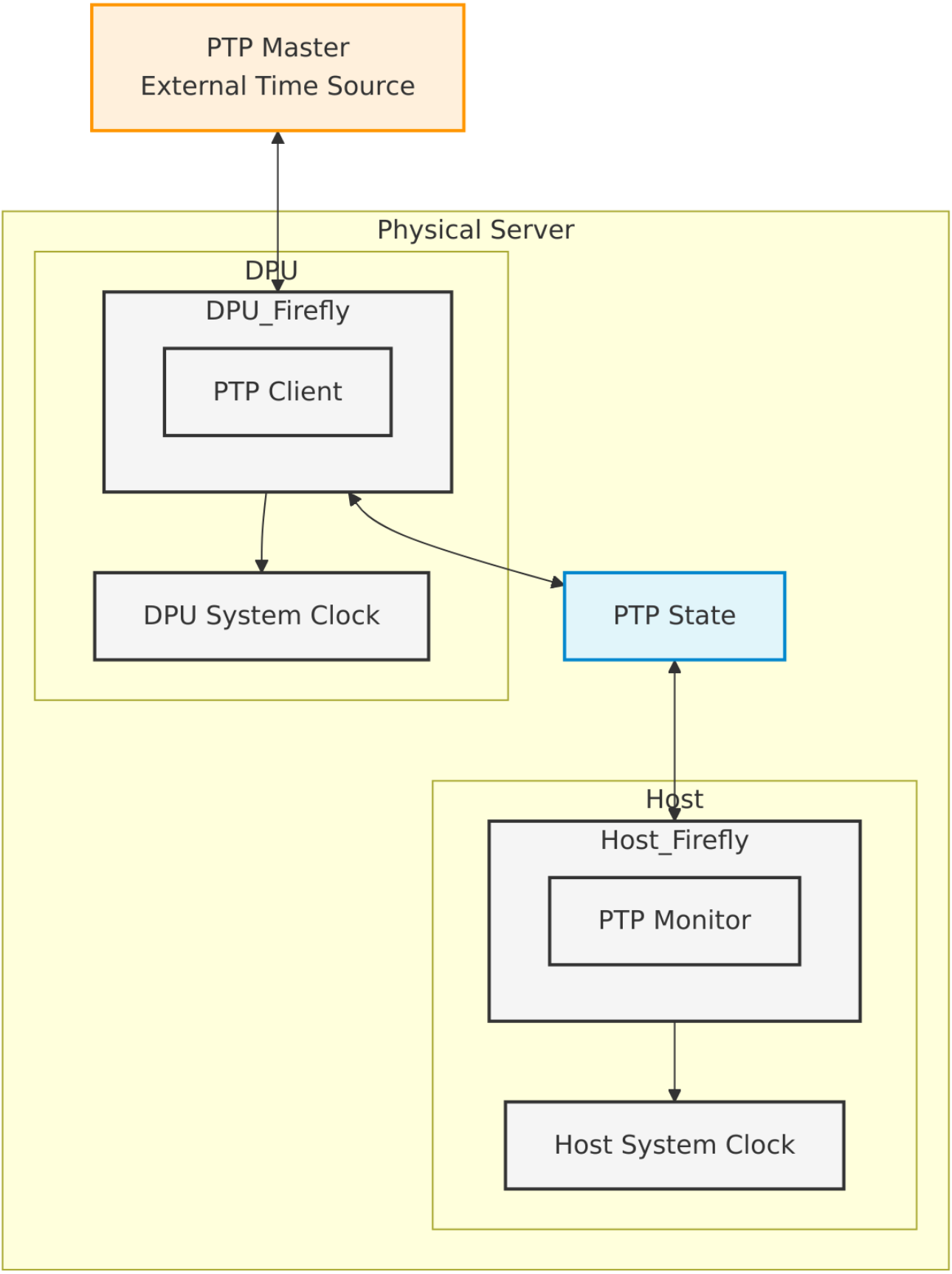
Main Firefly concepts are explained in the official DOCA Firefly documentation.

While the official documentation provides a more comprehensive overview, DPUService users should consult it for detailed explanation of PTP configuration and monitoring options.

The DOCA Firefly usecase in DPF is mainly to provide PTP time synchronization for the host system clocks.

We split the service into two components: one **running on the DPU**, which is running the PTP software stack, and the **other on the host**.

A high-level overview of the **Firefly DPF service architecture** is shown below.


PTP Master
External Time Source
Physical Server
DPU
DPU_Firefly
PTP Client
DPU System Clock
PTP State
Host
Host_Firefly
PTP Monitor
Host System Clock

Service Components

Firefly consists of two main components:

1. DPU Component:

```
* Runs on the DPU
* Acts as the PTP client
* Handles PTP time synchronization
* Sets DPU system clock
```

Configuration files:

DPUServiceConfiguration-dpu

```
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceConfiguration
metadata:
  name: firefly-dpu
  namespace: dpf-operator-system
spec:
  deploymentServiceName: firefly-dpu
  interfaces:
    - name: fireflyiface
      network: mybrsfc-firefly
  serviceConfiguration:
    serviceDaemonSet:
      labels:
        svc.dpu.nvidia.com/custom-flows: firefly
    configPorts:
      ports:
        - name: monitor
          port: 25600
          protocol: TCP
      serviceType: ClusterIP
    helmChart:
      values:
        exposedPorts:
          ports:
            monitor: true
        ptpConfig: ptp.conf
        ptpInterfaces: fireflyiface
        config:
          content:
            ptp.conf: |
              [global]
              domainNumber                    24
              clientOnly                      1
              verbose                         1
              logging_level                   6
```

```
                dataset_comparison              G.8275.x
                G.8275.defaultDS.localPriority  128
                maxStepsRemoved                 255
                logAnnounceInterval             -3
                logSyncInterval                 -4
                logMinDelayReqInterval          -4
                G.8275.portDS.localPriority     128
                ptp_dst_mac                     01:80:C2:00:00:0E
                network_transport               L2
                fault_reset_interval            1
                hybrid_e2e                      0

                [fireflyiface]
```

DPUServiceTemplate-dpu

```
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceTemplate
metadata:
  name: firefly-dpu
  namespace: dpf-operator-system
spec:
  deploymentServiceName: firefly-dpu
  helmChart:
    source:
      chart: doca-firefly
      repoURL: https://helm.ngc.nvidia.com/nvidia/doca
      version: 1.1.8
    values:
      containerImage: nvcr.io/nvidia/doca/doca_firefly:1.7.4-
doca3.2.0
      hostNetwork: false
      enableTXPortTimestampOffloading: true
      monitorState: 0.0.0.0
      phc2sysArgs: -a -r -l 6
      config:
        isLocalPath: false
  resourceRequirements:
    memory: 512Mi
```

2. Host Component:

```
* Runs on the host
* Monitors PTP time synchronization
* Sets host system clock
```

Configuration files:

DPUServiceConfiguration-host.yaml

```
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceConfiguration
metadata:
  name: firefly-host
  namespace: dpf-operator-system
spec:
  deploymentServiceName: firefly-host
  upgradePolicy:
    applyNodeEffect: false
  serviceConfiguration:
    deployInCluster: true
    helmChart:
      values:
        monitorState: '{{ (index .Services "firefly-dpu").Name }}.{{ (index .Services "firefly-dpu").Namespace }}'
```

DPUServiceTemplate-host

```
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceTemplate
metadata:
  name: firefly-host
  namespace: dpf-operator-system
spec:
  deploymentServiceName: firefly-host
  helmChart:
    source:
      chart: doca-firefly
      repoURL: https://helm.ngc.nvidia.com/nvidia/doca
      version: 1.1.8
    values:
      containerImage: nvcr.io/nvidia/doca/doca_firefly:1.7.4-
doca3.2.0-host
      hostNetwork: false
      monitorClientPhc2sysInterface: eth0
      monitorClientType: phc2sys
      phc2sysState: disable
      ppsDevice: disable
      ppsState: do_nothing
      ptpState: disable
      tolerations:
        - effect: NoSchedule
          key: k8s.ovn.org/network-unavailable
          operator: Exists
  resourceRequirements:
    memory: 512Mi
```

The general resources are:

DPUFlavor

Defines the DPU flavor for the Firefly service.

DPUFlavor

```yaml
apiVersion: provisioning.dpu.nvidia.com/v1alpha1
kind: DPUFlavor
metadata:
  annotations:
    provisioning.dpu.nvidia.com/num-of-trusted-sfs: "5"
  name: dpf-provisioning-firefly
  namespace: dpf-operator-system
spec:
  bfcfgParameters:
    - UPDATE_ATF_UEFI=yes
    - UPDATE_DPU_OS=yes
    - WITH_NIC_FW_UPDATE=yes
  configFiles:
    - operation: override
      path: /etc/mellanox/mlnx-bf.conf
      permissions: "0644"
      raw: |
        ALLOW_SHARED_RQ="no"
        IPSEC_FULL_OFFLOAD="no"
        ENABLE_ESWITCH_MULTIPORT="yes"
    - operation: override
      path: /etc/mellanox/mlnx-ovs.conf
      permissions: "0644"
      raw: |
        CREATE_OVS_BRIDGES="no"
        OVS_DOCA="yes"
    - operation: override
      path: /etc/mellanox/mlnx-sf.conf
      permissions: "0644"
      raw: ""
  grub:
    kernelParameters:
      - console=hvc0
      - console=ttyAMA0
      - earlycon=pl011,0x13010000
```

```
      - fixrttc
      - net.ifnames=0
      - biosdevname=0
      - iommu.passthrough=1
      - cgroup_no_v1=net_prio,net_cls
      - hugepagesz=2048kB
      - hugepages=3072
  nvconfig:
    - device: '*'
      parameters:
        - PF_BAR2_ENABLE=0
        - PER_PF_NUM_SF=1
        - PF_TOTAL_SF=20
        - PF_SF_BAR_SIZE=10
        - NUM_PF_MSIX_VALID=0
        - PF_NUM_PF_MSIX_VALID=1
        - PF_NUM_PF_MSIX=228
        - INTERNAL_CPU_MODEL=1
        - INTERNAL_CPU_OFFLOAD_ENGINE=0
        - SRIOV_EN=1
        - NUM_OF_VFS=46
        - LAG_RESOURCE_ALLOCATION=1
        - REAL_TIME_CLOCK_ENABLE=1
        - LINK_TYPE_P1=ETH
        - LINK_TYPE_P2=ETH
  ovs:
    rawConfigScript: |
      _ovs-vsctl() {
        ovs-vsctl --no-wait --timeout 15 "$@"
      }

      _ovs-vsctl set Open_vSwitch . other_config:doca-init=true
      _ovs-vsctl set Open_vSwitch . other_config:dpdk-max-
memzones=50000
      _ovs-vsctl set Open_vSwitch . other_config:hw-offload=true
```

```
      _ovs-vsctl set Open_vSwitch . other_config:pmd-quiet-
idle=true
      _ovs-vsctl set Open_vSwitch . other_config:max-idle=20000
      _ovs-vsctl set Open_vSwitch . other_config:max-
revalidator=5000
      _ovs-vsctl set Open_vSwitch . other_config:ctl-pipe-
size=1024
      _ovs-vsctl --if-exists del-br ovsbr1
      _ovs-vsctl --if-exists del-br ovsbr2
      _ovs-vsctl --may-exist add-br br-sfc
      _ovs-vsctl set bridge br-sfc datapath_type=netdev
      _ovs-vsctl set bridge br-sfc fail_mode=secure
      _ovs-vsctl --may-exist add-port br-sfc p0
      _ovs-vsctl set Interface p0 type=dpdk
      _ovs-vsctl set Port p0 external_ids:dpf-type=physical

      _ovs-vsctl set Open_vSwitch . external-ids:ovn-bridge-
datapath-type=netdev
      _ovs-vsctl --may-exist add-br br-ovn
      _ovs-vsctl set bridge br-ovn datapath_type=netdev
      _ovs-vsctl br-set-external-id br-ovn bridge-id br-ovn
      _ovs-vsctl br-set-external-id br-ovn bridge-uplink
puplinkbrovntobrsfc
      _ovs-vsctl --may-exist add-port br-ovn pf0hpf
      _ovs-vsctl set Interface pf0hpf type=dpdk

      # Disabling DPU NTP. Requires functional PTP setup.
      systemctl disable ntpsec --now
```

DPUServiceNAD

Defines the trusted Scalable Function (SF) for the Firefly service.

DPUServiceNAD

```yaml
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceNAD
metadata:
  name: mybrsfc-firefly
  namespace: dpf-operator-system
  annotations:
    dpuservicenad.svc.dpu.nvidia.com/use-trusted-sfs: ""
spec:
  resourceType: sf
  ipam: false
  bridge: "br-sfc"
  serviceMTU: 1500
```

DPUDeployment

Defines the DPUDeployment for the Firefly service.

DPUDeployment

```
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUDeployment
metadata:
  name: ovn-firefly
  namespace: dpf-operator-system
spec:
  dpus:
    bfb: bf-bundle
    dpuSets:
      - nameSuffix: dpuset1
        nodeSelector:
          matchLabels:
            feature.node.kubernetes.io/dpu-enabled: "true"
    flavor: dpf-provisioning-firefly
  serviceChains:
    switches:
      - ports:
          - serviceInterface:
              matchLabels:
                uplink: p0
          - serviceInterface:
              matchLabels:
                port: ovn
          - service:
              interface: fireflyiface
              name: firefly-dpu
  services:
    firefly-dpu:
      serviceConfiguration: firefly-dpu
      serviceTemplate: firefly-dpu
    firefly-host:
      serviceConfiguration: firefly-host
      serviceTemplate: firefly-host
      dependsOn:
        - name: firefly-dpu
```

```
    ovn:
      serviceConfiguration: ovn
      serviceTemplate: ovn
```

For information about OVN Kubernetes configuration see the OVN Kubernetes user guide

Configuration

Official Firefly documentation explains configuration options.

General note: In the official documentation, all options should be specified in the `ptp.conf` file.

In DPF the same options should be set via the `DPUServiceConfiguration`.

Preconfiguration

Our referenced example `DPUServiceConfigration` includes G.8275.1 PTP Profile configuration. You can set all necessary configurations to your needs.

Network Configuration

The Firefly service requires a **trusted Scalable Function (SF)** to enable secure PTP communication. This interface is added to the SFC bridge using a `DPUServiceNAD` (Network Attachment Definition). The `DPUService` controller takes care of injecting the correct resource to the Pod using that DPUServiceNAD.

DPUDeployment

The complete `DPUDeployment` configuration is in DPUDeployment.

Toleration Configuration

The Firefly service includes a special configuration (toleration) that ensures it always runs on the host system. This is important because:

1. The service needs to run on the host to properly synchronize the system time.
2. Without proper time synchronization, other system components might not work correctly.
3. The toleration prevents the service from being blocked from running on the host.

This configuration is automatically handled in the DPUServiceTemplate and does not require any user configuration.

Check the Status

You can check the status of the Firefly service using the following command:

```
$ kubectl -n dpf-operator-system exec deploy/dpf-operator-
controller-manager -- /dpfctl describe all --grouping=false --show-
conditions=dpuservices
NAME
NAMESPACE            STATUS        REASON             SINCE
MESSAGE
DPFOperatorConfig/dpfoperatorconfig                          dpf-
operator-system  Ready: Trye   Success            31s
  DPUClusters
    DPUCluster/dpu-cplane-tenant1                            dpu-
cplane-tenant1   Ready: True   HealthCheckPassed  27h
  DPUDeployments
    DPUDeployment/firefly                                    dpf-
operator-system  Ready: True   Success            12m
      DPUServices
        DPUService/firefly-dpu-v6pbk                         dpf-
operator-system
                      Ready
True          Success            8m24s
                      ApplicationPrereqsReconciled
True          Success            3h7m
                      ApplicationsReady
True          Success            8m24s
                      ApplicationsReconciled
True          Success            3h7m
                      ConfigPortsReconciled
True          Success            3h7m
                      DPUServiceInterfaceReconciled
True          Success            3h7m
        DPUService/firefly-host-jj98d                        dpf-
operator-system
                      Ready
True          Success            4m46s
                      ApplicationPrereqsReconciled
True          Success            27h
```

```
                        ApplicationsReady
True            Success             4m46s
                        ApplicationsReconciled
True            Success             27h
                        ConfigPortsReconciled
True            Success             27h
                        DPUServiceInterfaceReconciled
True            Success             27h
...
```

DOCA Telemetry Service

This documentation explains configuration and deployment of DOCA Telemetry service (DTS) as DPU service in DPF.

Main DTS concepts are explained in the official DTS documentation.

While the official documentation provides a more comprehensive overview, DPUService users should consult it for its detailed list of telemetry counters and explanation of user options.

Service configuration

Official DTS documentation explains configuration options.

General note: In the official documentation, all options should be specified in the `dts_config.ini` file using the `key=value` format. In DPF the same options should be set via `DPUService.yaml` as `key: value` in yaml format.

To start the DTS service, you need to create a `DPUService` resource.

This resource will define the service configuration and deployment details.

Configuration file:

DPUService

```yaml
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUService
metadata:
  name: doca-telemetry-service
  namespace: dpf-operator-system
spec:
  helmChart:
    source:
      repoURL: https://helm.ngc.nvidia.com/nvidia/doca
      version: 1.23.4
      chart: doca-telemetry
    values:
      configMapData:
        providers: "sysfs,ethtool"
        # aggregator_providers: "fluent_aggr,prometheus_aggr"
        general:
          update: 1000
          sync-time-limit: 10000
          event-buffer-size: 65536
          counter-buffer-size: 65536
        fluent:
          forward:
            enable: 0
            port: 24224
            host: "127.0.0.1"
            tag: "doca_telemetry"
          kafka:
            enable: 0
            brokers: "127.0.0.1:9092"
            topics: "telemetry"
            format: "json"
          influxdb:
            enable: 0
            host: "127.0.0.1"
            port: 8086
```

```
            tag: "dts_measurement"
          elasticsearch:
            enable: 0
            host: "127.0.0.1"
            port: 9200
            index: "dts"
  serviceDaemonSet:
    updateStrategy:
      type: RollingUpdate
      rollingUpdate:
        maxUnavailable: 2
    labels:
      dpuservice.dpu.nvidia.com/name: doca-telemetry-service
    annotations:
      dpuservice.dpu.nvidia.com/name: doca-telemetry-service
  configPorts:
    serviceType: None
    ports:
      - name: httpserverport
        port: 9100
        protocol: TCP
```

```
kubectl apply -f DPUService.yaml
```

Default configuration

The default configuration is set to collect `sysfs` and `ethtool` counters and expose them via Prometheus endpoint.

DPF configuration

User configuration is organized into the following groups:

- providers
- aggregator_providers
- general
- fluent

All user options should be set in DPUService under:

```
spec:
  helmChart:
    values:
      configMapData:
```

The following subsections explain each user option groups

Providers:

Available data providers are listed here: DTS providers.

Note: some of the providers are supported only on host and not on DPU.

To enable data providers in DTS service:

1. Set data providers csv-line under `spec.helmChart.values.configMapData.providers`. E.g.:

```
spec:
  helmChart:
    values:
      configMapData:
        providers: "sysfs,ethtool"
```

2. Optionally, set aggregator providers csv-line under `spec.helmChart.values.configMapData.aggregator_providers`.

General options

Except provider names, official documentation explains various options of `dts_config.ini` file in format of `key=value` pairs.

To set the these options in DPUService.yaml specify them in `key: value` pairs under `spec.helmChart.configMapData.general`.

For instance, DPU service exposes essential general options:

- Sampling interval options:
- `update: 1000` to set the sample interval in milliseconds.
- `sync-time-limit: 10000` buffer rotation time limit in seconds.

Configure Fluent exporter

Additionally Fluent Bit export files can be set via `spec.helmChart.configMapData.fluent.DESTINATION`. Available fluent export destinations are:

- forward
- kafka
- influxdb
- elasticsearch

Default values are exposed for each case in DPUService.yaml. See more details regarding each destination in official documentation.

Run Service

Start service with:

```
kubectl apply -f DPUService.yaml
```

Check the status of DPU service:

```
$ kubectl -n dpf-operator-system exec deploy/dpf-operator-
controller-manager -- /dpfctl describe all --show-
resources=dpuservice/doca-telemetry-service --show-conditions=all
NAME                                                NAMESPACE
READY  REASON   SINCE  MESSAGE
DPFOperatorConfig/dpfoperatorconfig                 dpf-operator-
system  True   Success  15h
             ImagePullSecretsReconciled
True   Success  25d
             SystemComponentsReady
True   Success  15h
             SystemComponentsReconciled
True   Success  25d
  DPUServices
    DPUService/doca-telemetry-service               dpf-operator-
system  True   Success  21d
                ApplicationPrereqsReconciled
True   Success  25d
                ApplicationsReady
True   Success  21d
                ApplicationsReconciled
True   Success  25d
                DPUServiceInterfaceReconciled
True   Success  25d
```

DOCA VPC OVN Service

> **Note**
>
> The OVN VPC service is considered a tech preview and is not recommended for production use.

Overview

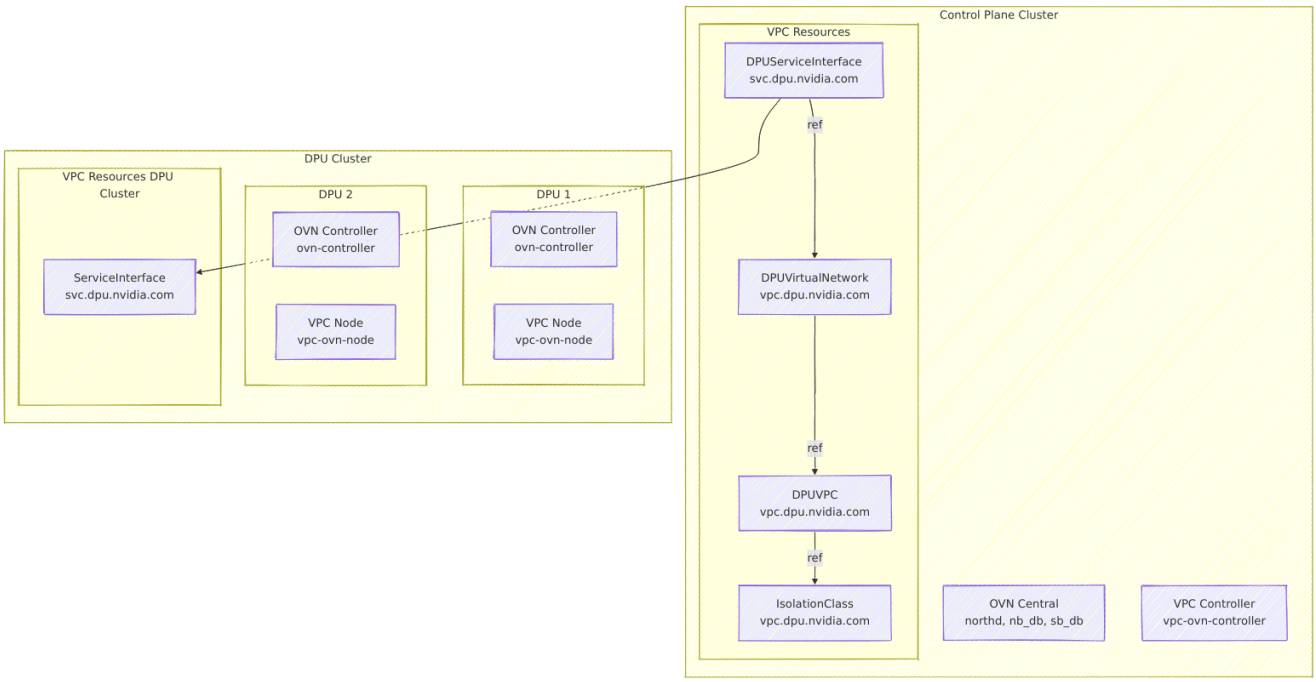
The DOCA VPC OVN Service provides accelerated Virtual Private Cloud (VPC) networking functionality for the DOCA Platform Framework (DPF). Built on top of Open Virtual Network (OVN), this service enables network isolation, virtualization, and advanced SDN capabilities directly on NVIDIA DPUs.

Key Features

- **Multi-tenant Network Isolation**: Create isolated VPCs for different tenants with guaranteed network separationSingle VPC with a single DPUVirtualNetwork and a DPUServiceInterface targeting all DPU Nodes
- **Virtual Network Management**: Support the creation of virtual networks with DHCP and custom IP addressing
- **External Connectivity**: Configurable external routing with NAT/masquerading capabilities
- **Hardware Acceleration**: Leverages DPU hardware acceleration for high-performance networking
- **Flexible Topology**: Support for complex network topologies with inter-network routing controls
- **Kubernetes Integration**: Native Kubernetes resources for declarative VPC management

Architecture

The OVN VPC service consists of several key components that work together:



Component Responsibilities

Control Plane Components

- **OVN Central**: Centralized SDN control plane managing OVN northbound and southbound databases
- **VPC Controller**: Manages VPC-specific Kubernetes resources and coordinates with OVN Central

DPU Cluster Components

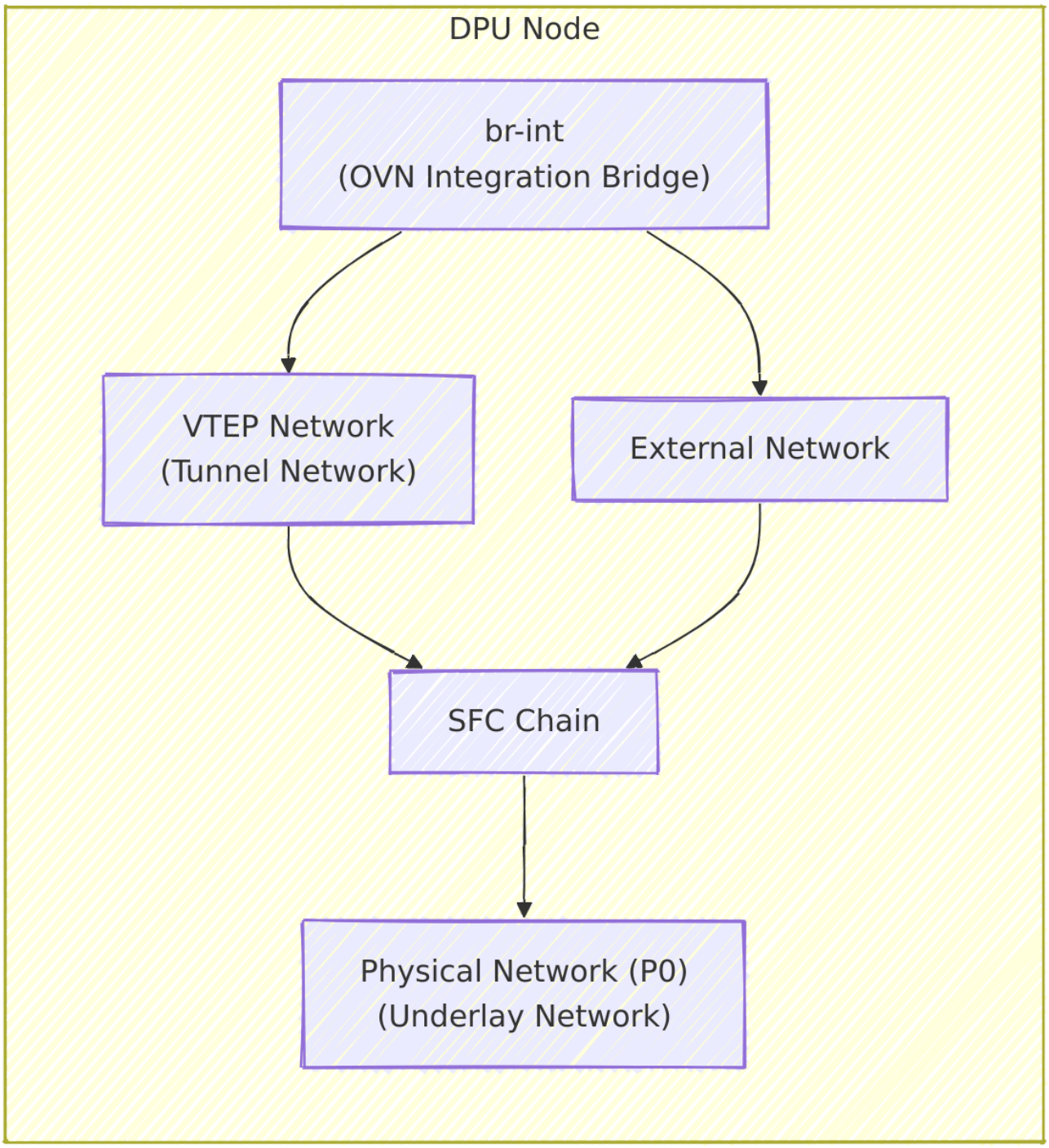
- **OVN Controller**: Local SDN data plane on the DPU, implementing network policies and packet forwarding
- **VPC Node**: Handles local VPC networking operations, including IP allocation and interface management

VPC Kubernetes Resources

- **IsolationClass**: Configuration parameters for VPC implementation
- **DPUVPC**: A Virtual Private Cloud resource that defines an isolated network environment for a tenant, including DPU node selection and isolation configuration
- **DPUVirtualNetwork**: A logical network segment within a VPC
- **DPUServiceInterface**: A control plane resource that defines network interface configuration for virtual networks
- **ServiceInterface**: Physical and virtual network interfaces managed by the DPU that implement network connectivity

DPU Network Configuration

The following diagram depicts the network configuration for a DPU, as facilitated by the OVN VPC service.



- **OVN Integration Bridge**: The main bridge (OVS) that connects interfaces to OVN and handles OVN flows
- **VTEP Network**: The tunnel endpoint interface that enables overlay network connectivity between DPUs
- **External Network**: The interface that provides connectivity to external networks outside the VPC

- **Physical Network**: The primary network interface (typically p0) that connects the DPU to the physical network infrastructure

API Overview

For the full API definition, refer to DPF API Documentation.

IsolationClass

An IsolationClass defines the parameters to be used by the VPC implementation when reconciling a DPUVPC. It specifies:

- **Provisioner**: The VPC provisioner (ovn.vpc.dpu.nvidia.com)
- **Parameters**: Implementation-specific parameters. In this case, OVN-specific configuration such as database endpoints
- **Cluster Scope**: Shared across all VPCs using the same isolation class

Reference Example

```yaml
apiVersion: vpc.dpu.nvidia.com/v1alpha1
kind: IsolationClass
metadata:
  name: ovn.vpc.dpu.nvidia.com
spec:
  # Provisioner type
  provisioner: "ovn.vpc.dpu.nvidia.com"

  # OVN-specific parameters
  parameters:
    # OVN northbound database endpoint
    ovn-nb-endpoint: "tcp:10.1.1.100:30641"
    # Connection retry interval for northbound database (Default: 5 seconds)
    ovn-nb-reconnect-time: "5"
    # OVN southbound database endpoint
    ovn-sb-endpoint: "tcp:10.1.1.100:30642"
    # Connection retry interval for southbound database (Default: 5 seconds)
    ovn-sb-reconnect-time: "5"
```

Note

`ovn-nb-endpoint, ovn-sb-endpoint` should be set to an IP address through which the OVN service is accessible (e.g. the IP of a control plane node).

DPUVPC

A DPUVPC represents a Virtual Private Cloud isolated network environment for a specific tenant. It defines:

- **Tenant Ownership**: Each VPC belongs to a specific tenant
- **Node Selection**: Specifies which DPU nodes participate in the VPC
- **Isolation Class**: The isolation mechanism (OVN-based) to use, referenced by name
- **Inter-Network Access**: Whether virtual networks within the VPC can communicate

Reference Example

```
apiVersion: vpc.dpu.nvidia.com/v1alpha1
kind: DPUVPC
metadata:
  name: example-vpc
  namespace: default
spec:
  # Tenant identifier
  tenant: my-tenant

  # Optional: Select specific DPU nodes (empty = all nodes)
  nodeSelector: {}

  # Isolation class to use, referenced by name
  isolationClassName: "ovn.vpc.dpu.nvidia.com"

  # Whether virtual networks within the DPUVPC can communicate
with each other
  interNetworkAccess: true
```

> **Note**
>
> A DPU node can belong to at most one DPUVPC, as selected by the `spec.nodeSelector` field.

DPUVirtualNetwork

A DPUVirtualNetwork defines a logical network segment within a VPC and supports:

- **Bridged Networks**: Layer 2 bridged networks with DHCP support
- **External Routing**: Optional external connectivity with masquerading
- **IP Address Management**: Automated DHCP with configurable subnets and exclusions
- **Node Selection**: Selective deployment across DPU nodes

Reference Example

```yaml
apiVersion: vpc.dpu.nvidia.com/v1alpha1
kind: DPUVirtualNetwork
metadata:
  name: example-network
  namespace: default
spec:
  # VPC this network belongs to, references a DPUVPC by name in the
same namespace
  vpcName: "example-vpc"

  # Network type (currently only Bridged is supported)
  type: "Bridged"

  # Enable routing to external networks
  externallyRouted: true

  # Enable NAT for external traffic (default: true)
  masquerade: true

  # Bridged network configuration
  bridgedNetwork:
    ipam:
      ipv4:
        # Enable DHCP server
        dhcp: true
        # Network subnet (first IP becomes gateway)
        subnet: "192.178.0.0/16"
        # Optional: Exclude specific IPs from DHCP
        excludeIPs:
          - ip: "192.178.100.10"
          - range:
              start: "192.178.100.100"
              end: "192.178.100.110"
```

DPUServiceInterface

A DPUServiceInterface defines a network interface for DPUs that can be used by workloads running on the DPU or Host. You can reference a DPUVirtualNetwork by name to indicate that the specified DPUServiceInterface should be connected to a specific virtual network.

Supported interface types: `PF`, `VF`, `Service`.

> **Note**
>
> A DPUServiceInterface of type `Service` is intended for advanced use cases where a DPUService network interface needs to be connected to a virtual network. see *Advanced Use-Cases* section below

Reference Example

```
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceInterface
metadata:
  name: vf2-example-network
  namespace: default
spec:
  template:
    spec:
      template:
        spec:
          interfaceType: vf
          vf:
            pfID: 0
            vfID: 2
            parentInterfaceRef: ""
            # DPUVirtualNetwork reference by name in the same
namespace
            virtualNetwork: example-network
```

Deployment & Prerequisites

Please follow the OVN VPC deployment guide to deploy the OVN VPC Service.

OVN VPC Network Features

1. **Intra-VPC Communication**:
   - Traffic between virtual networks within the same VPC is controlled by the `spec.interNetworkAccess` setting in DPUVPC.
   - When enabled, routing is handled by OVN logical routers.
   - When disabled, networks are completely isolated.
2. **External Connectivity**:

- Virtual networks with `spec.externallyRouted: true` can reach external networks.
- Outbound traffic is masqueraded by default unless `spec.masquerade: false`.

3. **DHCP Services**:
   - Each virtual network with `spec.bridgedNetwork.ipam.ipv4.dhcp: true` gets an OVN DHCP server.
   - The first IP in the subnet becomes the gateway.
   - IP exclusions are supported for static assignments.

VPC Network Topology Examples

All examples below assume OVN VPC is deployed together with an `IsolationClass` named `ovn.vpc.dpu.nvidia.com` as described in the deployment guide.

After applying each example, check the status of the resources. Once all resources are ready, you can test connectivity on the host between VF interfaces that correspond to the specified DPUServiceInterface by:

1. Requesting DHCP from OVN (e.g., using `dhclient -v -1 <interface>`).
2. Running traffic between host network interfaces.

1. Single VPC with a single DPUVirtualNetwork and a DPUServiceInterface targeting all DPU Nodes

This basic example allows hosts to communicate with each other (E/W) and to communicate with the external network behind NAT in an isolated manner (e.g., in case the network infrastructure is shared with additional nodes).

The topology consists of:

1. A DPUVPC: `testvpc`
2. A DPUVirtualNetwork: `testnet1`

3. A DPUServiceInterface of type `VF` targeting VF ID `2`: `testnet1-vf2`

VPC Example Topology

```
---
apiVersion: vpc.dpu.nvidia.com/v1alpha1
kind: DPUVPC
metadata:
  name: testvpc
  namespace: default
spec:
  tenant: foo
  isolationClassName: ovn.vpc.dpu.nvidia.com
  interNetworkAccess: true
  nodeSelector: {}
---
apiVersion: vpc.dpu.nvidia.com/v1alpha1
kind: DPUVirtualNetwork
metadata:
  name: testnet1
  namespace: default
spec:
  vpcName: testvpc
  nodeSelector: {}
  type: Bridged
  externallyRouted: true
  masquerade: true
  bridgedNetwork:
    ipam:
      ipv4:
        dhcp: true
        subnet: 192.178.0.0/16
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceInterface
metadata:
  name: testnet1-vf2
  namespace: default
spec:
```

```
  template:
    spec:
      template:
        spec:
          interfaceType: vf
          vf:
            pfID: 0
            vfID: 2
            parentInterfaceRef: ""
            virtualNetwork: testnet1
```

2. Single VPC with Two DPUVirtualNetworks and Two DPUServiceInterfaces targeting separate DPU nodes

This example associates different DPU nodes with different DPUVirtualNetworks within the same VPC.

The topology consists of:

1. A DPUVPC: `testvpc`
2. Two DPUVirtualNetworks: `testnet1`, `testnet2`
3. Two DPUServiceInterfaces targeting different DPU nodes by label: `testnet1-vf2`, `testnet2-vf2`

DPU nodes should be labeled with a distinct label to allow the association of each DPUServiceInterface to a distinct set of nodes.

In this example, we assume two groups of DPU nodes are labeled with `vpc.dpu.nvidia.com/network=testnet1` and `vpc.dpu.nvidia.com/network=testnet2` respectively.

Hosts can communicate with each other (E/W) in an isolated manner as well as reach the external network behind NAT. Nodes will belong to different virtual networks (subnets) according to the specified label.

VPC Example Topology

```
---
apiVersion: vpc.dpu.nvidia.com/v1alpha1
kind: DPUVPC
metadata:
  name: testvpc
  namespace: default
spec:
  tenant: foo
  isolationClassName: ovn.vpc.dpu.nvidia.com
  interNetworkAccess: true
  nodeSelector: {}
---
apiVersion: vpc.dpu.nvidia.com/v1alpha1
kind: DPUVirtualNetwork
metadata:
  name: testnet1
  namespace: default
spec:
  vpcName: testvpc
  nodeSelector: {}
  type: Bridged
  externallyRouted: true
  masquerade: true
  bridgedNetwork:
    ipam:
      ipv4:
        dhcp: true
        subnet: 192.178.0.0/16
---
apiVersion: vpc.dpu.nvidia.com/v1alpha1
kind: DPUVirtualNetwork
metadata:
  name: testnet2
  namespace: default
spec:
```

```
  vpcName: testvpc
  nodeSelector: {}
  type: Bridged
  externallyRouted: true
  masquerade: true
  bridgedNetwork:
    ipam:
      ipv4:
        dhcp: true
        subnet: 192.188.0.0/16
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceInterface
metadata:
  name: testnet1-vf2
  namespace: default
spec:
  template:
    spec:
      nodeSelector:
        matchLabels:
          vpc.dpu.nvidia.com/network: testnet1
      template:
        spec:
          interfaceType: vf
          vf:
            pfID: 0
            vfID: 2
            parentInterfaceRef: ""
            virtualNetwork: testnet1
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceInterface
metadata:
  name: testnet2-vf2
  namespace: default
```

```yaml
spec:
  template:
    spec:
      nodeSelector:
        matchLabels:
          vpc.dpu.nvidia.com/network: testnet2
      template:
        spec:
          interfaceType: vf
          vf:
            pfID: 0
            vfID: 2
            parentInterfaceRef: ""
            virtualNetwork: testnet2
```

3. Two VPCs, each with a Single DPUVirtualNetwork and a Single DPUServiceInterface Targeting Separate DPU Nodes

This example associates two VPCs with different subsets of nodes. Each VPC has a single DPUVirtualNetwork. Each DPUVirtualNetwork is associated with a single DPUServiceInterface. This example corresponds to a multi-tenant environment where different sets of nodes belong to different tenants (VPCs).

The topology consists of:

1. Two DPUVPCs: `redvpc`, `bluevpc`
2. Two DPUVirtualNetworks: `rednet`, `bluenet`
3. Two DPUServiceInterfaces: `rednet-vf2`, `bluenet-vf2`

DPU nodes should be labeled with a distinct label to allow the association to a specific VPC.

In this example, we assume two groups of DPU nodes are labeled with `vpc.dpu.nvidia.com/tenant=red` and `vpc.dpu.nvidia.com/tenant=blue` respectively. Nodes within the same VPC can communicate with each other (E/W) in an isolated manner, as well as reach the external network behind NAT (N/S).

Each VPC and its resources can be created in their own namespace. This example uses the `default` namespace for both.

VPC Example Topology

```
---
apiVersion: vpc.dpu.nvidia.com/v1alpha1
kind: DPUVPC
metadata:
  name: redvpc
  namespace: default
spec:
  tenant: red
  isolationClassName: ovn.vpc.dpu.nvidia.com
  interNetworkAccess: true
  nodeSelector:
    matchLabels:
      vpc.dpu.nvidia.com/tenant: red
---
apiVersion: vpc.dpu.nvidia.com/v1alpha1
kind: DPUVPC
metadata:
  name: bluevpc
  namespace: default
spec:
  tenant: blue
  isolationClassName: ovn.vpc.dpu.nvidia.com
  interNetworkAccess: true
  nodeSelector:
    matchLabels:
      vpc.dpu.nvidia.com/tenant: blue
---
apiVersion: vpc.dpu.nvidia.com/v1alpha1
kind: DPUVirtualNetwork
metadata:
  name: bluenet
  namespace: default
spec:
  vpcName: bluevpc
  nodeSelector:
```

```
    matchLabels:
      vpc.dpu.nvidia.com/tenant: blue
  type: Bridged
  externallyRouted: true
  masquerade: true
  bridgedNetwork:
    ipam:
      ipv4:
        dhcp: true
        subnet: 192.178.0.0/16
---
apiVersion: vpc.dpu.nvidia.com/v1alpha1
kind: DPUVirtualNetwork
metadata:
  name: rednet
  namespace: default
spec:
  vpcName: redvpc
  nodeSelector:
    matchLabels:
      vpc.dpu.nvidia.com/tenant: red
  type: Bridged
  externallyRouted: true
  masquerade: true
  bridgedNetwork:
    ipam:
      ipv4:
        dhcp: true
        subnet: 192.178.0.0/16
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceInterface
metadata:
  name: rednet-vf2
  namespace: default
spec:
```

```
  template:
    spec:
      nodeSelector:
        matchLabels:
          vpc.dpu.nvidia.com/tenant: red
      template:
        spec:
          interfaceType: vf
          vf:
            pfID: 0
            vfID: 2
            parentInterfaceRef: ""
            virtualNetwork: rednet
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceInterface
metadata:
  name: bluenet-vf2
  namespace: default
spec:
  template:
    spec:
      nodeSelector:
        matchLabels:
          vpc.dpu.nvidia.com/tenant: blue
      template:
        spec:
          interfaceType: vf
          vf:
            pfID: 0
            vfID: 2
            parentInterfaceRef: ""
            virtualNetwork: bluenet
```

Monitoring and Troubleshooting

Resource Status

Each VPC resource can be queried for its status, which contains a set of conditions depicting the readiness of each resource.

DPUVPC

```yaml
apiVersion: vpc.dpu.nvidia.com/v1alpha1
kind: DPUVPC
metadata:
  ...
spec:
  ...
status:
  conditions:
  - lastTransitionTime: "2025-06-17T13:51:16Z"
    message: ""
    observedGeneration: 1
    reason: Success
    status: "True"
    type: Ready
  - lastTransitionTime: "2025-06-17T13:51:16Z"
    message: ""
    observedGeneration: 1
    reason: Success
    status: "True"
    type: DPUNodesReconciled
  - lastTransitionTime: "2025-06-17T13:51:16Z"
    message: ""
    observedGeneration: 1
    reason: Success
    status: "True"
    type: DPUVPCReconciled
  - lastTransitionTime: "2025-06-17T13:51:16Z"
    message: ""
    observedGeneration: 1
    reason: Success
    status: "True"
    type: PrerequisitesReady
  - lastTransitionTime: "2025-06-17T13:51:16Z"
    message: ""
    observedGeneration: 1
```

```
    reason: Success
    status: "True"
    type: TopologyReconciled
  observedGeneration: 1
  virtualNetworks:
  - name: rednet
```

The following conditions are used to indicate the status of the VPC:

- `Ready`: Overall readiness of the VPC resource.
- `DPUVPCReconciled`: VPC has been successfully reconciled.
- `TopologyReconciled`: Network topology has been successfully reconciled.
- `DPUNodesReconciled`: DPU nodes have been successfully reconciled with the VPC configuration.
- `PrerequisitesReady`: All required prerequisites for the VPC are in place.

DPUVirtualNetwork

```yaml
apiVersion: vpc.dpu.nvidia.com/v1alpha1
kind: DPUVirtualNetwork
metadata:
  ...
spec:
  ...
status:
  conditions:
  - lastTransitionTime: "2025-06-17T13:51:14Z"
    message: ""
    observedGeneration: 1
    reason: Success
    status: "True"
    type: Ready
  - lastTransitionTime: "2025-06-17T13:51:14Z"
    message: ""
    observedGeneration: 1
    reason: Success
    status: "True"
    type: DPUVirtualNetworkReconciled
  - lastTransitionTime: "2025-06-17T13:51:09Z"
    message: ""
    observedGeneration: 1
    reason: Success
    status: "True"
    type: PrerequisitesReady
  observedGeneration: 1
```

The following conditions are used to indicate the status of the virtual network:

- `Ready`: Overall readiness of the virtual network resource.
- `DPUVirtualNetworkReconciled`: Virtual network has been successfully reconciled.

- `PrerequisitesReady`: All required prerequisites for the virtual network are in place.

DPUServiceInterface

> **Note**
>
> Due to a DPF limitation, the overall readiness of a DPUServiceInterface does not reflect the readiness of the individual ServiceInterfaces for each DPU. Refer to the ServiceInterface section below.

```yaml
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceInterface
  ...
spec:
  ...
status:
  conditions:
  - lastTransitionTime: "2025-06-17T13:51:34Z"
    message: ""
    observedGeneration: 2
    reason: Success
    status: "True"
    type: Ready
  - lastTransitionTime: "2025-06-17T13:51:19Z"
    message: ""
    observedGeneration: 2
    reason: Success
    status: "True"
    type: ServiceInterfacePreReqsReady
  - lastTransitionTime: "2025-06-17T13:51:34Z"
    message: ""
    observedGeneration: 2
    reason: Success
    status: "True"
    type: ServiceInterfaceSetReady
  - lastTransitionTime: "2025-06-17T13:51:19Z"
    message: ""
    observedGeneration: 2
    reason: Success
    status: "True"
    type: ServiceInterfaceSetReconciled
  observedGeneration: 2
```

The following conditions are used to indicate the status of the DPU service interface:

- `Ready`: Overall readiness of the DPU service interface resource.
- `ServiceInterfaceSetReady`: The service interface set in the underlying infrastructure is ready.
- `ServiceInterfaceSetReconciled`: The service interface set in the underlying infrastructure has been successfully reconciled.
- `ServiceInterfacePreReqsReady`: All required prerequisites for the service interface are in place.

ServiceInterface

In some cases, it is necessary to understand the status of the service interface in the underlying infrastructure, for example, when a specific VF/PF is not connected to a virtual network.

In such cases, the user should query the DPU Kubernetes cluster for individual ServiceInterface CRs to understand their overall status.

Relevant fields are shown in the example below. Checking the conditions of the object should generally be sufficient.

```yaml
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: ServiceInterface
metadata:
  annotations:
    ovn.vpc.dpu.nvidia.com/lsp-connected: "true"
    ovn.vpc.dpu.nvidia.com/lsp-mac-address: 5a:19:ad:cf:61:85
spec:
  interfaceType: vf
  node: worker2-0000-c8-00
  vf:
    parentInterfaceRef: -worker2-0000-c8-00
    pfID: 0
    vfID: 2
    virtualNetwork: bluenet
status:
  conditions:
  - lastTransitionTime: "2025-06-17T13:51:40Z"
    message: ""
    observedGeneration: 1
    reason: Success
    status: "True"
    type: Ready
  - lastTransitionTime: "2025-06-17T13:51:40Z"
    message: ""
    observedGeneration: 1
    reason: Success
    status: "True"
    type: ServiceInterfaceReconciled
  observedGeneration: 1
```

The following conditions are used to indicate the status of the service interface:

- `Ready`: Overall readiness of the service interface resource.

- `ServiceInterfaceReconciled`: The service interface has been successfully reconciled.

Check Status of VPC resources using dpfctl

Use `dpfctl` to get a quick view of VPC resources:

```
$ kubectl -n dpf-operator-system exec deploy/dpf-operator-
controller-manager -- /dpfctl describe dpuvpcs --show-
conditions=all
NAME
NAMESPACE            STATUS  REASON   SINCE  MESSAGE
DPFOperatorConfig/dpfoperatorconfig                      dpf-
operator-system
             Ready
True    Success  36m
             ImagePullSecretsReconciled
True    Success  6d
             SystemComponentsReady
True    Success  36m
             SystemComponentsReconciled
True    Success  5d7h
  DPUVPCs
    DPUVPC/bluevpc                                       default
                 Ready
True    Success  25m
                 DPUNodesReconciled
True    Success  25m
                 DPUVPCReconciled
True    Success  25m
                 PrerequisitesReady
True    Success  25m
                 TopologyReconciled
True    Success  25m
      DPUVirtualNetworks
        DPUVirtualNetwork/bluenet                        default
                     Ready
True    Success  25m
                     DPUVirtualNetworkReconciled
True    Success  25m
                     PrerequisitesReady
True    Success  25m
```

```
          DPUServiceInterfaces
            DPUServiceInterface/bluenet-vf2                default
                          Ready
True    Success  24m
                          ServiceInterfacePreReqsReady
True    Success  25m
                          ServiceInterfaceSetReady
True    Success  24m
                          ServiceInterfaceSetReconciled
True    Success  25m
    DPUVPC/redvpc                                          default
                  Ready
True    Success  25m
                  DPUNodesReconciled
True    Success  25m
                  DPUVPCReconciled
True    Success  25m
                  PrerequisitesReady
True    Success  25m
                  TopologyReconciled
True    Success  25m
      DPUVirtualNetworks
        DPUVirtualNetwork/rednet                           default
                      Ready
True    Success  25m
                      DPUVirtualNetworkReconciled
True    Success  25m
                      PrerequisitesReady
True    Success  25m
          DPUServiceInterfaces
            DPUServiceInterface/rednet-vf2                 default
                          Ready
True    Success  24m
                          ServiceInterfacePreReqsReady
True    Success  25m
```

```
                                ServiceInterfaceSetReady
True    Success  24m
                                ServiceInterfaceSetReconciled
True    Success  25m
```

Default route for VPC VirtualNetworks

Problems

- ServiceInterfaces attached to different VirtualNetworks within the same VPC cannot reach each other.
- ServiceInterfaces trying to reach the external network do not use the VPC interface.

In both cases traffic egresses via the primary cluster network instead of the VPC gateway.

Assumptions: The VPC is correctly configured, and the `DPUVPC` resource is configured with `spec.interNetworkAccess: true` and `DPUVirtualNetwork` resource is configured with `spec.externallyRouted: true`.

Cause

The primary network's default route has equal or higher priority than the VPC network's default route (its metric is less than or equal to the VPC route's metric), so traffic prefers the primary network and bypasses the VPC gateway.

Fix

Prefer the VPC default route. The VPC route cannot set a metric and defaults to 0, so raise the primary network's default-route metric above 0 (e.g., 100).

Check

Connect to the worker and check default routes:

```
ip route show

default via 10.100.0.1 dev enp202s0f0np0 # VPC related interface
default via 10.0.110.254 dev eno1 proto static metric 100 # The host's
cluster network interface
10.0.110.0/24 dev eno1 proto kernel scope link src 10.0.110.31
10.100.0.0/16 dev enp202s0f0np0 proto kernel scope link src 10.100.0.2
```

Ensure the VPC default route has default metric value 0 and the primary default route has a higher metric (e.g., 100).

Verify

Verify you have the VPC default route with higher priority as described in the Fix.

- ServiceInterfaces on different VirtualNetworks within the same VPC can reach each other.
- ServiceInterfaces targeting the external network use the VPC interface as their default route.
- Traffic uses the VPC gateway (e.g., traceroute shows the VPC gateway as the first hop).

Common Issues

VPC Not Ready:

- Verify the IsolationClass exists and is properly configured.
- Ensure OVN Central is accessible.

Infrastructure Network Connectivity Issues:

- Check OVN database connectivity (ports 30641, 30642).
- Validate VTEP and external network configuration.
- Verify physical network connectivity between DPU nodes on both VTEP and external networks.
- Validate the ServiceInterface was properly reconciled and is ready.

Network Connectivity Issues for host interfaces connected to VPC:

- Verify the specific ServiceInterface is ready.
- Ensure the target network interface (e.g., VF/PF netdev) has the same MAC address as in the ServiceInterface annotation `vpc.dpu.nvidia.com/lsp-mac-address`.
- Ensure the host interface is up and has an IP address (or assign it one using DHCP/manual).
- Ensure the MTU is set to 9000 for optimal performance.

DHCP Not Working:

- Confirm the virtual network has DHCP enabled (`spec.bridgedNetwork.ipam.ipv4.dhcp: true`).
- Check for IP conflicts in subnet ranges.
- Ensure the target ServiceInterface is ready.

Limitations

- **Production Use**: Not recommended for production use.
- **Network Types**: Only bridged networks are supported.
- **IPv6**: Limited IPv6 support.
- **High Availability**: High availability (HA) for OVN Central is supported, but it is not recommended for production deployments.

- **DPUVirtualNetwork nodeSelector**: The DPUVirtualNetwork `spec.nodeSelector` currently has no effect.
- **OVN Central connection is not secure**: Currently only `tcp` transport is supported.
- **Disabling Masquerade is not supported**: The DPUVirtualNetwork `spec.masquerade` should not be set to `false` as this use-case is not currently supported.

Advanced Use-cases

Service interface with unknown MAC address

There are cases where a service interface needs to have an unknown MAC address so that it can transmit packets with any source MAC (e.g. a network service transmitting packets from arbitrary L2 sources).

To support this case, add the following annotation to your DPUServiceInterface: `ovn.vpc.dpu.nvidia.com/unknown-mac: true`

Example

```
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceInterface
metadata:
  ...
spec:
  template:
    spec:
      template:
        metadata:
          annotations:
            ovn.vpc.dpu.nvidia.com/unknown-mac: "true"
    ...
```

Connecting a DPUServiceInterface of type Service to VPC

Similarly to DPUServiceInterfaces of types `pf` and `vf`, it is possible to connect a DPUServiceInterface of type `service`.

The interface of the DPU Service can then be assigned an IP from the VPC via DHCP. Since the DPU service is ultimately a Kubernetes Pod, this requires the use of the `dhcp` IPAM CNI plugin when setting up the secondary network for the DPU Service interface.

To achieve this, direct access to the DPU Cluster is currently required.

1. Create a custom `NetworkAttachmentDefinition` in the DPU cluster:

```
apiVersion: k8s.cni.cncf.io/v1
kind: NetworkAttachmentDefinition
metadata:
  annotations:
    k8s.v1.cni.cncf.io/resourceName: nvidia.com/bf_sf
  name: mybrint-vpc
  namespace: dpf-operator-system
spec:
  config: |-
    {
      "bridge": "br-int",
      "cniVersion": "0.4.0",
      "interface_type": "dpdk",
      "mtu": 1500,
      "type": "ovs",
      "ipam": {
        "type": "dhcp",
        "daemonSocketPath": "/run/vpc/cni/dhcp.sock"
      }
    }
```

2. Create your DPUService and DPUServiceInterface as usual, providing a reference to this network in `spec.serviceDaemonSet.annotations`:

```yaml
---
apiVersion: "svc.dpu.nvidia.com/v1alpha1"
kind: DPUServiceInterface
metadata:
  name: my-interface
  namespace: my-namespace
spec:
  template:
    spec:
      template:
        spec:
          interfaceType: service
          service:
              interfaceName: my-sf
              serviceID: my-service
              network: dpf-operator-system/mybrint-vpc
              virtualNetwork: testnet1
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUService
metadata:
  name: my-service
  namespace: my-namespace
spec:
  helmChart:
    source:
      ...
    values:
      ...
  serviceID: my-service
  serviceDaemonSet:
    resources:
      nvidia.com/bf_sf: 1
    annotations:
      k8s.v1.cni.cncf.io/networks: |
```

```
            [{"name": "mybrint-vpc", "namespace": "dpf-operator-system", "interface":
"my_sf"}]
```

Connecting a DPUServiceInterface of type Service to VPC with unknown MAC

Similarly to the previous section, this requires direct access to the DPU Cluster.

1. Create a custom `NetworkAttachmentDefinition` in the DPU cluster:

```
apiVersion: k8s.cni.cncf.io/v1
kind: NetworkAttachmentDefinition
metadata:
  annotations:
    k8s.v1.cni.cncf.io/resourceName: nvidia.com/bf_sf
  name: mybrint-vpc-nodhcp
  namespace: dpf-operator-system
spec:
  config: |-
    {
      "bridge": "br-int",
      "cniVersion": "0.4.0",
      "interface_type": "dpdk",
      "mtu": 1500,
      "type": "ovs"
    }
```

2. Create your DPUService and DPUServiceInterface as usual, providing a reference to this network in `spec.serviceDaemonSet.annotations`:

```yaml
---
apiVersion: "svc.dpu.nvidia.com/v1alpha1"
kind: DPUServiceInterface
metadata:
  name: my-interface
  namespace: my-namespace
spec:
  template:
    spec:
      template:
        metadata:
          annotations:
            ovn.vpc.dpu.nvidia.com/unknown-mac: "true"
        spec:
          interfaceType: service
          service:
              interfaceName: my-sf
              serviceID: my-service
              network: dpf-operator-system/mybrint-vpc-nodhcp
              virtualNetwork: testnet1
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUService
metadata:
  name: my-service
  namespace: my-namespace
spec:
  helmChart:
    source:
      ...
    values:
      resources:
        nvidia.com/bf_sf: 1
  serviceID: my-service
  serviceDaemonSet:
```

```
    annotations:
      k8s.v1.cni.cncf.io/networks: |
        [{"name": "mybrint-vpc", "namespace": "dpf-operator-system", "interface":
"my_sf"}]
```

VPC OVN Upgrade Guide

For zero-downtime VPC OVN upgrade guide refer to VPC OVN upgrade guide

OVN High Availability (HA) guide

> **Note**
>
> The OVN High Availability (HA) service is considered a tech preview and is not recommended for production use.

For OVN High Availability (HA) support guide refer to OVN HA guide

OVN High Availability (HA) guide

To achieve high availability (HA) for OVN, deploy the `ovn-central` DPUService with the `enableHA` option enabled.

0. Prerequisites

Requires at least three schedulable control-plane nodes for quorum.

Use `$HELM_REGISTRY_REPO_URL` and `TAG` to specify the Helm chart registry and your desired version respectively.

```
## The repository URL for the NVIDIA Helm chart registry.
## Usually this is the NVIDIA Helm NGC registry. For development
purposes, this can be set to a different repository.
export HELM_REGISTRY_REPO_URL=https://helm.ngc.nvidia.com/nvidia/doca
## The DPF TAG is the version of the OVN components which will be
deployed in this guide.
export TAG=v25.10.0
```

1. Deploy ovn-central DPUService

```
cat examples/dpuservice-ovn-central.yaml | envsubst | kubectl
apply -f -
```

This will deploy `ovn-central` DPUService with `enableHA` set to `true`, it will create 3 ovn-central pods on the control-plane nodes.

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUService
metadata:
  name: ovn-central
  namespace: dpf-operator-system
spec:
  deployInCluster: true
  helmChart:
    source:
      repoURL: $HELM_REGISTRY_REPO_URL
      version: $TAG
      chart: ovn-chart
    values:
      exposedPorts:
        ports:
          ovnnb: true
          ovnsb: true
      management:
        ovnCentral:
          enabled: true
          enableHA: true
          affinity:
            nodeAffinity:
              requiredDuringSchedulingIgnoredDuringExecution:
                nodeSelectorTerms:
                  - matchExpressions:
                      - key: "node-role.kubernetes.io/master"
                        operator: Exists
                  - matchExpressions:
                      - key: "node-role.kubernetes.io/control-plane"
                        operator: Exists
          tolerations:
            - key: node-role.kubernetes.io/master
              operator: Exists
```

```
            effect: NoSchedule
          - key: node-role.kubernetes.io/control-plane
            operator: Exists
            effect: NoSchedule
```

VPC OVN upgrade guide

To achieve a zero-downtime VPC OVN upgrade, upgrade the services in this order: vpc-ovn-node ovn-controller, ovn-central, then vpc-ovn-controller.

Assuming the deployment of all the DPF VPC OVN componenets are done based on the following guide. VPC OVN Deployment

0. Required variables

Use $TARGET_TAG as your desired version.

```
## The TARGET_TAG is the desired version of the DPF VPC OVN
components which will be upgraded.
export TARGET_TAG=v25.10.0
```

1. Upgrade vpc-ovn-node service

Patch the `DPUServiceTemplate` of the vpc-ovn-node service with the upgraded chart version.

```
kubectl patch dpuservicetemplate vpc-ovn-node -n dpf-operator-
system --type=json --patch="[{\"op\": \"replace\", \"path\":
\"/spec/helmChart/source/version\", \"value\": \"$TARGET_TAG\"}]"
```

Ensure the `DPUDeployment` is Ready.

```
kubectl wait --for=condition=ready --namespace dpf-operator-system
dpudeployment vpc-ovn
```

2. Upgrade ovn-controller service

Patch the `DPUServiceTemplate` of the ovn-controller service with the upgraded chart version.

```
kubectl patch dpuservicetemplate ovn-controller -n dpf-operator-
system --type=json --patch="[{\"op\": \"replace\", \"path\":
\"/spec/helmChart/source/version\", \"value\": \"$TARGET_TAG\"}]"
```

Ensure the `DPUDeployment` is Ready.

```
kubectl wait --for=condition=ready --namespace dpf-operator-system
dpudeployment vpc-ovn
```

3. Upgrade ovn-central service

Patch the `DPUServiceTemplate` of the ovn-central service with the upgraded chart version.

```
kubectl patch dpuservicetemplate ovn-central -n dpf-operator-
system --type=json --patch="[{\"op\": \"replace\", \"path\":
\"/spec/helmChart/source/version\", \"value\": \"$TARGET_TAG\"}]"
```

Ensure the `DPUDeployment` is Ready.

```
kubectl wait --for=condition=ready --namespace dpf-operator-system
dpudeployment vpc-ovn
```

4. Upgrade vpc-ovn-controller service

Patch the `DPUServiceTemplate` of the vpc-ovn-controller service with the upgraded chart version.

```
kubectl patch dpuservicetemplate vpc-ovn-controller -n dpf-
operator-system --type=json --patch="[{\"op\": \"replace\", \"path\":
\"/spec/helmChart/source/version\", \"value\": \"$TARGET_TAG\"}]"
```

Ensure the `DPUDeployment` is Ready.

```
kubectl wait --for=condition=ready --namespace dpf-operator-system
dpudeployment vpc-ovn
```

Host Trusted Multi-DPU support OVN-Kubernetes and HBN Services

> **Note**
>
> **Tech Preview**: This feature is currently in technical preview and may be subject to changes in future releases.

This guide describes how to use the `dpuSelector` to target particular DPUs. This approach supports multiple DPUs and distributes OVN-Kubernetes, HBN, BlueMan and

DTS services across them, providing more granular control over which DPUs run specific services and allowing for better resource allocation, service isolation, and multi-DPU scalability.

Overview

OVN-Kubernetes serves as the primary CNI in host clusters. To achieve hardware acceleration for host workload pods, OVN-Kubernetes is split into two components: a host-side component and a DPU-side component. The acceleration workflow operates as follows:

1. The OVN-Kubernetes resource injector (running on the host) adds VF resource requests to host workload pods based on the `dpf-ovn-kubernetes` NetworkAttachmentDefinition (NAD).

2. The SR-IOV Device Plugin allocates a VF from the host's physical function and attaches it to the pod.

3. The OVN-Kubernetes host component communicates VF information to the DPU component via pod annotations.

4. The OVN-Kubernetes DPU component identifies the corresponding VF representor and adds it to the OVS bridge on the DPU, enabling hardware acceleration.

Multi-DPU Constraint

In multi-DPU setups (one host worker node with two or more DPUs), both OVN-Kubernetes and HBN (Host-Based Networking) must run on the same single DPU. OVN-Kubernetes manages a single OVS bridge per host, so running it on multiple DPUs would create conflicting bridge configurations. When deployed together in the HBN+OVNK use case, HBN must be colocated with OVN-Kubernetes because they are integrated through service chains, which operate on a single DPU.

To maintain hardware acceleration in multi-DPU environments, the `SRIOVNetworkNodePolicy` must be configured to target only the Physical Function (PF) of the DPU where OVN-Kubernetes runs (via the `pfNames` field, e.g., `$DPU_P0#2-45`). This configuration ensures that:

- The SR-IOV Device Plugin exposes VFs exclusively from the DPU running OVN-Kubernetes

- When the resource injector adds VF requests to pods, the SR-IOV Device Plugin allocates VFs only from that DPU

- The OVN-Kubernetes DPU component can properly add VF representors to OVS on the same DPU

Using dpuSelector for Multi-DPU Deployments

The OVN Kubernetes with Host Based Networking guide uses only `nodeSelector` to target nodes with DPUs, which deploys all services (OVN-Kubernetes, HBN, DTS, Blueman) to all DPUs on those nodes. This creates conflicts when multiple DPUs are present.

The `dpuSelector` approach solves this problem by enabling precise DPU targeting. It allows you to:

1. **Run OVN-Kubernetes and HBN together on a single DPU** (critical requirement - they are integrated through service chains)

2. Deploy other services (e.g., DTS, Blueman) on different DPUs to distribute workload

3. Target specific DPUs based on their device characteristics (e.g., PF names, PCI addresses)

4. Achieve better resource isolation and performance optimization across multiple DPUs

Prerequisites

This guide is based on the OVN Kubernetes with Host Based Networking use case. Follow that guide until you reach the DPU Provisioning and Service Installation section where you are about to deploy the DPUDeployment. At that point, use this guide to modify the DPUDeployment for multi-DPU support instead of using the original single-DPU approach.

Tip

SR-IOV Device Plugin and OVN-Kubernetes Configuration Requirements:

In multi-DPU setups, you must ensure that both the SR-IOV device plugin and OVN-Kubernetes are correctly configured to use the appropriate DPU P0 interface.

- Set the `DPU_P0` environment variable to the name of the P0 interface on the DPU where OVN-Kubernetes will run.
- Set the `DPU_P0_VF1` environment variable to the name of the VF1 interface on the same DPU.

1. **SR-IOV Device Plugin Configuration**: The `SriovNetworkNodePolicy` must be configured to target the correct p0 interface of the DPU that will run OVN-Kubernetes. In the HBN-OVNK guide, this is configured via the `pfNames` field (e.g., `$DPU_P0#2-45`) in the `SriovNetworkNodePolicy` resource.
2. **OVN-Kubernetes Configuration**: The OVN-Kubernetes installation must be configured with the correct `gatewayOpts` parameter pointing to the p0 interface of the DPU that will run OVN-Kubernetes (e.g., `--gateway-interface=$DPU_P0`), and the `nodeMgmtPortNetdev` must be set to the correct VF1 interface (e.g., `$DPU_P0_VF1`).

For detailed configuration examples, refer to the OVN Kubernetes with Host Based Networking guide, specifically the SR-IOV Network Operator Policy and OVN-Kubernetes Helm values sections.

Key Differences

Node Selector

```
dpuSets:
- nameSuffix: "dpuset1"
  nodeSelector:
    matchLabels:
      feature.node.kubernetes.io/dpu-enabled: "true"
```

This targets all nodes that have the `feature.node.kubernetes.io/dpu-enabled: "true"` label, and all services run on all DPUs on those nodes.

Both Node Selector and DPU Selector

```
dpuSets:
- nameSuffix: "dpuset1"
  nodeSelector:
    matchLabels:
      feature.node.kubernetes.io/dpu-enabled: "true"
  dpuSelector:
    provisioning.dpu.nvidia.com/dpudevice-pf0-name: ens1f0np0
```

This targets specific DPUs based on their device characteristics, allowing for more precise service placement and multi-DPU distribution.

DPUDevice Labels

DPUDevices are automatically labeled with device-specific information that can be used in `dpuSelector`. The available labels include:

- `provisioning.dpu.nvidia.com/dpudevice-name`: The name of the DPUDevice

- `provisioning.dpu.nvidia.com/dpudevice-num-of-pfs`: The number of PFs on the DPU device
- `provisioning.dpu.nvidia.com/dpudevice-pciAddress`: The PCI address of the DPU device
- `provisioning.dpu.nvidia.com/dpudevice-pf0-name`: The name of PF0 on the DPU device
- `provisioning.dpu.nvidia.com/dpunode-name`: The name of the DPUNode the DPU is part of

Example Interface Names for Multi-DPU Setup

In the examples below, we use the following interface names to demonstrate multi-DPU deployment:

- `ens1f0np0`: Example name for the first port of DPU 1 (for OVN/HBN services)
- `ens2f0np0`: Example name for the first port of DPU 2 (for Blueman/DTS services)

Replace these with the actual interface names from your DPU devices. You can extend this pattern to support additional DPUs by creating more DPUDeployments with different dpuSelector values. Remember that OVN-Kubernetes should only run on one DPU per host.

Example: Multi-DPU Deployment with dpuSelector

This section shows how to modify the existing DPUDeployment from the OVN Kubernetes with Host Based Networking guide and create an additional one for multi-DPU support.

Step 1: Modify the Existing DPUDeployment for OVN and HBN Services

Instead of creating a new DPUDeployment, modify the existing one from the OVN-Kubernetes HBN guide. Simply add a `dpuSelector` to the existing `ovn-hbn` DPUDeployment:

> **Note**
>
> We removed the serviceChains section for simplicity. You should keep the serviceChains section if you have defined any service chains in your original DPUDeployment.

Modified DPUDeployment:

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUDeployment
metadata:
  name: ovn-hbn
  namespace: dpf-operator-system
spec:
  dpus:
    bfb: bf-bundle
    flavor: hbn-ovn
    dpuSets:
      - nameSuffix: "dpuset1"
        nodeSelector:
          matchLabels:
            feature.node.kubernetes.io/dpu-enabled: "true"
        dpuSelector:
          provisioning.dpu.nvidia.com/dpudevice-pf0-name:
ens1f0np0
  services:
    ovn:
      serviceTemplate: ovn
      serviceConfiguration: ovn
    hbn:
      serviceTemplate: hbn
      serviceConfiguration: hbn
  serviceChains:
...
```

Step 2: Create Additional DPUDeployment for Blueman and DTS Services

Create a **new, additional** DPUDeployment for Blueman and DTS services that targets the second DPU. This new DPUDeployment must use a different `dpuSelector` to target the

second DPU's PF0 interface.

Creating a Separate DPUFlavor

The second DPU requires a slightly different `DPUFlavor` without the `hostNetworkInterfaceConfigs` section.

> **Note**
>
> **When to use `hostNetworkInterfaceConfigs`:**
>
> The `hostNetworkInterfaceConfigs` section (which configures DHCP and MTU settings for host-side network interfaces) is only needed for the DPU that handles host networking (the one running OVN-Kubernetes and HBN). Since the second DPU only runs Blueman and DTS services, it does not need to configure the host network interfaces, and therefore the `hostNetworkInterfaceConfigs` section should be omitted.

To create the new flavor, modify the existing `dpuflavor-hbn-ovn.yaml` file by removing the `hostNetworkInterfaceConfigs` section as shown below and save it as a new file (e.g., `dpuflavor-other-services.yaml`) with a new `metadata.name`:

> **Tip**
>
> The diff below shows the path from the documentation repository for reference. Use your actual file location when making these changes.

```
--- a/docs/public/user-guides/host-trusted/use-cases/hbn-ovnk/manifests/05-dpudeployment-installation/dpuflavor-hbn-ovn.yaml
+++ b/docs/public/user-guides/host-trusted/use-cases/hbn-ovnk/manifests/05-dpudeployment-installation/dpuflavor-hbn-ovn.yaml
@@ -2,7 +2,7 @@
 apiVersion: provisioning.dpu.nvidia.com/v1alpha1
 kind: DPUFlavor
 metadata:
-  name: hbn-ovn
+  name: hbn-ovn-other-services
   namespace: dpf-operator-system
 spec:
   grub:
@@ -74,11 +74,6 @@ spec:
     - UPDATE_DPU_OS=yes
     - WITH_NIC_FW_UPDATE=yes
-  hostNetworkInterfaceConfigs:
-    - portNumber: 0
-      dhcp: true
-      mtu: 1500
-
   configFiles:
   - path: /etc/mellanox/mlnx-bf.conf
     operation: override
```

Creating the DPUDeployment

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUDeployment
metadata:
  name: blueman-dts
  namespace: dpf-operator-system
spec:
  dpus:
    bfb: bf-bundle
    flavor: hbn-ovn-other-services
    dpuSets:
      - nameSuffix: "dpuset1"
        nodeSelector:
          matchLabels:
            feature.node.kubernetes.io/dpu-enabled: "true"
        dpuSelector:
          provisioning.dpu.nvidia.com/dpudevice-pf0-name:
ens2f0np0
  services:
    dts:
      serviceTemplate: dts
      serviceConfiguration: dts
    blueman:
      serviceTemplate: blueman
      serviceConfiguration: blueman
```

Deployment Steps

1. Prepare Configuration Files

Create the modified DPUDeployment files based on the examples above.

2. Modify the Existing DPUDeployment

Update the existing `ovn-hbn` DPUDeployment to use `dpuSelector` instead of `nodeSelector`:

```
# Apply the modified DPUDeployment (replace the existing one)
kubectl apply -f dpudeployment-ovn-hbn-modified.yaml
```

3. Deploy Additional DPUDeployment for Blueman and DTS

Create and apply the new DPUDeployment for Blueman and DTS services:

```
kubectl apply -f dpudeployment-blueman-dts.yaml
```

4. Verify Deployment

You can monitor prevalidation status using either dpfctl or kubectl:

Using dpfctl (recommended):

```
# Check overall DPF status
dpfctl describe all

# Check specific conditions for troubleshooting
dpfctl describe all --show-conditions=all
```

Using kubectl:

```
# Check DPUDeployments
kubectl get dpudeployments -n dpf-operator-system

# Check DPUServices
kubectl get dpuservices -n dpf-operator-system
```

Host Trusted with Non Kubernetes Workers

This section covers an **advanced configuration** of the Host Trusted use case for environments where worker nodes are not part of a Kubernetes cluster.

Before proceeding with the advanced configuration, ensure you have installed DPF following the Host Trusted Getting Started Guide.

This documentation is organized into the following sections:

- Host Installation for Non-Kubernetes Environments - Complete step-by-step installation guide for non-Kubernetes workers
- Custom Provisioning with DMS CA - Advanced configuration for custom certificate authority

Enable Custom Certificate Authority for DMS

This document provides instructions on how to configure DPF to use a custom Certificate Authority (CA) for mutual TLS (mTLS) authentication between the provisioning controller and DOCA Management Service (DMS) in non-Kubernetes(only Kubernetes control plane) system. To enable mTLS, the user needs to create a Kubernetes Secret containing the required certificates and keys, and configure the `DPFOperatorConfig` to use this Secret.

Step 1: Prepare Certificates and Keys

Ensure have the following files ready:

- **Server Certificate**: A PEM-encoded certificate for the server (tls.crt)
- **Private Key**: A PEM-encoded private key corresponding to the server certificate (tls.key)
- **CA Certificate**: A PEM-encoded certificate for the custom Certificate Authority (ca.crt)

Step 2: Create a Kubernetes Secret

Create a Kubernetes Secret of type `kubernetes.io/tls` that includes the `tls.crt`, `tls.key`, and `ca.crt` fields in `dpf-operator-system` namespace.

```
kubectl create secret tls custom-ca-secret  --cert=tls.crt --
key=tls.key --certificate-authority=ca.crt -n dpf-operator-system
```

Step 3: Configure DPFOperatorConfig

When creating or updating the `DPFOperatorConfig`, specify the name of the Secret created in the previous step.

Example DPFOperatorConfig Configuration

In the `DPFOperatorConfig` configuration, set the `customCASecretName` field to the name of the Secret (e.g., custom-ca-secret):

```
apiVersion: operator.dpu.nvidia.com/v1alpha1
kind: DPFOperatorConfig
metadata:
  name: dpfoperatorconfig
  namespace: dpf-operator-system
spec:
  provisioningController:
    bfbPVCName: "bfb-pvc"
    customCASecretName: "custom-ca-secret"
  kamajiClusterManager: {}
```

Host Installation for non-Kubernetes Environments

Installation Steps

1. Install DPF

Select a use case from the host trusted use cases and complete all prerequisite steps before provisioning.

> Note: The DPU Detector is not used in non-Kubernetes environments. After creating `DPFOperatorConfig`, you can ignore any related errors, or disable the DPU Detector by setting `dpuDetector.disable=true`.

2. Create Bootstrap Token

For security reasons, it's recommended that the host agent join the cluster using a bootstrap token. The following commands create a bootstrap token that expires in 24 hours and use that token to generate a kubeconfig.

Run the following commands on any of the **control plane nodes**

```
export TOKEN_SUFFIX=$(cat /dev/urandom | tr -dc 'a-z0-9' | fold -w 6 | head
-n 1)
export TOKEN_SECRET=$(cat /dev/urandom | tr -dc 'a-z0-9' | fold -w 16 | head
-n 1)
export CLUSTER_NAME="cluster.local"
export BOOTSTRAP_TOKEN="$TOKEN_SUFFIX.$TOKEN_SECRET"
export ADMIN_KUBECONFIG_FILE=${HOME}/.kube/config
export BOOTSTRAP_KUBECONFIG_FILE="bootstrap.kubeconfig"

# create a token that expires in 24 hours
kubectl create secret generic bootstrap-token-$TOKEN_SUFFIX \
  --type=bootstrap.kubernetes.io/token \
  --namespace=kube-system \
  --from-literal=description="Bootstrap token for host-agent" \
  --from-literal=token-id=$TOKEN_SUFFIX \
  --from-literal=token-secret=$TOKEN_SECRET \
  --from-literal=expiration=$(date -u -d '+24 hours' +%Y-%m-
%dT%H:%M:%SZ) \
  --from-literal=usage-bootstrap-authentication=true \
  --from-literal=usage-bootstrap-signing=true \
  --from-literal=auth-extra-groups=system:bootstrappers:dpf:host-
agent

# create a kubeconfig file out of the token and the admin kubeconfig
cp ${ADMIN_KUBECONFIG_FILE} $BOOTSTRAP_KUBECONFIG_FILE
kubectl config unset --kubeconfig=$BOOTSTRAP_KUBECONFIG_FILE
current-context
kubectl config unset --kubeconfig=$BOOTSTRAP_KUBECONFIG_FILE users
kubectl config unset --kubeconfig=$BOOTSTRAP_KUBECONFIG_FILE
contexts
kubectl config set-credentials bootstrap-user \
  --token=$BOOTSTRAP_TOKEN \
  --kubeconfig=$BOOTSTRAP_KUBECONFIG_FILE
kubectl config set-context bootstrap-context \
  --cluster=$CLUSTER_NAME \
```

```
  --user=bootstrap-user \
  --kubeconfig=$BOOTSTRAP_KUBECONFIG_FILE
kubectl config use-context bootstrap-context \
  --kubeconfig=$BOOTSTRAP_KUBECONFIG_FILE
```

3. Distribute The Kubeconfig File

Distribute the generated kubeconfig file under /var/lib/dpf/hostagent/ of each worker node.

4. Install DOCA and containerd

Run the following commands on worker host.

```
# Set DOCA repository URL
export DOCA_URL="https://linux.mellanox.com/public/repo/doca/3.2.1/ubuntu24.04/x86_64/"

# Add NVIDIA Mellanox GPG key
curl https://linux.mellanox.com/public/repo/doca/GPG-KEY-Mellanox.pub | sudo gpg --dearmor > /etc/apt/trusted.gpg.d/GPG-KEY-Mellanox.pub

# Add DOCA repository
echo "deb [signed-by=/etc/apt/trusted.gpg.d/GPG-KEY-Mellanox.pub] $DOCA_URL ./" |
sudo tee /etc/apt/sources.list.d/doca.list

# Update and install required packages
sudo apt-get update
sudo apt-get -y install doca-all doca-networking containerd
```

5. Pull The HostDriver Image From NGC

Run the following commands on worker host.

```
export IMAGE=<the hostdriver image>
export IMAGE_PULL_KEY=<your NGC key granted permission for pulling image>
sudo ctr image pull --user \$oauthtoken:$IMAGE_PULL_KEY  $IMAGE
```

6. Initialize Host Agent Flags

Before you run the following commands, you need to set the BFB_REGISTRY_ADDRESS with the same value as you set in DPFOperatorConfig.spec.provisioningController.registry

Run the following commands on worker host.

```
export BFB_REGISTRY_ADDRESS=<address of the bfb-registry>
export HOST_AGENT_DIR=/var/lib/dpf/hostagent
export HOST_AGENT_ENV_FILE=$HOST_AGENT_DIR/dpf-host-agent.env

sudo mkdir -p $HOST_AGENT_DIR
sudo tee $HOST_AGENT_ENV_FILE >/dev/null <<EOF
BFB_REGISTRY_ADDRESS="$BFB_REGISTRY_ADDRESS"
HOST_AGENT_CONTAINER_NAME="dpf-host-agent"
DMSD_CONTAINER_NAME="dpf-dmsd"
RUN_CONTAINER_ARGS="--rm --net-host --privileged --mount
type=bind,src=/var/run/dbus/system_bus_socket,dst=/var/run/dbus/system_bus_socket,options=rbind:rw --
mount type=bind,src=/tmp,dst=/tmp,options=rbind:rw --mount
type=bind,src=/etc/netplan,dst=/etc/netplan,options=rbind:rw --mount
type=bind,src=/run/systemd,dst=/run/systemd,options=rbind:rw --mount
type=bind,src=/run/udev,dst=/run/udev,options=rbind:rw  --mount
type=bind,src=/usr/lib/systemd/network,dst=/usr/lib/systemd/network,options=rbind:rw --mount
type=bind,src=/proc,dst=/proc,options=rbind:rw --mount type=bind,src=/dev,dst=/dev,options=rbind:rw --
mount type=bind,src=/sys,dst=/sys,options=rbind:rw --mount
type=bind,src=/lib/modules,dst=/lib/modules,options=rbind:ro --mount
type=bind,src=/var/lib/dpf,dst=/var/lib/dpf,options=rbind:rw $IMAGE"
EOF
```

7. Start rshim and DMS

Run the following commands on worker host.

```
export DMSD_SERVICE_FILE=/etc/systemd/system/dpf-dmsd.service
sudo tee $DMSD_SERVICE_FILE >/dev/null <<EOF
[Unit]
Description=DMS server
Wants=network.target
After=network-pre.target

[Service]
Type=exec
EnvironmentFile=$HOST_AGENT_ENV_FILE
ExecStartPre=-/usr/bin/ctr snapshot delete \$DMSD_CONTAINER_NAME
ExecStartPre=-/usr/bin/ctr container delete \$DMSD_CONTAINER_NAME
ExecStart=/usr/bin/ctr run \$RUN_CONTAINER_ARGS
\$DMSD_CONTAINER_NAME /hostagent rundms

TimeoutStopSec=30

Restart=on-failure
RestartSec=5

StartLimitIntervalSec=120
StartLimitBurst=5

[Install]
WantedBy=multi-user.target
EOF

sudo systemctl enable rshim
sudo systemctl start rshim
sudo systemctl enable dpf-dmsd
sudo systemctl start dpf-dmsd
```

8. Start Host Agent With Systemd and containerd

Run the following commands on worker host.

```
export HOST_AGENT_SERVICE_FILE=/etc/systemd/system/dpf-host-
agent.service
sudo tee $HOST_AGENT_SERVICE_FILE >/dev/null <<EOF
[Unit]
Description=DPF Host Agent
Wants=network.target
After=network-pre.target

[Service]
Type=exec
EnvironmentFile=$HOST_AGENT_ENV_FILE
ExecStartPre=-/usr/bin/ctr snapshot delete
\$HOST_AGENT_CONTAINER_NAME
ExecStartPre=-/usr/bin/ctr container delete
\$HOST_AGENT_CONTAINER_NAME
ExecStart=/usr/bin/ctr run \$RUN_CONTAINER_ARGS
\$HOST_AGENT_CONTAINER_NAME /hostagent serve --bootstrap-
kubeconfig=/var/lib/dpf/hostagent/bootstrap.kubeconfig --
kubeconfig=/var/lib/dpf/hostagent/kubeconfig --bfb-registry-
address=\${BFB_REGISTRY_ADDRESS} -v 3

TimeoutStopSec=30

Restart=on-failure
RestartSec=5

StartLimitIntervalSec=120
StartLimitBurst=5

[Install]
WantedBy=multi-user.target
EOF

sudo systemctl enable dpf-host-agent
```

```
sudo systemctl start dpf-host-agent
```

Verification

To verify the installation is working correctly, run the following commands on worker host.

```
# Check services status
sudo systemctl status rshim
sudo systemctl status dpf-dmsd
sudo systemctl status dpf-host-agent
```

External Host Reboot

After the worker node has rebooted, remove the `provisioning.dpu.nvidia.com/dpunode-external-reboot-required` annotation from the DPUNode using kubectl:

```
kubectl annotate dpunode <node-name>
provisioning.dpu.nvidia.com/dpunode-external-reboot-required-
```

This step is only required for hosts that are not managed by Kubernetes.

Observability

Introduction

This guide covers the steps to enable observability components such as Prometheus, Grafana, and kube-state-metrics for the operator. By default, these components are disabled in the Helm chart and need to be enabled explicitly.

Purpose of kube-state-metrics

`kube-state-metrics` is a service that listens to the Kubernetes API server and generates metrics about the state of the objects (such as deployments, nodes, pods, etc.) managed by Kubernetes. In the context of our operator, we deploy `kube-state-metrics` to expose detailed metrics for the custom resource definitions (CRDs) managed by the operator.

Enabling Observability Components

To enable Prometheus, Grafana and Kube-State-Metrics, you have to install the helm charts as described in the Helm Prerequisites guide.

Included Grafana Dashboards

We have preconfigured three Grafana dashboards to provide monitoring and insights into the operator and its controllers:

1. **DOCA Platform Framework State**: This dashboard provides a high-level overview of the operator and its controllers, highlighting key metrics such as resource status, condition states, and time to readiness.

2. **Controller Runtime Dashboard**: This dashboard provides detailed metrics and visualizations for the controllers, including information on reconciliation times, queue depths, and error rates.

3. **Kubernetes API Server Requests Dashboard**: This dashboard monitors the requests made to the Kubernetes API server, helping you to identify any performance bottlenecks or excessive API usage.

These dashboards are automatically deployed when Grafana is enabled. Once enabled, you can access them through the Grafana web UI under the "Dashboards" section.

Setting the Grafana Admin Password

You can set the Grafana admin password manually by configuring the `grafana.adminPassword` value in Grafana values file:

```
adminPassword: <your-password>
```

Alternatively, if you prefer Grafana to generate a custom password, leave `adminPassword` unset. After deployment, you can retrieve the autogenerated password using the following command:

```
kubectl -n dpf-operator-system get secret grafana -
ojsonpath='{.data.admin-password}' | base64 -d
```

This command fetches the password from the Kubernetes secret created for Grafana.

Note on Storage Solution

By default, Grafana and Prometheus use `hostPath` for storage. This is not recommended for production environments due to the potential for data loss and lack of scalability. You should configure a more reliable storage solution.

To change the storage solution, you need to modify the respective storage configurations in the `values.yaml` file:

Prometheus:

```
server:
  persistentVolume:
    enabled: true
    storageClass: <your-storage-class>
```

Grafana:

```
persistence:
  enabled: true
  storageClassName: <your-storage-class>
```

Make sure to replace `<your-storage-class>` with the appropriate storage class for your environment.

Secondary CNI support for HBN-OVNK use case

This section covers an **advanced configuration** of the secondary CNI feature of the Host Based Networking and OVN Kubernetes use case. Enabling and configuring this feature allows for the creation of pods with secondary networks which are also accelerated by OVN Kubernetes as the secondary CNI.

Before proceeding with this advanced configuration, please ensure you have reviewed the Host Based Networking and OVN Kubernetes configuration guide first and completed that first. This advanced configuration builds upon that setup and provides additional steps to enable secondary cni support.

1. Upgrade OVN Kubernetes from the Helm Chart

Upgrade the OVN Kubernetes CNI components from the helm chart. Ensure environment variables are set before running this command.

```
envsubst < manifests/01-cni-installation/ovn-kubernetes_secondary_network.yml | helm upgrade --install -n ovn-kubernetes ovn-kubernetes ${OVN_KUBERNETES_REPO_URL}/ovn-kubernetes-chart --version $TAG --values -
```

OVN-Kubernetes Helm values

```
commonManifests:
  enabled: true
nodeWithoutDPUManifests:
  enabled: true
controlPlaneManifests:
  enabled: true
  ovnMultiNetworkEnable: true # enables secondary CNI/network
support
nodeWithDPUManifests:
  enabled: true
  nodeMgmtPortNetdev: $DPU_P0_VF1
  dpuServiceAccountNamespace: dpf-operator-system
gatewayOpts: --gateway-interface=$DPU_P0
## Note this CIDR is followed by a trailing /24 which informs OVN
Kubernetes on how to split the CIDR per node.
podNetwork: $POD_CIDR/24
serviceNetwork: $SERVICE_CIDR
k8sAPIServer:
https://$TARGETCLUSTER_API_SERVER_HOST:$TARGETCLUSTER_API_SERVER_PORT
```

Verification

These verification commands may need to be run multiple times to ensure the condition is met.

Verify the CNI installation with:

```
## Ensure all nodes in the cluster are ready.
kubectl wait --for=condition=ready nodes --all
## Ensure all pods in the ovn-kubernetes namespace are ready.
kubectl wait --for=condition=ready --namespace ovn-kubernetes pods
--all --timeout=300s
```

2. Update the DPUServiceTemplate

Ensure environment variables are set before running this command.

```
cat manifests/02-dpudeployment-modifications/*.yaml | envsubst |
kubectl apply -f -
```

OVN DPUServiceTemplate to deploy OVN workloads to the DPUs

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceTemplate
metadata:
  name: ovn
  namespace: dpf-operator-system
spec:
  deploymentServiceName: "ovn"
  helmChart:
    source:
      repoURL: $OVN_KUBERNETES_REPO_URL
      chart: ovn-kubernetes-chart
      version: $TAG
    values:
      commonManifests:
        enabled: true
      dpuManifests:
        enabled: true
        ovnMultiNetworkEnable: true # enables secondary
CNI/network support
        nodeMgmtPortNetdev: $DPU_P0_VF1
      leaseNamespace: "ovn-kubernetes"
      gatewayOpts: "--gateway-interface=br-ovn"
```

Verification

These verification commands may need to be run multiple times to ensure the condition is met.

Note that the DPUService name will have a random suffix. For example, `ovn-hbn-doca-hbn-l2xsl`. Use the correct name for the verification.

Verify the DPU and Service installation with:

```
## Ensure the DPUServices are created and have been reconciled.
kubectl wait --for=condition=ApplicationsReconciled --namespace
dpf-operator-system dpuservices -l svc.dpu.nvidia.com/owned-by-
dpudeployment=dpf-operator-system_ovn-hbn
## Ensure the DPUServiceIPAMs have been reconciled
kubectl wait --for=condition=DPUIPAMObjectReconciled --namespace
dpf-operator-system dpuserviceipam --all
## Ensure the DPUServiceInterfaces have been reconciled
kubectl wait --for=condition=ServiceInterfaceSetReconciled --
namespace dpf-operator-system dpuserviceinterface --all
## Ensure the DPUServiceChains have been reconciled
kubectl wait --for=condition=ServiceChainSetReconciled --namespace
dpf-operator-system dpuservicechain --all
```

3. Test Traffic

If you want to create pods with secondary networks, first create a secondary network NetworkAttachmentDefinition.

```
kubectl apply -f manifests/03-test-traffic/nad_bf3_p0_vfs.yaml
```

Now you can create pods with secondary network interfaces using the following command:

```
kubectl apply -f manifests/03-test-traffic/pods-secondary-network.yaml
```

Once the pods are running, you can check the network interfaces inside the pods to verify that the secondary network interfaces have been created. They should be created with the interface name `net1` and should have an IP address from the `192.168.100.0/24` as defined in the NetworkAttachmentDefinition.

You can ping or run iperf traffic between the pods using the secondary network interfaces to test connectivity and performance.

Using Private Registries

The system components and images for DPF are published to publicly available repos that have no authentication. Users who consume these artifacts from registries with authentication and will need to create Kubernetes Secrets to manage access.

DPF uses needs to authenticate with registries in the following ways:

- **DPF Operator Installation**: For pulling the operator image
- **Helm login**: To install the DPFOperatorConfig helm chart
- **DPFOperatorConfig**: contains a field `.spec.imagePullSecrets` which injects secrets into system components
- **DPUServices**: To pull DPUService images
- **ArgoCD repository Secret**: To enable ArgoCD to pull DPUService helm charts

Image pull secrets

Kubernetes Pods which run images from an authenticated registry will need a secret to access the registry.

To create an image pull secret, you need to specify the following environment variables:

```
## The registry the image pull secret will be created for.
export REGISTRY=${REGISTRY:?Must specify the registry}

## The namespace to which the image pull secret will be created.
## Note: If you're creating DPUServices in other namespaces, you'll need to create the secret for each namespace.
export IMAGE_PULL_SECRET_NAMESPACE="${IMAGE_PULL_SECRET_NAMESPACE:-dpf-operator-system}"

## The username used to log in to the registry.
export IMAGE_REGISTRY_USERNAME=${IMAGE_REGISTRY_USERNAME:?Must specify the registry username}

## The image pull key for the registry.
export IMAGE_PULL_KEY=${IMAGE_PULL_KEY:?Must specify the image pull key}
```

Log in to the registry to ensure the variables are correct:

```
echo "$IMAGE_PULL_KEY" | docker login --username "$IMAGE_REGISTRY_USERNAME" --password-stdin $REGISTRY
```

Create the image pull secret:

```
echo "Creating image pull secret in namespace: $ns"
kubectl -n "$ns" create secret docker-registry dpf-pull-secret --docker-server="$REGISTRY" --docker-username="$IMAGE_REGISTRY_USERNAME" --docker-password="$IMAGE_PULL_KEY" --dry-run=client -o yaml | kubectl apply -f -
```

Using the DPF pull secret for DPUServices

DPUServices run on a DPUCluster and image pull secrets must be explicitly mirrored to them. This mirroring is done by labelling the secret:

```
kubectl -n $IMAGE_PULL_SECRET_NAMESPACE label secret dpf-pull-secret dpu.nvidia.com/image-pull-secret=""
```

Any Secret with this label will be mirrored to the DPUCluster and can be used there.

ArgoCD repository secrets

DPUServices which reference helm charts from public registries will need a secret to access the helm chart repository.

To create an ArgoCD repository secret, you need to specify the following environment variables:

```
## The registry the image pull secret will be created for.
export HELM_REPOSITORY_URL=${HELM_REPOSITORY_URL:?Must specify the helm repository url}

## The name of the repository secret and the registry
export HELM_REPOSITORY_NAME=${HELM_REPOSITORY_NAME:-dpf-helm-repository}

## The username used to log in to the registry.
export HELM_REPOSITORY_USERNAME=${HELM_REPOSITORY_USERNAME:?Must specify the helm repository username}

## The key/password used to authenticate with the helm repository
export HELM_REPOSITORY_KEY=${HELM_REPOSITORY_KEY:?Must specify the helm repository key}
```

```
envsubst < using-private-registries-argocd-repository-secret.yaml
| kubectl apply -f -
```

```
apiVersion: v1
kind: Secret
metadata:
  name: $HELM_REPOSITORY_NAME
  namespace: dpf-operator-system
  labels:
    argocd.argoproj.io/secret-type: repository
stringData:
  name: $HELM_REPOSITORY_NAME
  url: $HELM_REPOSITORY_URL
  type: helm
  username: $HELM_REPOSITORY_USERNAME
  password: $HELM_REPOSITORY_KEY
```

Zero Trust Advanced Configuration

This section includes advanced configuration and additional information for the Zero Trust use case.

DPU Discovery and DPUNode and DPUDevice Object Creation

DPF provides two approaches for discovering and creating DPU resources:

1. **Automated Discovery**: Using `DPUDiscovery` to automatically scan for DPUs and create `DPUDevice` and `DPUNode` resources.

2. **Manual Creation**: Manually creating `DPUDevice` and `DPUNode` resources for each DPU.

You can choose either approach based on your deployment requirements. Automated discovery is recommended for larger deployments, while manual creation provides more control for smaller or specific configurations.

Automated DPU Discovery

DPUDiscovery enables automatic discovery of DPU devices and nodes by scanning specified IP ranges. This approach automatically creates `DPUDevice` and `DPUNode` resources for any discovered DPUs.

1. First, create a YAML file for the DPUDiscovery resource. Let's call it `dpudiscovery.yaml`:

```
apiVersion: provisioning.dpu.nvidia.com/v1alpha1
kind: DPUDiscovery
metadata:
  name: dpu-discovery-192.168.1-10
  namespace: dpf-operator-system
spec:
  # Define the IP range to scan
  ipRangeSpec:
    ipRange:
      startIP: "10.0.110.120"    # Replace with your start IP
      endIP: "10.0.110.125"     # Replace with your end IP

  # Optional: Set scan interval
  scanInterval: "3m"
  # Optional: Set number of workers (default is 1 per 255 IPs)
  workers: 1
```

2. Apply the resource using kubectl:

```
kubectl apply -f dpudiscovery.yaml
```

3. Check the status of the crawler:

```
kubectl get dpudiscovery dpu-discovery-192.168.1-10 -o yaml
```

The DPU discovery will:

1. Start scanning the specified IP range
2. Create DPUDevice and DPUNode* resources for any discovered DPUs
3. Continue scanning at the specified interval
4. Update its status with the last scan time and found DPUs

You can monitor the discovered DPUs with:

```
# List discovered DPU devices
kubectl get dpudevices

# List discovered DPU nodes
kubectl get dpunodes
```

* DPUDiscovery will skip the creation of a DPUNode if there is an existing one with the spec.dpus field containing the DPUDevices serial number.

Limitations

- When using autodiscovery for DPUNodes, the created DPUNodes will be named after `dpunode-<DPU_SERIAL_NUMBER>`. In case the HBN DPUService is used in conjuction with this DPU provisioning mode, the HBN configuration needs to be adjusted to match the discovered nodes accordingly.

Manual DPU Resource Creation

If you prefer to manually create DPU resources or need more control over the creation process, you can create `DPUDevice` and `DPUNode` resources manually.

Creating DPUDevice manually

Create a `DPUDevice` resource for each DPU:

> **Note**: The `DPUDevice` is immutable, and creating a DPUDevice will not trigger DPU provisioning.

```
---
apiVersion: provisioning.dpu.nvidia.com/v1alpha1
kind: DPUDevice
metadata:
  name: dpu-device-1
  namespace: dpf-operator-system
spec:
    bmcIp: 10.0.110.122
```

Creating a DPUNode manually

Create a `DPUNode` resource for each host that has a DPU:

> **Note**
>
> The `.spec.dpus` field contains the names of each DPUDevice attached to the node. Currently, DPF only supports setting a single DPU for each DPUNode.

```yaml
---
apiVersion: provisioning.dpu.nvidia.com/v1alpha1
kind: DPUNode
metadata:
  labels:
    feature.node.kubernetes.io/dpu-enabled: "true"
  name: worker1
  namespace: dpf-operator-system
spec:
  dpus:
  - name: dpu-device-1
  nodeRebootMethod:
    external: {}
```

External Host Reboot

In the Zero Trust scenario, DPF cannot manage the DPU's host machine. During the DPU provisioning process, when the DPU CR reaches the `rebooting` phase, manual power-cycling is required by the user. The power-cycle operation must be completed within two hours; otherwise, the DPU join cluster's secret will expire, causing DPU CR pending in `DPU Cluster Config` phase. After the worker node boots up, the `provisioning.dpu.nvidia.com/dpunode-external-reboot-required` annotation on the DPUNode must be manually removed.

Developer Guides

This section provides comprehensive guides for developers working with the DOCA Platform Framework (DPF). It includes detailed information on system architecture, API usage, DPU service development, and more.

Developer Guide Sections

- Architecture - System design, architectural patterns, and deployment modes
- System - Building conformant DPF systems, development environments, and testing strategies
- Services - Creating and extending DPU services, including storage and networking services
- API - Complete API reference and usage for all DPF Custom Resource Definitions

API

This section contains comprehensive API documentation for the DOCA Platform Framework (DPF).

Each API document provides detailed specifications, examples, and usage guidelines. For implementation details, refer to the API source code in the repository.

API Documentation Files

- API - Complete API reference with all CRD specifications
- DPUFlavor - DPUFlavor CRD defining configuration templates for DPU system-level settings
- DPUService - DPUService CRD for deploying applications on DPU nodes
- DPUDeployment - DPUDeployment CRD for provisioning multiple DPUs with multiple DPUServices running on top of them

- DPUServiceChain - DPUServiceChain CRD for traffic steering and service chaining
- DPUSet - DPUSet CRD for managing DPU provisioning
- DPUCluster - DPUCluster CRD for DPU cluster control plane management
- DPFOperatorConfig - DPFOperatorConfig CRD for managing the operator configuration
- DPUServiceNAD - DPUServiceNAD CRD for enabling secondary networks on the DPUCluster
- DPUServiceIPAM - DPUServiceIPAM CRD for IP address management of the DPUServices
- DPUServiceCredentialRequest - DPUServiceCredentialRequest CRD for enabling cross Kubernetes cluster access

DPU Management Resources

- DPUDevice - DPUDevice CRD for managing individual DPU hardware devices
- DPUNode - DPUNode CRD for node-level DPU management and host operations
- DPUDiscovery - DPUDiscovery CRD for automatic discovery of DPU devices in IP ranges

API reference

API Reference

Packages

- operator.dpu.nvidia.com/v1alpha1
- provisioning.dpu.nvidia.com/v1alpha1
- storage.dpu.nvidia.com/v1alpha1
- svc.dpu.nvidia.com/v1alpha1
- vpc.dpu.nvidia.com/v1alpha1

operator.dpu.nvidia.com/v1alpha1

Package v1alpha1 contains API Schema definitions for the operator v1alpha1 API group

Resource Types

- DPFOperatorConfig
- DPFOperatorConfigList

BFBRegistryConfiguration

Appears in: - InstallViaRedfish

Field	Description	Default	Validation
`disable` *boolean*	Disable ensures the BFB Registry is not deployed when set to true.		
`port` *integer*	Port is the port on which the BFB Registry will listen		

BaseComponentConfig

BaseComponentConfig provides common configuration fields that can be embedded by all component configurations to reduce code duplication.

Appears in: - CNIInstallerConfiguration - DPUDetectorConfiguration - DPUServiceControllerConfiguration - FlannelConfiguration - KamajiClusterManagerConfiguration - MultusConfiguration - NVIPAMConfiguration - OVSCNIConfiguration - ProvisioningControllerConfiguration - SFCControllerConfiguration - SRIOVDevicePluginConfiguration - ServiceSetControllerConfiguration - StaticClusterManagerConfiguration

Field	Description	Default	Validation
`disable` *boolean*	Disable ensures the component is not deployed when set to true.		

CNIInstallerConfiguration

Appears in: - DPFOperatorConfigSpec

Field	Description	Default	Validation
`disable` *boolean*	Disable ensures the component is not deployed when set to true.		
`helmChart` *HelmChart*	HelmChart overrides the helm chart used by the ServiceSet controller. The URL must begin with either 'oci://' or 'https://', ensuring it points to a validOCI registry or a web-based repository.		Pattern: `^(oci://|https://).+$`
`deployInTargetCluster` *boolean*	Deprecated: this field is not supported and will be removed with v26.4.0.		

Field	Description	Default	Validation
`installer` *DefaultOverridesConfiguration*	Installer contains the configuration for the CNI-Installer component. It contains the image for the controller and its resource requirements.		

DPFOperatorConfig

DPFOperatorConfig is the Schema for the dpfoperatorconfigs API

Appears in: - DPFOperatorConfigList

Field	Description	Default	Validati
`apiVersion` *string*	`operator.dpu.nvidia.com/v1alpha1`		
`kind` *string*	`DPFOperatorConfig`		
`metadata` *ObjectMeta*	Refer to Kubernetes API documentation for fields of `metadata`.		
`spec` *DPFOperatorConfigSpec*			
`status` *DPFOperatorConfigStatus*			

DPFOperatorConfigList

DPFOperatorConfigList contains a list of DPFOperatorConfig

Field	Description	Default	Validation
`apiVersion` *string*	`operator.dpu.nvidia.com/v1alpha1`		
`kind` *string*	`DPFOperatorConfigList`		
`metadata` *ListMeta*	Refer to Kubernetes API documentation for fields of `metadata`.		
`items` *DPFOperatorConfig array*			

DPFOperatorConfigSpec

DPFOperatorConfigSpec defines the desired state of DPFOperatorConfig

Appears in: - DPFOperatorConfig

Field	Description	Default	Va
`overrides` *Overrides*			
`networking` *Networking*		{ controlPlaneMTU:1500 }	
`imagePullSecrets` *string array*	List of secret names which are used to pull images for DPF system components and DPUServices. These secrets must be in the same namespace as the DPF Operator Config and should be created before the config is created.System reconciliation will not proceed until these secrets are available.		

Field	Description	Default	Va
`dpuServiceController` *DPUServiceControllerConfiguration*	DPUServiceController is the configuration for the DPUServiceController		
`provisioningController` *ProvisioningControllerConfiguration*	ProvisioningController is the configuration for the ProvisioningController		
`serviceSetController` *ServiceSetControllerConfiguration*	ServiceSetController is the configuration for the ServiceSetController		
`dpuDetector` *DPUDetectorConfiguration*	DPUDetector is the configuration for the DPUDetector.		
`multus` *MultusConfiguration*	Multus is the configuration for Multus		
`sriovDevicePlugin` *SRIOVDevicePluginConfiguration*	SRIOVDevicePlugin is the configuration for the SRIOVDevicePlugin		
`flannel` *FlannelConfiguration*	Flannel is the configuration for Flannel		
`ovsCNI` *OVSCNIConfiguration*	OVSCNI is the configuration for OVSCNI		
`nvipam` *NVIPAMConfiguration*	NVIPAM is the configuration for NVIPAM		
`cniInstaller` *CNIInstallerConfiguration*	CNIInstaller is the configuration for the cni-installer		
`sfcController` *SFCControllerConfiguration*	SFCController is the configuration for the SFCController		

Field	Description	Default	Va
`kamajiClusterManager` *KamajiClusterManagerConfiguration*	KamajiClusterManager is the configuration for the kamaji-cluster-manager		
`staticClusterManager` *StaticClusterManagerConfiguration*	StaticClusterManager is the configuration for the static-cluster-manager		

DPFOperatorConfigStatus

DPFOperatorConfigStatus defines the observed state of DPFOperatorConfig

Appears in: - DPFOperatorConfig

Field	Description	Default	Validation
`conditions` *Condition array*	Conditions exposes the current state of the OperatorConfig.		
`observedGeneration` *integer*	ObservedGeneration records the Generation observed on the object the last time it was patched.		
`version` *string*	Version is the version of the DPF Operator that is currently deployed.		

DPUDetectorConfiguration

Appears in: - DPFOperatorConfigSpec

Field	Description	Default	Validation
`disable` *boolean*	Disable ensures the component is not deployed when set to true.		

Field	Description	Default	Validation
`image` *Image*	Image overrides the container image used by the DPUDetector Container. Deprecated: This field is deprecated and will be removed with v26.4.0.Use the new field `daemon` instead.		Pattern: `^((?:(?:(?:[a-zA-Z0-9]|[a-zA-Z0-9][a-zA-Z0-9-]*[a-zA-Z0-9])(?:\.(?:[a-zA-Z0-9]|[a-zA-Z0-9][a-zA-Z0-9-]*[a-zA-Z0-9]))*|\[(?:[a-fA-F0-9:]+)\])(?::[0-9]+)?/)?[a-z0-9]+(?:(?:[._]|__|[-]+)[a-z0-9]+)*(?:/[a-z0-9]+(?:(?:[._]|__|[-]+)[a-z0-9]+)*)*)(?::([\w][\w.-]\{0,127\}))?(?:@([A-Za-z][A-Za-z0-9]*(?:[-_+.][A-Za-z][A-Za-z0-9]*)*[:][[:xdigit:]]\{32,\}))?$`
`daemon` *DefaultOverridesConfiguration*	Daemon contains the configuration for the DPU Detector component. It contains the image for the controller and its resource requirements.		

DPUServiceControllerConfiguration

Appears in: - DPFOperatorConfigSpec

Field	Description	Default	Validation
`disable` *boolean*	Disable ensures the component is not deployed when set to true.		
`image` *Image*	Image overrides the container image used by the DPUService controller. Deprecated: This field is deprecated and will be removed with v26.4.0.Use the new field `controller` instead.		Pattern: `^((?:(?:(?:[a-zA-Z0-9]|[a-zA-Z0-9][a-zA-Z0-9-]*[a-zA-Z0-9])(?:\.(?:[a-zA-Z0-9]|[a-zA-Z0-9][a-zA-Z0-9-]*[a-zA-Z0-9]))*|\[(?:[a-fA-F0-9:]+)\])(?::[0-9]+)?/)?[a-z0-9]+(?:(?:[._]|__|[-]+)[a-z0-9]+)*(?:/[a-z0-9]+(?:(?:[._]|__|[-]+)[a-z0-9]+)*)*)(?::([\w][\w.-]\{0,127\}))?(?:@([A-Za-z][A-Za-z0-9]*(?:[-_+.][A-Za-z][A-Za-z0-9]*)*[:][[:xdigit:]]\{32,\}))?$`

Field	Description	Default	Validation
`controller` *DefaultOverridesConfiguration*	Controller contains the configuration for the DPU Service controller component. It contains the image for the controller and its resource requirements.		
`disableDPUReadyTaints` *boolean*	DisableDPUReadyTaints disables the DPU ready taints feature in the DPU Service Controller. This feature adds taints to the worker nodes when the DPU is not ready.This is useful when the DPU is used for networking and the node should not be scheduled until the DPU is ready.		

DefaultOverridesConfiguration

Appears in: - CNIInstallerConfiguration - DPUDetectorConfiguration - DPUServiceControllerConfiguration - KamajiClusterManagerConfiguration - MultusConfiguration - OVSCNIConfiguration - ProvisioningControllerConfiguration - SFCControllerConfiguration - SRIOVDevicePluginConfiguration - ServiceSetControllerConfiguration - StaticClusterManagerConfiguration

Field	Description	Default	Validation
`image` *Image*			Pattern: `^((?:(?:(?:[a-zA-Z0-9]|[a-zA-Z0-9][a-zA-Z0-9-]*[a-zA-Z0-9])(?:\.(?:[a-zA-Z0-9]|[a-zA-Z0-9][a-zA-Z0-9-]*[a-zA-Z0-9]))*|\[(?:[a-fA-F0-9:]+)\])(?::[0-9]+)?/)?[a-z0-9]+(?:(?:[._]|__|[-]+)[a-z0-9]+)*(?:/[a-z0-9]+(?:(?:[._]|__|[-]+)[a-z0-9]+)*)*)(?::([\w][\w.-]\{0,127\}))?(?:@([A-Za-z][A-Za-z0-9]*(?:[-_+.][A-Za-z][A-Za-z0-9]*)*[:][[:xdigit:]]\{32,\}))?$`
`resources` *ResourceRequirements*	Resources defines the memory and CPU resource requests and limits for the component. This field is optional, and if not set, the component will use the default resource.		

FlannelCNI

Appears in: - FlannelConfiguration

Field	Description	Default	Validation
`image` *Image*			Pattern: `^((?:(?:(?:[a-zA-Z0-9]|[a-zA-Z0-9][a-zA-Z0-9-]*[a-zA-Z0-9])(?:\.(?:[a-zA-Z0-9]|[a-zA-Z0-9][a-zA-Z0-9-]*[a-zA-Z0-9]))*|\[(?:[a-fA-F0-9:]+)\])(?::[0-9]+)?/)?[a-z0-9]+(?:(?:[._]|__|[-]+)[a-z0-9]+)*(?:/[a-z0-9]+(?:(?:[._]|__|[-]+)[a-z0-9]+)*)*)(?::([\w][\w.-]\{0,127\}))?(?:@([A-Za-z][A-Za-z0-9]*(?:[-_+.][A-Za-z][A-Za-z0-9]*)*[:][[:xdigit:]]\{32,\}))?$`

FlannelConfiguration

Appears in: - DPFOperatorConfigSpec

Field	Description	Default	Validation
`disable` *boolean*	Disable ensures the component is not deployed when set to true.		
`helmChart` *HelmChart*	HelmChart overrides the helm chart used by the ServiceSet controller. The URL must begin with either 'oci://' or 'https://', ensuring it points to a validOCI registry or a web-based repository.		Pattern: `^(oci://|https://).+$`

Field	Description	Default	Validation
`deployInTargetCluster` *boolean*	Deprecated: this field is not supported and will be removed with v26.4.0.		
`cni` *FlannelCNI*	CNI is the configuration for the Flannel CNI component. It contains the image for the CNI init container.Note: The resources for the CNI container are not configurable.		
`daemon` *FlannelDaemon*	Daemon is the configuration for the Flannel Daemon component. It contains the image for the Flannel Daemon container and its resource requirements.		
`image` *FlannelImages*	Images overrides the container images used by flannel Deprecated: This field is deprecated and will be removed with v26.4.0.Use the new fields `cni` and `daemon` instead.		

Field	Description	Default	Validation
`podCIDR` *string*	PodCIDR is the pod cidr for flannel.		

FlannelDaemon

Appears in: - FlannelConfiguration

Field	Description	Default	Validation
`image` *Image*			Pattern: `^((?:(?:(?:[a-zA-Z0-9]|[a-zA-Z0-9][a-zA-Z0-9-]*[a-zA-Z0-9])(?:\.(?:[a-zA-Z0-9]|[a-zA-Z0-9][a-zA-Z0-9-]*[a-zA-Z0-9]))*|\[(?:[a-fA-F0-9:]+)\])(?::[0-9]+)?/)?[a-z0-9]+(?:(?:[._]|__|[-]+)[a-z0-9]+)*(?:/[a-z0-9]+(?:(?:[._]|__|[-]+)[a-z0-9]+)*)*)(?::([\w][\w.-]\{0,127\}))?(?:@([A-Za-z][A-Za-z0-9]*(?:[-_+.][A-Za-z][A-Za-z0-9]*)*[:][[:xdigit:]]\{32,\}))?$`
`resources` *ResourceRequirements*	Resources defines the memory and CPU resource requests and limits for the component. This field is optional, and if not set, the component will use the default resource.		

FlannelImages

Appears in: - FlannelConfiguration

Field	Description	Default	Validation
`flannelCNI` *string*	FlannelCNI must be set if FlannelImages is set.		MinLength: 1
`kubeFlannel` *string*	KubeFlannel must be set if FlannelImages is set.		MinLength: 1

HelmChart

Underlying type:string

HelmChart is a reference to a helm chart.

Validation: - Pattern: `^(oci://|https://).+$`

Appears in: - CNIInstallerConfiguration - FlannelConfiguration - HelmComponentConfig - MultusConfiguration - NVIPAMConfiguration - OVSCNIConfiguration - SFCControllerConfiguration - SRIOVDevicePluginConfiguration - ServiceSetControllerConfiguration

HelmComponentConfig

Appears in: - CNIInstallerConfiguration - FlannelConfiguration - MultusConfiguration - NVIPAMConfiguration - OVSCNIConfiguration - SFCControllerConfiguration - SRIOVDevicePluginConfiguration - ServiceSetControllerConfiguration

Field	Description	Default	Validation
`helmChart` *HelmChart*	HelmChart overrides the helm chart used by the ServiceSet controller. The URL must begin with either 'oci://' or 'https://', ensuring it points to a validOCI registry or a web-based repository.		Pattern: `^(oci://|https://).+$`

Image

Underlying type:string

Image is a reference to a container image.

Validation: - Pattern:

```
^((?:(?:(?:[a-zA-Z0-9]|[a-zA-Z0-9][a-zA-Z0-9-]*[a-zA-Z0-9])(?:\.(?:
[a-zA-Z0-9]|[a-zA-Z0-9][a-zA-Z0-9-]*[a-zA-Z0-9]))*|\[(?:[a-fA-F0-
9:]+)\])(?::[0-9]+)?/)?[a-z0-9]+(?:(?:[._]|__|[-]+)[a-z0-9]+)*
(?:/[a-z0-9]+(?:(?:[._]|__|[-]+)[a-z0-9]+)*)*)(?::([\w][\w.-]
{0,127}))?(?:@([A-Za-z][A-Za-z0-9]*(?:[-_+.][A-Za-z][A-Za-z0-9]*)*
[:][[:xdigit:]]{32,}))?$
```

Appears in: - DPUDetectorConfiguration - DPUServiceControllerConfiguration - DefaultOverridesConfiguration - FlannelCNI - FlannelDaemon - ImageComponentConfig - KamajiClusterManagerConfiguration - MultusConfiguration - NVIPAMConfiguration - NVIPAMController - OVSCNIConfiguration - ProvisioningControllerConfiguration - SFCControllerConfiguration - SRIOVDevicePluginConfiguration - ServiceSetControllerConfiguration - StaticClusterManagerConfiguration

ImageComponentConfig

ImageComponentConfig provides common configuration fields that can be embedded by all component configurations to reduce code duplication.

Appears in: - DefaultOverridesConfiguration - FlannelCNI - FlannelDaemon - NVIPAMController

Field	Description	Default	Validation
`image` *Image*			Pattern: `^((?:(?:(?:[a-zA-Z0-9]|[a-zA-Z0-9][a-zA-Z0-9-]*[a-zA-Z0-9])(?:\.(?:[a-zA-Z0-9]|[a-zA-Z0-9][a-zA-Z0-9-]*[a-zA-Z0-9]))*|\[(?:[a-fA-F0-9:]+)\])(?::[0-9]+)?/)?[a-z0-9]+(?:(?:[._]|__|[-]+)[a-z0-9]+)*(?:/[a-z0-9]+(?:(?:[._]|__|[-]+)[a-z0-9]+)*)*)(?::([\w][\w.-]\{0,127\}))?(?:@([A-Za-z][A-Za-z0-9]*(?:[-_+.][A-Za-z][A-Za-z0-9]*)*[:][[:xdigit:]]\{32,\}))?$`

InClusterDeploymentConfig

Appears in: - CNIInstallerConfiguration - FlannelConfiguration - MultusConfiguration - NVIPAMConfiguration - OVSCNIConfiguration - SFCControllerConfiguration - SRIOVDevicePluginConfiguration - ServiceSetControllerConfiguration

Field	Description	Default	Validation
`deployInTargetCluster` *boolean*	Deprecated: this field is not supported and will be removed with v26.4.0.		

InstallViaGNOI

InstallViaGNOI is the interface used to install the BFB via GNOI

Appears in: - ProvisioningInstallInterface

InstallViaHostAgent

InstallViaHostAgent is the interface used to install the BFB

Appears in: - ProvisioningInstallInterface

InstallViaRedfish

InstallViaRedfish is the interface used to install the BFB via Redfish

Appears in: - ProvisioningInstallInterface

Field	Description	Default	Validation
`bfbRegistryAddress` *string*	BFBRegistryAddress is the address of the BFB Registry Deprecated: Use RegistryConfiguration instead.		MinLength: 1
`bfbRegistry` *BFBRegistryConfiguration*	BFBRegistry is the configuration for the BFB Registry Deprecated: Use RegistryConfiguration instead.		
`skipDPUNodeDiscovery` *boolean*	SkipDPUNodeDiscovery is a flag to skip the DPU node discovery.	true	

KamajiClusterManagerConfiguration

Appears in: - DPFOperatorConfigSpec

Field	Description	Default	Validation
`disable` *boolean*	Disable ensures the component is not deployed when set to true.		

Field	Description	Default	Validation
`image` *Image*	Image overrides the container image used by the Kamaji Cluster Manager. Deprecated: This field is deprecated and will be removed with v26.4.0.Use the new field `controller` instead.		Pattern: `^((?:(?:(?:[a-zA-Z0-9]|[a-zA-Z0-9][a-zA-Z0-9-]*[a-zA-Z0-9])(?:\.(?:[a-zA-Z0-9]|[a-zA-Z0-9][a-zA-Z0-9-]*[a-zA-Z0-9]))*|\[(?:[a-fA-F0-9:]+)\])(?::[0-9]+)?/)?[a-z0-9]+(?:(?:[._]|__|[-]+)[a-z0-9]+)*(?:/[a-z0-9]+(?:(?:[._]|__|[-]+)[a-z0-9]+)*)*)(?::([\w][\w.-]\{0,127\}))?(?:@([A-Za-z][A-Za-z0-9]*(?:[-_+.][A-Za-z][A-Za-z0-9]*)*[:][[:xdigit:]]\{32,\}))?$`
`controller` *DefaultOverridesConfiguration*	Controller contains the configuration for the Kamaji Cluster Manager component. It contains the image for the controller and its resource requirements.		

MultusConfiguration

Appears in: - DPFOperatorConfigSpec

Field	Description	Default	Validation
`disable` *boolean*	Disable ensures the component is not deployed when set to true.		
`helmChart` *HelmChart*	HelmChart overrides the helm chart used by the ServiceSet controller. The URL must begin with either 'oci://' or 'https://', ensuring it points to a validOCI registry or a web-based repository.		Pattern: `^(oci://|https://).+$`
`deployInTargetCluster` *boolean*	Deprecated: this field is not supported and will be removed with v26.4.0.		

Field	Description	Default	Validation
`image` *Image*	Image overrides the container image used by the Multus Container. Deprecated: This field is deprecated and will be removed with v26.4.0.Use the new field `cni` instead.		Pattern: `^((?:(?:(?:[a-zA-Z0-9]|[a-zA-Z0-9][a-zA-Z0-9-]*[a-zA-Z0-9])(?:\.(?:[a-zA-Z0-9]|[a-zA-Z0-9][a-zA-Z0-9-]*[a-zA-Z0-9]))*|\[(?:[a-fA-F0-9:]+)\])(?::[0-9]+)?/)?[a-z0-9]+(?:(?:[._]|__|[-]+)[a-z0-9]+)*(?:/[a-z0-9]+(?:(?:[._]|__|[-]+)[a-z0-9]+)*)*)(?::([\w][\w.-]\{0,127\}))?(?:@([A-Za-z][A-Za-z0-9]*(?:[-_+.][A-Za-z][A-Za-z0-9]*)*[:][[:xdigit:]]\{32,\}))?$`
`cni` *DefaultOverridesConfiguration*	CNI contains the configuration for the Multus CNI component. It contains the image for the controller and its resource requirements.		

NVIPAMConfiguration

Appears in: - DPFOperatorConfigSpec

Field	Description	Default	Validation
`disable` *boolean*	Disable ensures the component is not deployed when set to true.		
`helmChart` *HelmChart*	HelmChart overrides the helm chart used by the ServiceSet controller. The URL must begin with either 'oci://' or 'https://', ensuring it points to a validOCI registry or a web-based repository.		Pattern: `^(oci://|https://).+$`
`deployInTargetCluster` *boolean*	Deprecated: this field is not supported and will be removed with v26.4.0.		

Field	Description	Default	Validation
`image` *Image*	Image overrides the container image used by the NVIPAM controller. Deprecated: This field is deprecated and will be removed with v26.4.0.Use the new field `controller` instead.		Pattern: `^((?:(?:(?:[a-zA-Z0-9]\\|[a-zA-Z0-9][a-zA-Z0-9-]*[a-zA-Z0-9])(?:\.(?:[a-zA-Z0-9]\\|[a-zA-Z0-9][a-zA-Z0-9-]*[a-zA-Z0-9]))*\\|\[(?:[a-fA-F0-9:]+)\])(?::[0-9]+)?/)?[a-z0-9]+(?:(?:[._]\\|__\\|[-]+)[a-z0-9]+)*(?:/[a-z0-9]+(?:(?:[._]\\|__\\|[-]+)[a-z0-9]+)*)*)(?::([\w][\w.-]\{0,127\}))?(?:@([A-Za-z][A-Za-z0-9]*(?:[-_+.][A-Za-z][A-Za-z0-9]*)*[:][[:xdigit:]]\{32,\}))?$`
`controller` *NVIPAMController*	Controller contains the configuration for the NVIPAM controller component. It contains the image for the controller and its resource requirements.		
`node` *NVIPAMNode*	Node contains the configuration for the NVIPAM node component. It contains the image for the node and its resource requirements.		

NVIPAMController

Appears in: - NVIPAMConfiguration

Field	Description	Default	Validation
`image` *Image*			Pattern: `^((?:(?:(?:[a-zA-Z0-9]|[a-zA-Z0-9][a-zA-Z0-9-]*[a-zA-Z0-9])(?:\.(?:[a-zA-Z0-9]|[a-zA-Z0-9][a-zA-Z0-9-]*[a-zA-Z0-9]))*|\[(?:[a-fA-F0-9:]+)\])(?::[0-9]+)?/)?[a-z0-9]+(?:(?:[._]|__|[-]+)[a-z0-9]+)*(?:/[a-z0-9]+(?:(?:[._]|__|[-]+)[a-z0-9]+)*)*)(?::([\w][\w.-]\{0,127\}))?(?:@([A-Za-z][A-Za-z0-9]*(?:[-_+.][A-Za-z][A-Za-z0-9]*)*[:][[:xdigit:]]\{32,\}))?$`
`resources` *ResourceRequirements*	Resources defines the memory and CPU resource requests and limits for the component. This field is optional, and if not set, the component will use the default resource.		

NVIPAMNode

Appears in: - NVIPAMConfiguration

Field	Description	Default	Validation
`resources` *ResourceRequirements*	Resources defines the memory and CPU resource requests and limits for the component. This field is optional, and if not set, the component will use the default resource.		

Networking

Networking defines the networking configuration for the system components.

Appears in: - DPFOperatorConfigSpec

Field	Description	Default	Validation
`controlPlaneMTU` *integer*	ControlPlaneMTU is the MTU value to be set on the management network. The default is 1500.	1500	Maximum: 9216 Minimum: 1280
`highSpeedMTU` *integer*	HighSpeedMTU is the MTU value to be set on the high-speed interface. The default is 1500.	1500	Maximum: 9216 Minimum: 1280

OVSCNIConfiguration

Appears in: - DPFOperatorConfigSpec

Field	Description	Default	Validation
`disable` *boolean*	Disable ensures the component is not deployed when set to true.		

Field	Description	Default	Validation
`helmChart` *HelmChart*	HelmChart overrides the helm chart used by the ServiceSet controller. The URL must begin with either 'oci://' or 'https://', ensuring it points to a validOCI registry or a web-based repository.		Pattern: `^(oci://\\|https://).+$`
`deployInTargetCluster` *boolean*	Deprecated: this field is not supported and will be removed with v26.4.0.		
`image` *Image*	Image overrides the container image used by the OVS CNI. Deprecated: This field is deprecated and will be removed with v26.4.0.Use the new field `cni` instead.		Pattern: `^((?:(?:(?:[a-zA-Z0-9]\\|[a-zA-Z0-9][a-zA-Z0-9-]*[a-zA-Z0-9])(?:\.(?:[a-zA-Z0-9]\\|[a-zA-Z0-9][a-zA-Z0-9-]*[a-zA-Z0-9]))*\\|\[(?:[a-fA-F0-9:]+)\])(?::[0-9]+)?/)?[a-z0-9]+(?:(?:[._]\\|__\\|[-]+)[a-z0-9]+)*(?:/[a-z0-9]+(?:(?:[._]\\|__\\|[-]+)[a-z0-9]+)*)*)(?::([\w][\w.-]\{0,127\}))?(?:@([A-Za-z][A-Za-z0-9]*(?:[-_+.][A-Za-z][A-Za-z0-9]*)*[:][[:xdigit:]]\{32,\}))?$`

Field	Description	Default	Validation
`cni` *DefaultOverridesConfiguration*	CNI contains the configuration for the OVS CNI component. It contains the image for the controller and its resource requirements.		

Overrides

Overrides exposes a set of fields which impact the recommended behavior of the DPF Operator. These fields should only be set for advanced use cases. The fields here have no stability guarantees.

Appears in: - DPFOperatorConfigSpec

Field	Description
`paused` *boolean*	Paused disables all reconciliation of the DPFOperatorConfig when set to true.
`dpuCNIBinPath` *string*	DPUCNIBinPath is the path at which the CNI binaries will be installed to on the DPU. This is /opt/cni/bin by default.This setting does not change where kubelet is configured to use the CNI from.
`dpuCNIPath` *string*	DPUCNIConfigPath is the path to which the CNI config files will be installed on the DPU. This is /etc/cni/net.d by default.This setting does not change where kubelet is configured to read the CNI config from.

Field	Description
`dpuOpenvSwitchRunPath` *string*	DPUOpenvSwitchPath is the path at which the openvSwitch run directory can be found on the DPU. This is /var/run/openvswitch by default.This setting does not change where components are installed. Installation location fixed in the BFB.
`dpuOpenvSwitchBinPath` *string*	DPUOpenvSwitchBinPath is the path at which the openvSwitch bin directory can be found on the DPU node. This is /usr/bin/ by default.This setting does not change where components are installed. Installation location fixed in the BFB.
`dpuOpenvSwitchSystemSharedPath` *string*	DPUOpenvSwitchSystemSharedLibPath is the path at which the system lib used by OVS components can be found on the DPU. This is /lib by default.This setting does not change where components are installed. Installation location fixed in the BFB.
`flannelSkipCNIConfigInstallation` *boolean*	FlannelSkipCNIConfigInstallation controls whether Flannel should skip CNI config installation. This is true by default, meaning Flannel does not manage its own CNI configuration.Set to false if you want Flannel to install a CNI configuration.
`dpuOpenvSwitchSystemSharedLib64Path` *string*	DPUOpenvSwitchSystemSharedLib64Path is the path at which the system lib64 used by OVS components can be found on the DPU. If this field is not set, no lib64 volume mount will be configured in the SFC Controller component.This setting does not change where components are installed. Installation location fixed in the BFB.

Field	Description
`kubernetesAPIServerVIP` *string*	KubernetesAPIServerVIP is the VIP the Kubernetes API server is accessible at. This setting enables specific underlying components deployed directly or indirectly by the DPF Operator to reachthe Kubernetes API Server when the ClusterIP Kubernetes Service is not functional.If set, it should be set to an IP to ensure that components work even if DNS is not available in the cluster.
`kubernetesAPIServerPort` *integer*	KubernetesAPIServerPort is the port the Kubernetes API server is accessible at. This setting is usually used together with the kubernetesAPIServerVIP setting. It enables specific underlyingcomponents deployed directly or indirectly by the DPF Operator to reach the Kubernetes API Server when theClusterIP Kubernetes Service is not functional.

ProvisioningControllerConfiguration

Appears in: - DPFOperatorConfigSpec

Field	Description
`disable` *boolean*	Disable ensures the component is not deployed

Field	Description
`image` *Image*	Image overrides the container image used by th controller. Deprecated: This field is deprecated and will be v26.4.0.Use the new field `controller` instead
`controller` *DefaultOverridesConfiguration*	Controller contains the configuration for the Pr component. It contains the image for the controller and its r
`bfCFGTemplateConfigMap` *string*	BFCFGTemplateConfigMap is the name of a cor template for the BF.cfg file used by the DPU co By default the provisioning controller use a hard https://github.com/NVIDIA/doca-platform/blob/ v24.10/internal/provisioning/controllers/dpu/bfc Replacing the bf.cfg is an advanced use case. Th designed for most use cases.
`bfbPVCName` *string*	BFBPersistentVolumeClaimName is the name of PersistentVolumeClaim used by dpf-provisioning
`dmsTimeout` *integer*	DMSTimeout is the max time in seconds within respond, 0 is unlimited

Field	Description
`customCASecretName` *string*	CustomCASecretName indicates the name of th object which containing the custom CA certificate
`installInterface` *ProvisioningInstallInterface*	InstallInterface is the interface through which th
`registry` *RegistryConfiguration*	Registry is the configuration for the BFB Registr
`maxDPUParallelInstallations` *integer*	MaxDPUParallelInstallations specifies the maxin that can be provisioned concurrently. A DPU is removed from the concurrent provisior finishes the "OS Installing" phase andenters the its provisioning lifecycle.
`multiDPUOperationsSyncWaitTime` *Duration*	MultiDPUOperationsSyncWaitTime is the wait ti sync operations on the same node. It would take effect only on DPUNode objects w one DPU.
`maxUnavailableDPUNodes` *integer*	MaxUnavailableDPUNodes is the maximum num are unavailable during the node effect period.

ProvisioningInstallInterface

ProvisioningInstallInterface is the interface used to install the BFB

Appears in: - ProvisioningControllerConfiguration

Field	Description	Default	Validation
`installViaGNOI` *InstallViaGNOI*	InstallViaGNOI is the interface used to install the BFB via GNOI Deprecated: Use InstallViaHostAgent instead.		
`installViaHostAgent` *InstallViaHostAgent*	InstallViaHostAgent is the interface used to install the BFB via HostAgent		

Field	Description	Default	Validation
`installViaRedfish` *InstallViaRedfish*	InstallViaRedfish is the interface used to install the BFB via Redfish		

RegistryConfiguration

Appears in: - ProvisioningControllerConfiguration

Field	Description	Default	Validation
`address` *string*	Address is the address used to access the BFB Registry. The address must start with "http://". By default, the BFB Registry can be accessed via its Service.For non-kubernetes environments, this must be set due to the lack of kubelet on worker nodes.For zero-trust environments, this must be set so that the BFB Registry can be accessed from DPU BMC.		Pattern: `^http://`
`port` *integer*	Port is the port on which the registry instances will listen		Maximum: 65535 Minimum: 1

ResourceComponentConfig

ResourceComponentConfig defines the resource requirements for a container.

Appears in: - DefaultOverridesConfiguration - FlannelDaemon - NVIPAMController - NVIPAMNode

Field	Description	Default	Validation
`resources` *ResourceRequirements*	Resources defines the memory and CPU resource requests and limits for the component. This field is optional, and if not set, the component will use the default resource.		

ResourceRequirements

Appears in: - DefaultOverridesConfiguration - FlannelDaemon - NVIPAMController - NVIPAMNode - ResourceComponentConfig

Field	Description	Default	Validation
`requests` *Resources*	Requests defines the resource requests for the component.		
`limits` *Resources*	Limits defines the resource limits for the component.		

Resources

Appears in: - ResourceRequirements

Field	Description	Default	Validation
`cpu` *Quantity*	CPU is the amount of CPU requested by the component.		
`memory` *Quantity*	Memory is the amount of Memory requested by the component.		

SFCControllerConfiguration

Appears in: - DPFOperatorConfigSpec

Field	Description	Default
`disable` *boolean*	Disable ensures the component is not deployed when set to true.	
`helmChart` *HelmChart*	HelmChart overrides the helm chart used by the ServiceSet controller. The URL must begin with either 'oci://' or 'https://', ensuring it points to a validOCI registry or a web-based repository.	

Field	Description	Default
`deployInTargetCluster` *boolean*	Deprecated: this field is not supported and will be removed with v26.4.0.	
`image` *Image*	Image overrides the container image used by the SFC controller. Deprecated: This field is deprecated and will be removed with v26.4.0.Use the new field `controller` instead.	
`controller` *DefaultOverridesConfiguration*	Controller contains the configuration for the SFC controller component. It contains the image for the controller and its resource requirements.	
`secureFlowDeletionTimeout` *Duration*	SecureFlowDeletionTimeout controls the timeout for which the API server is unreachable after which all the flows are deleted to prevent unintended packet leaks. It has effect when is greater than zero.Value must be in units accepted by Go time.ParseDuration https://golang.org/pkg/time/#ParseDuration.	

SRIOVDevicePluginConfiguration

Appears in: - DPFOperatorConfigSpec

Field	Description	Default	Validation
`disable` *boolean*	Disable ensures the component is not deployed when set to true.		
`helmChart` *HelmChart*	HelmChart overrides the helm chart used by the ServiceSet controller. The URL must begin with either 'oci://' or 'https://', ensuring it points to a validOCI registry or a web-based repository.		Pattern: `^(oci://\\|https://).+$`
`deployInTargetCluster` *boolean*	Deprecated: this field is not supported and will be removed with v26.4.0.		

Field	Description	Default	Validation
`image` *Image*	Image overrides the container image used by the SRIOV Device Plugin container. Deprecated: This field is deprecated and will be removed with v26.4.0.Use the new field `deviceplugin` instead.		Pattern: `^((?:(?:(?:[a-zA-Z0-9]|[a-zA-Z0-9][a-zA-Z0-9-]*[a-zA-Z0-9])(?:\.(?:[a-zA-Z0-9]|[a-zA-Z0-9][a-zA-Z0-9-]*[a-zA-Z0-9]))*|\[(?:[a-fA-F0-9:]+)\])(?::[0-9]+)?/)?[a-z0-9]+(?:(?:[._]|__|[-]+)[a-z0-9]+)*(?:/[a-z0-9]+(?:(?:[._]|__|[-]+)[a-z0-9]+)*)*)(?::([\w][\w.-]\{0,127\}))?(?:@([A-Za-z][A-Za-z0-9]*(?:[-_+.][A-Za-z][A-Za-z0-9]*)*[:][[:xdigit:]]\{32,\}))?$`
`deviceplugin` *DefaultOverridesConfiguration*	DevicePlugin contains the configuration for the SRIOV Device Plugin component. It contains the image for the controller and its resource requirements.		

ServiceSetControllerConfiguration

Appears in: - DPFOperatorConfigSpec

Field	Description	Default	Validation
`disable` *boolean*	Disable ensures the component is not deployed when set to true.		
`helmChart` *HelmChart*	HelmChart overrides the helm chart used by the ServiceSet controller. The URL must begin with either 'oci://' or 'https://', ensuring it points to a validOCI registry or a web-based repository.		Pattern: `^(oci://|https://).+$`
`deployInTargetCluster` *boolean*	Deprecated: this field is not supported and will be removed with v26.4.0.		

Field	Description	Default	Validation
`image` *Image*	Image overrides the container image used by the ServiceChainSet Controller. Deprecated: This field is deprecated and will be removed with v26.4.0.Use the new field `controller` instead.		Pattern: `^((?:(?:(?:[a-zA-Z0-9]|[a-zA-Z0-9][a-zA-Z0-9-]*[a-zA-Z0-9])(?:\.(?:[a-zA-Z0-9]|[a-zA-Z0-9][a-zA-Z0-9-]*[a-zA-Z0-9]))*|\[(?:[a-fA-F0-9:]+)\])(?::[0-9]+)?/)?[a-z0-9]+(?:(?:[._]|__|[-]+)[a-z0-9]+)*(?:/[a-z0-9]+(?:(?:[._]|__|[-]+)[a-z0-9]+)*)*)(?::([\w][\w.-]\{0,127\}))?(?:@([A-Za-z][A-Za-z0-9]*(?:[-_+.][A-Za-z][A-Za-z0-9]*)*[:][[:xdigit:]]\{32,\}))?$`
`controller` *DefaultOverridesConfiguration*	Controller contains the configuration for the ServiceChainSet controller component. It contains the image for the controller and its resource requirements.		

StaticClusterManagerConfiguration

Appears in: - DPFOperatorConfigSpec

Field	Description	Default	Validation
`disable` *boolean*	Disable ensures the component is not deployed when set to true.		
`image` *Image*	Image overrides the container image used by the Static Cluster Manager. Deprecated: This field is deprecated and will be removed with v26.4.0.Use the new field `controller` instead.		Pattern: `^((?:(?:(?:[a-zA-Z0-9]|[a-zA-Z0-9][a-zA-Z0-9-]*[a-zA-Z0-9])(?:\.(?:[a-zA-Z0-9]|[a-zA-Z0-9][a-zA-Z0-9-]*[a-zA-Z0-9]))*|\[(?:[a-fA-F0-9:]+)\])(?::[0-9]+)?/)?[a-z0-9]+(?:(?:[._]|__|[-]+)[a-z0-9]+)*(?:/[a-z0-9]+(?:(?:[._]|__|[-]+)[a-z0-9]+)*)*)(?::([\w][\w.-]\{0,127\}))?(?:@([A-Za-z][A-Za-z0-9]*(?:[-_+.][A-Za-z][A-Za-z0-9]*)*[:][[:xdigit:]]\{32,\}))?$`
`controller` *DefaultOverridesConfiguration*	Controller contains the configuration for the Static Cluster Manager controller component. It contains the image for the controller and its resource requirements.		

provisioning.dpu.nvidia.com/v1alpha1

Package v1alpha1 contains API Schema definitions for the provisioning.dpu v1alpha1 API group

Resource Types

- BFB
- BFBList
- DPU
- DPUCluster
- DPUClusterList
- DPUDevice
- DPUDeviceList
- DPUDiscovery
- DPUDiscoveryList
- DPUFlavor
- DPUFlavorList
- DPUList
- DPUNode
- DPUNodeList
- DPUNodeMaintenance
- DPUNodeMaintenanceList
- DPUSet

- DPUSetList

Action

Appears in: - DPUs - NodeEffect

Field	Description	Default	Valida
`taint` *Taint*	Add specify taint on the DPU node		
`noEffect` *boolean*	Do not do any action on the DPU node		
`customLabel` *object (keys:string, values:string)*	Add specify labels on the DPU node		
`drain` *boolean*	Drain the K8s host node by NodeMaintenance operator		
`customAction` *string*	Name of a config map which contains a pod yaml definition to run which will apply the nodeEffect. The pod is expected to exit when node effect is done, if pod terminates with error then DPU would move to an error phase.The DPUNode's name will be exported as an environment variable, named as DPUNODE_NAME, to each container and init container in the pod.The labels and annotations of DPUNode will be exported in `/etc/dpu/dpf-pod-info/labels` and `/etc/dpu/dpf-pod-info/annotations` accordingly; the volume name `dpf-pod-info` is used to mount the labels and annotations.If any name confliction for env or volume, the controller will not export the name or labels/annotations of DPUNode accordingly.		
`hold` *boolean*	Places annotation `wait-for-external-nodeeffect` and waits for it to be removed this is the default behavior in a non K8S environment		

Field	Description	Default	Valida
`force` *boolean*	Force is the flag to indicate if the node effect should be applied immediately. If true, dpfOperatorConfig.multiDPUOperationsSyncWaitTime and dpfOperatorConfig.maxUnavailableDPUNodes will be ignored when applying node effect for DPUNodeMaintenance CR	false	

BFB

BFB is the Schema for the bfbs API

Appears in: - BFBList

Field	Description	Default	Valida
`apiVersion` *string*	`provisioning.dpu.nvidia.com/v1alpha1`		
`kind` *string*	`BFB`		
`metadata` *ObjectMeta*	Refer to Kubernetes API documentation for fields of `metadata`.		
`spec` *BFBSpec*			
`status` *BFBStatus*		{ phase:Initializing }	

BFBList

BFBList contains a list of BFB

Field	Description	Default	Validation
`apiVersion` *string*	`provisioning.dpu.nvidia.com/v1alpha1`		

Field	Description	Default	Validation
`kind` *string*	`BFBList`		
`metadata` *ListMeta*	Refer to Kubernetes API documentation for fields of `metadata`.		
`items` *BFB array*			

BFBPhase

Underlying type:string

BFBPhase describes current state of BFB CR. Only one of the following state may be specified. Default is Initializing.

Validation: - Enum: [Initializing Downloading Ready Deleting Error]

Appears in: - BFBStatus

Field	Description
`Initializing`	BFB CR is created
`Downloading`	Downloading BFB file
`Ready`	Finished downloading BFB file, ready for DPU to use
`Deleting`	Delete BFB
`Error`	Error happens during BFB downloading

BFBReference

BFBReference is a reference to a specific BFB

Appears in: - DPUTemplateSpec

Field	Description	Default	Validation
`name` *string*	Specifies name of the bfb CR to use for this DPU		

BFBSpec

BFBSpec defines the content of the BFB

Appears in: - BFB

Field	Description	Default	Validation
`fileName` *string*	Specifies the file name where the BFB is downloaded on the volume.		Pattern: `^[A-Za-z0-9\_\-\.]+\.bfb$`
`url` *string*	The url of the bfb image to download.		Pattern: `^(http\\|https)://.+$`

BFBStatus

BFBStatus defines the observed state of BFB

Appears in: - BFB

Field	Description	Default	Validation
`fileName` *string*	Filename is the name of the file where the BFB can be accessed on its volume. This is the same as `.spec.Filename` if set.		
`phase` *BFBPhase*	The current state of BFB.	Initializing	Enum: [Initializing Downloading Ready Deleting Error]
`versions` *BFBVersions*	BFB versions - BSP, DOCA, UEFI and ATF Holds detailed version information for each component within the BFB		

BFBVersions

BFBVersions represents the version information for BFB components.

Appears in: - BFBStatus

Field	Description	Default	Validation
`bsp` *string*	BSP (Board Support Package) version. This field stores the version of the BSP, which provides essentialsupport and drivers for the hardware platform.		
`doca` *string*	DOCA version Specifies the version of NVIDIA's Data Center-on-a-Chip Architecture (DOCA),a platform for developing applications on DPUs		
`uefi` *string*	UEFI (Unified Extensible Firmware Interface) version. Indicates the UEFI firmware version, which is responsible for bootingthe operating system and initializing hardware components		
`atf` *string*	ATF (Arm Trusted Firmware) version. Contains the version of ATF, which provides a secure runtime environment		

ClusterEndpointSpec

Appears in: - DPUClusterSpec

Field	Description	Default	Validation
`keepalived` *KeepalivedSpec*	Keepalived configures the keepalived that will be deployed for the cluster control-plane		

ClusterPhase

Underlying type:string

ClusterPhase describes current state of DPUCluster. Only one of the following state may be specified. Default is Pending.

Validation: - Enum: [Pending Creating Ready NotReady Failed]

Appears in: - DPUClusterStatus

Field	Description
`Pending`	
`Creating`	
`Ready`	
`NotReady`	
`Failed`	

ClusterSpec

Appears in: - DPUTemplateSpec

Field	Description	Default	Validation
`nodeLabels` *object (keys:string, values:string)*	NodeLabels specifies the labels to be added to the node.		

ConfigFile

Appears in: - DPUFlavorSpec

Field	Description	Default	Validation
`path` *string*	Path is the path of the file to be written.		
`operation` *DPUFlavorFileOp*	Operation is the operation to be performed on the file.		Enum: [override append]
`raw` *string*	Raw is the raw content of the file.		
`permissions` *string*	Permissions are the permissions to be set on the file.		

ContainerdConfig

Appears in: - DPUFlavorSpec

Field	Description	Default	Validation
`registryEndpoint` *string*	RegistryEndpoint is the endpoint of the container registry.		

DMSAddress

DMSAddress represents the IP and Port configuration for DMS.

Appears in: - DPUNodeSpec

Field	Description	Default	Validation
`ip` *string*	IP address in IPv4 format.		Format: ipv4
`port` *integer*	Port number.		Minimum: 1

DPU

DPU is the Schema for the dpus API

Appears in: - DPUList

Field	Description	Default	Valida
`apiVersion` *string*	`provisioning.dpu.nvidia.com/v1alpha1`		
`kind` *string*	`DPU`		
`metadata` *ObjectMeta*	Refer to Kubernetes API documentation for fields of `metadata`.		

Field	Description	Default	Valida
`spec` *DPUSpec*			
`status` *DPUStatus*		{ phase:Initializing }	

DPUCluster

DPUCluster is the Schema for the dpuclusters API

Appears in: - DPUClusterList

Field	Description	Default	Vali
`apiVersion` *string*	`provisioning.dpu.nvidia.com/v1alpha1`		
`kind` *string*	`DPUCluster`		
`metadata` *ObjectMeta*	Refer to Kubernetes API documentation for fields of `metadata`.		
`spec` *DPUClusterSpec*			
`status` *DPUClusterStatus*		{ phase:Pending }	

DPUClusterList

DPUClusterList contains a list of DPUCluster

Field	Description	Default	Validation
`apiVersion` *string*	`provisioning.dpu.nvidia.com/v1alpha1`		
`kind` *string*	`DPUClusterList`		

Field	Description	Default	Validation
`metadata` *ListMeta*	Refer to Kubernetes API documentation for fields of `metadata`.		
`items` *DPUCluster array*			

DPUClusterSpec

DPUClusterSpec defines the desired state of DPUCluster

Appears in: - DPUCluster

Field	Description	Default	Validation
`type` *string*	Type of the cluster with few supported values static - existing cluster that is deployed by user. For DPUCluster of this type, the kubeconfig field must be set.kamaji - DPF managed cluster. The kamaji-cluster-manager will create a DPU cluster on behalf of this CR.$(others) - any string defined by ISVs, such type names must start with a prefix.		Pattern: `kamaji\\|static\\|[^/]+/.*`
`maxNodes` *integer*	MaxNodes is the max amount of node in the cluster	1000	Maximum: 1000 Minimum: 1
`kubeconfig` *string*	Kubeconfig is the secret that contains the admin kubeconfig		
`clusterEndpoint` *ClusterEndpointSpec*	ClusterEndpoint contains configurations of the cluster entry point		

DPUClusterStatus

DPUClusterStatus defines the observed state of DPUCluster

Appears in: - DPUCluster

Field	Description	Default	Validation
`phase` *ClusterPhase*		Pending	Enum: [Pending Creating Ready NotReady Failed]
`version` *string*	Version is the K8s control-plane version of the cluster		
`conditions` *Condition array*			

DPUDevice

DPUDevice is the Schema for the dpudevices API

Appears in: - DPUDeviceList

Field	Description	Default	Validation
`apiVersion` *string*	`provisioning.dpu.nvidia.com/v1alpha1`		
`kind` *string*	`DPUDevice`		
`metadata` *ObjectMeta*	Refer to Kubernetes API documentation for fields of `metadata`.		
`spec` *DPUDeviceSpec*			
`status` *DPUDeviceStatus*			

DPUDeviceList

DPUDeviceList contains a list of DPUDevices

Field	Description	Default	Validation
`apiVersion` *string*	`provisioning.dpu.nvidia.com/v1alpha1`		
`kind` *string*	`DPUDeviceList`		
`metadata` *ListMeta*	Refer to Kubernetes API documentation for fields of `metadata`.		
`items` *DPUDevice array*			

DPUDeviceSpec

DPUDeviceSpec defines the content of DPUDevice

Appears in: - DPUDevice

Field	Description	Default	Validation
`psid` *string*	PSID is the Product Serial ID of the device. It's used to track the device's lifecycle and for inventory management.This value is immutable and should not be changed once set.Example: "MT_0001234567", "MT25066004C7"Deprecated: This field is deprecated and will be removed in a future version. Use status.psid instead.		Pattern: `^MT_?[A-Z0-9]+$`
`serialNumber` *string*	SerialNumber is the serial number of the device. It's used to track the device's lifecycle and for inventory management.This value is immutable and should not be changed once set.Example: "MT_0001234567", "MT25066004C7"		

Field	Description	Default	Validation
`opn` *string*	OPN is the Ordering Part Number of the device. It's used to track the device's compatibility with different software versions.This value is immutable and should not be changed once set.Example: "900-9D3B4-00SV-EA0"Deprecated: This field is deprecated and will be removed in a future version. Use status.opn instead.		Pattern: `^\d\{3\}-[A-Z0-9]\{5\}-[A-Z0-9]\{4\}-[A-Z0-9]\{3\}$`
`bmcIp` *string*	BMCIP is the IP address of the BMC (Base Management Controller) on the device. This is used for remote management and monitoring of the device.This value is immutable and should not be changed once set.Example: "10.1.2.3"		Format: ipv4
`bmcPort` *integer*	BMCPort is the port number of the BMC (Base Management Controller) on the device. This is used for remote management and monitoring of the device.This value is immutable and should not be changed once set.Example: 443	443	Minimum: 1
`numberOfPFs` *integer*	NumberOfPFs is the number of PFs on the device. This value is immutable and should not be changed once set.Example: 1	1	Minimum: 1
`pf0Name` *string*	PF0Name is the name of the PF0 on the device. This value is immutable and should not be changed once set.Example: "eth0"Deprecated: This field is deprecated and will be removed in a future version. Use status.pf0Name instead.		

DPUDeviceStatus

Appears in: - DPUDevice

Field	Description	Default	Validation
`psid` *string*	PSID is the Product Serial ID of the device. It's used to track the device's lifecycle and for inventory management.This value is discovered and should not be changed once set.Example: "MT_0001234567", "MT25066004C7"		Pattern: `^MT_?[A-Z0-9]+$`
`serialNumber` *string*	SerialNumber is the serial number of the device. It's used to track the device's lifecycle and for inventory management.This value is discovered and should not be changed once set.Example: "MT_0001234567", "MT25066004C7"		
`opn` *string*	OPN is the Ordering Part Number of the device. It's used to track the device's compatibility with different software versions.This value is discovered and should not be changed once set.Example: "900-9D3B4-00SV-EA0"		
`bmcIp` *string*	BMCIP is the IP address of the BMC (Base Management Controller) on the device. This is used for remote management and monitoring of the device.This value is discovered and should not be changed once set.Example: "10.1.2.3"		Format: ipv4
`bmcPort` *integer*	BMCPort is the port number of the BMC (Base Management Controller) on the device. This is used for remote management and monitoring of the device.This value is immutable and should not be changed once set.Example: 443	443	Minimum: 1
`pciAddress` *string*	PCIAddress is the PCI address of the device in the host system. Example: "0000-03-00", "03-00"		

Field	Description	Default	Validation
`pf0Name` *string*	PF0Name is the name of the PF0 on the device. Example: "eth0"		
`pf0Mac` *string*	PF0MAC is the MAC address of the PF0 on the device. Example: "00:00:00:00:00:00"		Pattern: `^([0-9A-Fa-f]\{2\}[:-])\{5\}([0-9A-Fa-f]\{2\})$`
`conditions` *Condition array*			

DPUDiscovery

Appears in: - DPUDiscoveryList

Field	Description	Default	Validatic
`apiVersion` *string*	`provisioning.dpu.nvidia.com/v1alpha1`		
`kind` *string*	`DPUDiscovery`		
`metadata` *ObjectMeta*	Refer to Kubernetes API documentation for fields of `metadata`.		
`spec` *DPUDiscoverySpec*			
`status` *DPUDiscoveryStatus*			

DPUDiscoveryList

DPUDiscoveryList contains a list of DPUDiscovery types

Field	Description	Default	Validation
`apiVersion` *string*	`provisioning.dpu.nvidia.com/v1alpha1`		
`kind` *string*	`DPUDiscoveryList`		
`metadata` *ListMeta*	Refer to Kubernetes API documentation for fields of `metadata`.		
`items` *DPUDiscovery array*			

DPUDiscoverySpec

DPUDiscoverySpec defines the desired state of DPUDiscovery

Appears in: - DPUDiscovery

Field	Description	Default	Validation
`ipRangeSpec` *IPRangeValidationSpec*	IPRange defines the range of IP addresses to scan		
`scanInterval` *Duration*	ScanInterval defines how often to perform the scan	1h	
`workers` *integer*	Workers defines the number of workers to use for the scan (default 1 worker for each 255 IPs in the range)		

DPUDiscoveryStatus

DPUDiscoveryStatus defines the observed state of DPUDiscovery

Appears in: - DPUDiscovery

Field	Description	Default	Validation
`lastScanTime` *Time*	LastScanTime is the timestamp of the last successful scan		
`foundDPUs` *integer*	FoundDPUs is the list of discovered DPU BMC IPs		

DPUFLavorSysctl

Appears in: - DPUFlavorSpec

Field	Description	Default	Validation
`parameters` *string array*	Parameters are the sysctl parameters to be set.		

DPUFlavor

DPUFlavor is the Schema for the dpuflavors API

Appears in: - DPUFlavorList

Field	Description	Default	Validation
`apiVersion` *string*	`provisioning.dpu.nvidia.com/v1alpha1`		
`kind` *string*	`DPUFlavor`		
`metadata` *ObjectMeta*	Refer to Kubernetes API documentation for fields of `metadata`.		
`spec` *DPUFlavorSpec*			

DPUFlavorFileOp

Underlying type:string

DPUFlavorFileOp defines the operation to be performed on the file

Validation: - Enum: [override append]

Appears in: - ConfigFile

Field	Description
`override`	
`append`	

DPUFlavorGrub

Appears in: - DPUFlavorSpec

Field	Description	Default	Validation
`kernelParameters` *string array*	KernelParameters are the kernel parameters to be set in the grub configuration.		

DPUFlavorList

DPUFlavorList contains a list of DPUFlavor

Field	Description	Default	Validation
`apiVersion` *string*	`provisioning.dpu.nvidia.com/v1alpha1`		
`kind` *string*	`DPUFlavorList`		
`metadata` *ListMeta*	Refer to Kubernetes API documentation for fields of `metadata`.		

Field	Description	Default	Validation
`items` *DPUFlavor array*			

DPUFlavorOVS

Appears in: - DPUFlavorSpec

Field	Description	Default	Validation
`rawConfigScript` *string*	RawConfigScript is the raw configuration script for OVS.		

DPUFlavorSpec

DPUFlavorSpec defines the content of DPUFlavor

Appears in: - DPUFlavor

Field	Description	Default	Validation
`grub` *DPUFlavorGrub*	Grub contains the grub configuration for the DPUFlavor.		
`sysctl` *DPUFLavorSysctl*	Sysctl contains the sysctl configuration for the DPUFlavor.		
`nvconfig` *NVConfig array*	NVConfig contains the global DPU-wide configuration (firmware settings, global device parameters). This applies to the DPU device and should not overlap with per-interface NVConfig settings.		

Field	Description	Default	Validation
`ovs` *DPUFlavorOVS*	OVS contains the OVS configuration for the DPUFlavor.		
`bfcfgParameters` *string array*	BFCfgParameters are the parameters to be set in the bf.cfg file.		
`configFiles` *ConfigFile array*	ConfigFiles are the files to be written on the DPU.		
`containerdConfig` *ContainerdConfig*	ContainerdConfig contains the configuration for containerd.		
`dpuResources` *ResourceList*	DPUResources indicates the minimum amount of resources needed for a BFB with that flavor to be installed on a DPU. Using this field, the controller can understand if that flavor can be installed on a particular DPU. Itshould be set to the total amount of resources the system needs + the resources that should be made available forDPUServices to consume.		
`systemReservedResources` *ResourceList*	SystemReservedResources indicates the resources that are consumed by the system (OS, OVS, DPF system etc) and are not made available for DPUServices to consume. DPUServices can consume the difference between DPUResources andSystemReservedResources. This field must not be specified if dpuResources are not specified.		

Field	Description	Default	Validation
`dpuMode` *DpuModeType*	Specifies the DPU Mode type: one of dpu,zero-trust	dpu	Enum: [dpu zero-trust]
`hostNetworkInterfaceConfigs` *NetworkInterfaceConfig array*	HostNetworkInterfaceConfigs contains the configuration for the host-side network interfaces.		

DPUList

DPUList contains a list of DPU

Field	Description	Default	Validation
`apiVersion` *string*	`provisioning.dpu.nvidia.com/v1alpha1`		
`kind` *string*	`DPUList`		
`metadata` *ListMeta*	Refer to Kubernetes API documentation for fields of `metadata`.		
`items` *DPU array*			

DPUNode

DPUNode is the Schema for the dpunodes API

Appears in: - DPUNodeList

Field	Description	Default	Validation
`apiVersion` *string*	`provisioning.dpu.nvidia.com/v1alpha1`		
`kind` *string*	`DPUNode`		

Field	Description	Default	Validation
`metadata` *ObjectMeta*	Refer to Kubernetes API documentation for fields of `metadata`.		
`spec` *DPUNodeSpec*			
`status` *DPUNodeStatus*			

DPUNodeList

DPUNodeList contains a list of DPUNode

Field	Description	Default	Validation
`apiVersion` *string*	`provisioning.dpu.nvidia.com/v1alpha1`		
`kind` *string*	`DPUNodeList`		
`metadata` *ListMeta*	Refer to Kubernetes API documentation for fields of `metadata`.		
`items` *DPUNode array*			

DPUNodeMaintenance

DPUNodeMaintenance is the Schema for the dpunodemaintenances API

Appears in: - DPUNodeMaintenanceList

Field	Description	Default
`apiVersion` *string*	`provisioning.dpu.nvidia.com/v1alpha1`	
`kind` *string*	`DPUNodeMaintenance`	
`metadata` *ObjectMeta*	Refer to Kubernetes API documentation for fields of `metadata`.	

Field	Description	Default
`spec` *DPUNodeMaintenanceSpec*		
`status` *DPUNodeMaintenanceStatus*		

DPUNodeMaintenanceList

DPUNodeMaintenanceList contains a list of DPUNodeMaintenance

Field	Description	Default	Validat
`apiVersion` *string*	`provisioning.dpu.nvidia.com/v1alpha1`		
`kind` *string*	`DPUNodeMaintenanceList`		
`metadata` *ListMeta*	Refer to Kubernetes API documentation for fields of `metadata`.		
`items` *DPUNodeMaintenance array*			

DPUNodeMaintenanceSpec

DPUNodeMaintenanceSpec is the specification of the DPUNodeMaintenance object

Appears in: - DPUNodeMaintenance

Field	Description	Default	Validation
`dpuNodeName` *string*	DPUNodeName is the name of the DPUNode that is being maintained.		
`nodeEffect` *NodeEffect*	NodeEffect is the effect to be applied to the node.		
`requestor` *string array*	Requestor is the list of consumers for the maintenance.		

DPUNodeMaintenanceStatus

DPUNodeMaintenanceStatus defines the observed state of DPUNodeMaintenance

Appears in: - DPUNodeMaintenance

Field	Description	Default	Va
`conditions` *Condition array*	Conditions reflect the status of the object		
`nodeEffectSyncStartTime` *Time*	NodeEffectSyncStartTime is the time when the node effect sync started.		
`multiDPUOperationsSyncWaitTime` *Duration*	MultiDPUOperationsSyncWaitTime is the wait time between DPUs on the same node.		
`maxUnavailableDPUNodes` *integer*	MaxUnavailableDPUNodes is the maximum number of DPUNodes that are unavailable during the node effect period.		M 1

DPUNodeSpec

DPUNodeSpec defines the desired state of DPUNode

Appears in: - DPUNode

Field	Description	Default	Validation
`nodeRebootMethod` *NodeRebootMethod*	Defines the method for rebooting the host. One of the following options can be chosen for this field:- "external": Reboot the host via an external means, not controlled by theDPU controller.- "script": Reboot the host by executing a custom script.- "hostAgent": Use the host agent to reboot the host."hostAgent" is the default value.	{ hostAgent:map[] }	
`nodeDMSAddress` *DMSAddress*	The IP address and port where the DMS is exposed. Only applicable if dpuInstallInterface is set to gNOI. Deprecated: this field is no longer used.		
`dpus` *DPURef array*	A map containing names of each DPUDevice attached to the node.		

DPUNodeStatus

DPUNodeStatus defines the observed state of DPUNode

Appears in: - DPUNode

Field	Description	Default	Validation
`conditions` *Condition array*	Conditions represent the latest available observations of an object's state.		Type: array
`dpuInstallInterface` *string*	The name of the interface which will be used to install the bfb image, can be one of hostAgent,redfish		Enum: [gNOI hostAgent redfish]

Field	Description	Default	Validation
`kubeNodeRef` *string*	The name of the Kubernetes Node object that this DPUNode represents. This field is optional and only relevant if the x86 host is part of the DPF Kubernetes cluster.		
`rebootInProgress` *boolean*	RebootInProgress indicates if the node is in the process of rebooting.		

DPUPhase

Underlying type:string

DPUPhase describes current state of DPU. Only one of the following state may be specified. Default is Initializing.

Validation: - Enum: [Initializing Node Effect Pending Config FW Parameters Prepare BFB OS Installing DPU Cluster Config Host Network Configuration Ready Error Deleting Rebooting Initialize Interface Checking Host Reboot Required Node Effect Removal]

Appears in: - DPUSetStatus - DPUStatus

Field	Description
`Initializing`	DPUInitializing is the first phase after the DPU is created.
`Node Effect`	DPUNodeEffect means the controller will handle the node effect provided by the user.
`Pending`	DPUPending means the controller is waiting for the BFB to be ready.
`Prepare BFB`	DPUPrepareBFB means the controller is preparing the BFB and bf.cfg to be installed to DPU
`Config FW Parameters`	DPUConfigFWParameters means the controller will manipulate DPU firmware, e.g., set DPU mode, check firmware version
`Initialize Interface`	DPUInitializeInterface means the controller will intitialize the interface used to provision the DPUs, e.g., create the DMS pod, set up RedFish account.

Field	Description
`OS Installing`	DPUOSInstalling means the controller will provision the DPU through the DMS gNOI interface.
`DPU Cluster Config`	DPUClusterConfig means the node configuration and Kubernetes Node join procedure are in progress .
`Host Network Configuration`	DPUHostNetworkConfiguration means the host network configuration is running.
`Node Effect Removal`	DPUNodeEffectRemoval means the controller will remove the node effect from the DPU.
`Ready`	DPUReady means the DPU is ready to use.
`Error`	DPUError means error occurred.
`Deleting`	DPUDeleting means the DPU CR will be deleted, controller will do some cleanup works.
`Rebooting`	DPURebooting means the host of DPU is rebooting.
`Checking Host Reboot Required`	DPUCheckingHostRebootNeed means the checking of whether the host required a reboot.

DPURef

Appears in: - DPUNodeSpec

Field	Description	Default	Validation
`name` *string*	Name of the DPU device.		MinLength: 1

DPUSet

DPUSet is the Schema for the dpusets API

Appears in: - DPUSetList

Field	Description	Default	Validation
`apiVersion` *string*	`provisioning.dpu.nvidia.com/v1alpha1`		
`kind` *string*	`DPUSet`		
`metadata` *ObjectMeta*	Refer to Kubernetes API documentation for fields of `metadata`.		
`spec` *DPUSetSpec*			
`status` *DPUSetStatus*			

DPUSetList

DPUSetList contains a list of DPUSet

Field	Description	Default	Validation
`apiVersion` *string*	`provisioning.dpu.nvidia.com/v1alpha1`		
`kind` *string*	`DPUSetList`		
`metadata` *ListMeta*	Refer to Kubernetes API documentation for fields of `metadata`.		
`items` *DPUSet array*			

DPUSetSpec

DPUSetSpec defines the desired state of DPUSet

Appears in: - DPUSet

Field	Description	Default	Validation
`strategy` *DPUSetStrategy*	The rolling update strategy to use to updating existing DPUs with new ones.		
`dpuNodeSelector` *LabelSelector*	Select the DPUNodes with specific labels		
`dpuSelector` *object (keys:string, values:string)*	Select the DPU with specific labels		
`dpuTemplate` *DPUTemplate*	Object that describes the DPU that will be created if insufficient replicas are detected		

DPUSetStatus

DPUSetStatus defines the observed state of DPUSet

Appears in: - DPUSet

Field	Description	Default	Validation
`dpuStatistics` *object (keys:DPUPhase, values:integer)*	DPUStatistics is a map of DPUPhase to the number of DPUs in that phase.		
`conditions` *Condition array*	Conditions reflect the status of the object		
`observedGeneration` *integer*	ObservedGeneration records the Generation observed on the object the last time it was patched.		

DPUSetStrategy

Appears in: - DPUSetSpec

Field	Description	Default	Validation
`type` *StrategyType*	Can be "OnDelete" or "RollingUpdate".	OnDelete	Enum: [OnDelete RollingUpdate]
`rollingUpdate` *RollingUpdateDPU*	Rolling update config params. Present only if StrategyType = RollingUpdate.		

DPUSpec

DPUSpec defines the desired state of DPU

Appears in: - DPU

Field	Description	Default	Validation
`dpuNodeName` *string*	Specifies the DPUNode this DPU belongs to		
`dpuDeviceName` *string*	Specifies the name of the DPUDevice this DPU is associated with		MinLength: 1
`bfb` *string*	Specifies name of the bfb CR to use for this DPU		
`serialNumber` *string*	The serial number of the DPU		MinLength: 1
`pciAddress` *string*	The PCI device related DPU Example: "0000-03-00", "03-00"		Pattern: `^([0-9a-fA-F]\{4\}[-])?[0-9a-fA-F]\{2\}[-][0-9a-fA-F]\{2\}$`
`nodeEffect` *NodeEffect*	Specifies how changes to the DPU should affect the Node	{ drain:true }	
`cluster` *K8sCluster*	Specifies details on the K8S cluster to join		

Field	Description	Default	Validation
`dpuFlavor` *string*	DPUFlavor is the name of the DPUFlavor that will be used to deploy the DPU.		MinLength: 1
`bmcIP` *string*	BMCIP is the ip address of the DPU BMC Deprecated: Use BMCIP from DPUDevice instead.		

DPUStatus

DPUStatus defines the observed state of DPU

Appears in: - DPU

Field	Description	Default	Validation
`phase` *DPUPhase*	The current state of DPU.	Initializing	Enum: [Initializing Node Effect Pending Config FW Parameters Prepare BFB OS Installing DPU Cluster Config Host Network Configuration Ready Error Deleting Rebooting Initialize Interface Checking Host Reboot Required Node Effect Removal]
`conditions` *Condition array*			
`bfbFile` *string*	BFBFile is the path to the BFB file		

Field	Description	Default	Validation
`bfCFGFile` *string*	BFCFGFile is the path to the bf.cfg		
`bfbVersion` *string*	bfb version of this DPU		
`dpfVersion` *string*	DPF version used to install this DPU		
`pciDevice` *string*	pci device information of this DPU		
`requiredReset` *boolean*	whether require reset of DPU		
`firmware` *Firmware*	the firmware information of DPU		
`addresses` *NodeAddress array*	The DPU node's IP addresses		
`dpuInstallInterface` *string*	the name of the interface which will be used to install the bfb image, and communicate with DPU, can be one of hostAgent,redfish		Enum: [gNOI hostAgent redfish]
`postProvisioningNodeEffect` *boolean*	Indicates that node effect was triggered by post-provisioning label changes		
`observedGeneration` *integer*	ObservedGeneration records the Generation observed on the object the last time it was patched.		

DPUTemplate

DPUTemplate is a template for DPU

Appears in: - DPUSetSpec

Field	Description	Default	Validation
`annotations` *object (keys:string, values:string)*	Annotations specifies annotations which are added to the DPU.		
`spec` *DPUTemplateSpec*	Spec specifies the DPU specification.		

DPUTemplateSpec

Appears in: - DPUTemplate

Field	Description	Default	Validation
`bfb` *BFBReference*	Specifies a BFB CR		
`nodeEffect` *NodeEffect*	Specifies how changes to the DPU should affect the Node	{ drain:true }	
`cluster` *ClusterSpec*	Specifies details on the K8S cluster to join		
`dpuFlavor` *string*	DPUFlavor is the name of the DPUFlavor that will be used to deploy the DPU.		

DpuModeType

Underlying type:string

DpuModeType defines the mode of the DPU

Validation: - Enum: [dpu zero-trust]

Appears in: - DPUFlavorSpec

Field	Description
`dpu`	
`zero-trust`	

External

Appears in: - NodeRebootMethod

Firmware

Appears in: - DPUStatus

Field	Description	Default	Validation
`bmc` *string*	BMC is the used BMC firmware version		
`nic` *string*	NIC is the used NIC firmware version		
`uefi` *string*	UEFI is the used UEFI firmware version		

GNOI

Appears in: - NodeRebootMethod

HostAgent

Appears in: - NodeRebootMethod

IPRange

IPRange represents a range of IP addresses to scan

Appears in: - IPRangeValidationSpec

Field	Description	Default	Validation
`startIP` *string*			Pattern: `^((25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.)\{3\}(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)$`
`endIP` *string*			Pattern: `^((25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.)\{3\}(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)$`
`port` *integer*	Port defines the port to on which BMC is listening	443	Maximum: 65535 Minimum: 1

IPRangeValidationSpec

IPRangeValidationSpec defines the desired state of IPRangeValidation IPRange defines the IP range to validate

Appears in: - DPUDiscoverySpec

Field	Description	Default	Validation
`ipRange` *IPRange*			

K8sCluster

Appears in: - DPUSpec

Field	Description	Default	Validation
`name` *string*	Name is the name of the DPUs Kubernetes cluster		

Field	Description	Default	Validation
`namespace` *string*	Namespace is the tenants namespace name where the Kubernetes cluster will be deployed		
`nodeLabels` *object (keys:string, values:string)*	NodeLabels define the labels that will be added to the nodes.		

KeepalivedSpec

Appears in: - ClusterEndpointSpec

Field	Description	Default	Validation
`vip` *string*	VIP is the virtual IP owned by the keepalived instances		
`virtualRouterID` *integer*	VirtualRouterID is the virtual_router_id in keepalived.conf		Maximum: 255 Minimum: 1
`interface` *string*	Interface specifies on which interface the VIP should be assigned		MinLength: 1
`nodeSelector` *object (keys:string, values:string)*	NodeSelector is used to specify a subnet of control plane nodes to deploy keepalived instances. Note: keepalived instances are always deployed on control plane nodes		

NVConfig

Appears in: - DPUFlavorSpec - NetworkInterfaceConfig

Field	Description	Default	Validation
`device` *string*	Device is the device to which the configuration applies. If not specified, the configuration applies to all.		

Field	Description	Default	Validation
`parameters` *string array*	Parameters are the parameters to be set for the device.		
`hostPowerCycleRequired` *boolean*	HostPowerCycleRequired indicates if the host needs to be power cycled after applying the configuration. Deprecated: This field is unused and deprecated. It will be removed with the release v26.4.0.		

NetworkInterfaceConfig

NetworkInterfaceConfig defines the configuration for a network interface

Appears in: - DPUFlavorSpec

Field	Description	Default	Validation
`mtu` *integer*	MTU is the MTU value to be set on the network interface.		Maximum: 9216 Minimum: 1280
`dhcp` *boolean*	DHCP is the DHCP configuration for the network interface.		
`portNumber` *integer*	PortNumber identifies which port this configuration applies to.		Maximum: 1 Minimum: 0
`nvconfig` *NVConfig*	NVConfig contains port-specific configuration for this network interface. This configuration is applied in addition to the global NVConfig settings in DPUFlavorSpec.Both global and per-interface NVConfig settings can coexist without collision.		

NodeEffect

NodeEffect is the effect the DPU has on Nodes during provisioning. Only one of Taint, NoEffect, CustomLabel, Drain, CustomAction, Hold can be set.

Appears in: - DPUNodeMaintenanceSpec - DPUSpec - DPUTemplateSpec

Field	Description
`taint` *Taint*	Add specify taint on the DPU node
`noEffect` *boolean*	Do not do any action on the DPU node
`customLabel` *object (keys:string, values:string)*	Add specify labels on the DPU node
`drain` *boolean*	Drain the K8s host node by NodeMainten operator
`customAction` *string*	Name of a config map which contains a p definition to run which will apply the node The pod is expected to exit when node ef if pod terminates with error then DPU wo an error phase.The DPUNode's name will l as an environment variable, named as DPUNODE_NAME, to each container and in the pod.The labels and annotations of D be exported in `/etc/dpu/dpf-pod-inf` and `/etc/dpu/dpf-pod-info/annota` accordingly; the volume name `dpf-pod-` used to mount the labels and annotations confliction for env or volume, the controll export the name or labels/annotations of accordingly.
`hold` *boolean*	Places annotation `wait-for-external-nodeeffect` and to be removed this is the default behavior in a non K8S e
`force` *boolean*	Force is the flag to indicate if the node ef be applied immediately. If true, dpfOperatorConfig.multiDPUOperationsS and dpfOperatorConfig.maxUnavailableD be ignored when applying node effect for DPUNodeMaintenance CR

Field	Description
`applyOnLabelChange` *boolean*	Apply node effect when labels change on object When set to true, label changes in Ready trigger node effect logic
`nodeMaintenanceAdditionalRequestors` *string array*	Additional requestors to be added to the NvidiaNodeMaintenance CR when Drain is

NodeRebootMethod

NodeRebootMethod defines the desired reboot method

Appears in: - DPUNodeSpec

Field	Description	Default	Validation
`gNOI` *GNOI*	Use the DPU's DMS interface to reboot the host. Deprecated: Use HostAgent instead.		
`hostAgent` *HostAgent*	Use the HostAgent to reboot the host.		
`external` *External*	Reboot the host via an external means, not controlled by the DPU controller.		
`script` *Script*	Reboot the host by executing a custom script. This field defined which ConfigMap store the custom script. The ConfigMap should include a pod template of Job object under the `pod-template` key.That pod template will be put in a Job object to be executed.		

RollingUpdateDPU

RollingUpdateDPU is the rolling update strategy for a DPUSet.

Appears in: - DPUSetStrategy

Field	Description	Default	Validation
`maxUnavailable` *IntOrString*	MaxUnavailable is the maximum number of DPUs that can be unavailable during the update. Deprecated: This field is deprecated and will be removed with v26.4.0.		

Script

Appears in: - NodeRebootMethod

Field	Description	Default	Validation
`name` *string*			MinLength: 1

StrategyType

Underlying type:string

StrategyType describes strategy to use to reprovision existing DPUs. Default is "OnDelete".

Validation: - Enum: [OnDelete RollingUpdate]

Appears in: - DPUSetStrategy

Field	Description
`OnDelete`	New DPU CR will only be created when you manually delete old DPU CR.
`RollingUpdate`	Gradually scale down the old DPUs and scale up the new one.

UpgradePolicy

UpgradePolicy is the policy for the upgrade of the DPUSet.

Appears in: - NodeEffect

Field	Description	Default	Valida
`applyOnLabelChange` *boolean*	Apply node effect when labels change on the DPU object When set to true, label changes in Ready state will trigger node effect logic	false	
`nodeMaintenanceAdditionalRequestors` *string array*	Additional requestors to be added to the NvidiaNodeMaintenance CR when Drain is selected		

storage.dpu.nvidia.com/v1alpha1

Package v1alpha1 contains API Schema definitions for the storage v1alpha1 API group

Resource Types

- DPUStoragePolicy
- DPUStoragePolicyList
- DPUStorageVendor
- DPUStorageVendorList
- DPUVolume
- DPUVolumeAttachment
- DPUVolumeAttachmentList
- DPUVolumeList

- SVVolumeAttachment
- SVVolumeAttachmentList
- Volume
- VolumeAttachment
- VolumeAttachmentList
- VolumeList

AttachmentStatusDPU

AttachmentStatusDPU describe the information of DPU volume

Appears in: - DPUVolumeAttachmentStatus

Field	Description	Default	Validation
`pciAddress` *string*	PCI device address in the following format: (bus:device.function)		
`deviceName` *string*	The name of the device that was created by the storage vendor plugin		
`nvmeAttrs` *NVMEAttrs*	The attributes of the emulated NVME function		
`virtioFSAttrs` *VirtioFSAttrs*	The attributes of the emulated VirtioFS function		

BdevAttrs

BdevAttrs represents the attributes of the underlying block device

Appears in: - VolumeAttachmentStatusDPU

Field	Description	Default	Validation
`nvmeNsID` *integer*	The namespace ID within the NVME controller		
`nvmeUUID` *string*	The nvme namespace UUID		

CSIReference

CSIReference reference to CSI object

Appears in: - VolumeSpecDPU

Field	Description	Default	Validation
`csiDriverName` *string*			
`storageClassName` *string*			
`pvcRef` *ObjectRef*			

CapacityRange

CapacityRange represents the capacity of the required storage space in bytes

Appears in: - VolumeRequest

Field	Description	Default	Validation
`request` *Quantity*			
`limit` *Quantity*			

DPUStoragePolicy

DPUStoragePolicy represents a DPUStoragePolicy CR

Appears in: - DPUStoragePolicyList

Field	Description	Default	Validation
`apiVersion` *string*	`storage.dpu.nvidia.com/v1alpha1`		
`kind` *string*	`DPUStoragePolicy`		

Field	Description	Default	Validation
`metadata` *ObjectMeta*	Refer to Kubernetes API documentation for fields of `metadata`.		
`spec` *DPUStoragePolicySpec*			
`status` *DPUStoragePolicyStatus*			

DPUStoragePolicyList

DPUStoragePolicyList contains a list of DPUStoragePolicy objects

Field	Description	Default	Validation
`apiVersion` *string*	`storage.dpu.nvidia.com/v1alpha1`		
`kind` *string*	`DPUStoragePolicyList`		
`metadata` *ListMeta*	Refer to Kubernetes API documentation for fields of `metadata`.		
`items` *DPUStoragePolicy array*			

DPUStoragePolicySpec

DPUStoragePolicySpec defines the desired state of DPUStoragePolicy

Appears in: - DPUStoragePolicy

Field	Description	Default	Validation
`dpuStorageVendors` *string array*	List of storage vendors		MinItems: 1
`parameters` *object (keys:string, values:string)*	Parameters supported by the policy	{ }	

Field	Description	Default	Validation
`selectionAlgorithm` *SelectionAlgorithm*	Selection algorithm used to select DPUStorageVendor	NumberVolumes	Enum: [Random NumberVolumes]

DPUStoragePolicyStatus

DPUStoragePolicyStatus defines the observed state of DPUStoragePolicy

Appears in: - DPUStoragePolicy

Field	Description	Default	Validation
`conditions` *Condition array*	Current service state conditions		
`observedGeneration` *integer*	ObservedGeneration records the Generation observed on the object the last time it was patched.		

DPUStorageVendor

DPUStorageVendor represents a StorageVendor CR on the DPU cluster.

Appears in: - DPUStorageVendorList

Field	Description	Default	Validatior
`apiVersion` *string*	`storage.dpu.nvidia.com/v1alpha1`		
`kind` *string*	`DPUStorageVendor`		
`metadata` *ObjectMeta*	Refer to Kubernetes API documentation for fields of `metadata`.		
`spec` *DPUStorageVendorSpec*			
`status` *DPUStorageVendorStatus*			

DPUStorageVendorList

DPUStorageVendorList contains a list of DPUStorageVendor

Field	Description	Default	Validation
`apiVersion` *string*	`storage.dpu.nvidia.com/v1alpha1`		
`kind` *string*	`DPUStorageVendorList`		
`metadata` *ListMeta*	Refer to Kubernetes API documentation for fields of `metadata`.		
`items` *DPUStorageVendor array*			

DPUStorageVendorSpec

DPUStorageVendorSpec defines the desired state of DPUStorageVendor

Appears in: - DPUStorageVendor

Field	Description	Default	Validation
`storageClassName` *string*	Storage vendor class name, deployed on the DPU K8S cluster.		MinLength: 1
`pluginName` *string*	Storage vendor DPU plugin name		MinLength: 1

DPUStorageVendorStatus

DPUStorageVendorStatus defines the observed state of DPUStorageVendor

Appears in: - DPUStorageVendor

Field	Description	Default	Validation
`dpuClusters` *ObjectReference array*	DPUClusters is the list of clusters on which the DPUStorageVendor is deployed.		
`conditions` *Condition array*	Conditions defines current service state.		
`observedGeneration` *integer*	ObservedGeneration records the Generation observed on the object the last time it was patched.		

DPUVolume

DPUVolume represents a DPUVolume CR.

Appears in: - DPUVolumeList

Field	Description	Default	Validation
`apiVersion` *string*	`storage.dpu.nvidia.com/v1alpha1`		
`kind` *string*	`DPUVolume`		
`metadata` *ObjectMeta*	Refer to Kubernetes API documentation for fields of `metadata`.		
`spec` *DPUVolumeSpec*			
`status` *DPUVolumeStatus*			

DPUVolumeAttachment

DPUVolumeAttachment represents a Volume CR on the DPU cluster.

Appears in: - DPUVolumeAttachmentList

Field	Description	Default	Valid
`apiVersion` *string*	`storage.dpu.nvidia.com/v1alpha1`		
`kind` *string*	`DPUVolumeAttachment`		
`metadata` *ObjectMeta*	Refer to Kubernetes API documentation for fields of `metadata`.		
`spec` *DPUVolumeAttachmentSpec*			
`status` *DPUVolumeAttachmentStatus*			

DPUVolumeAttachmentList

DPUVolumeAttachmentList contains a list of DPUVolumeAttachment

Field	Description	Default	Validation
`apiVersion` *string*	`storage.dpu.nvidia.com/v1alpha1`		
`kind` *string*	`DPUVolumeAttachmentList`		
`metadata` *ListMeta*	Refer to Kubernetes API documentation for fields of `metadata`.		
`items` *DPUVolumeAttachment array*			

DPUVolumeAttachmentSpec

DPUVolumeAttachmentSpec defines the desired state of DPUVolumeAttachment

Appears in: - DPUVolumeAttachment

Field	Description	Default	Validation
`dpuNodeName` *string*	DPUNodeName is the name of DPUNode object that represents the node to which the volume should be attached		MinLength: 1
`dpuVolumeName` *string*	DPUVolumeName is the name of DPUVolume object that represents the volume to be attached		MinLength: 1
`functionType` *FunctionType*	FunctionType is the type of the emulated function that should be used to attach the volume		Enum: [pf vf]
`hotplugFunction` *boolean*	HotplugFunction is a boolean flag that indicates if the emulated function should be hotplugged		

DPUVolumeAttachmentStatus

DPUVolumeAttachmentStatus defines the observed state of DPUVolumeAttachment

Appears in: - DPUVolumeAttachment

Field	Description	Default	Validation
`controllerAttached` *boolean*	Indicates the volume is successfully attached to by the Vendor CSI driver		
`dpuAttached` *boolean*	Indicates the volume is successfully attached to the node by DPU		
`attachmentMetadata` *object (keys:string, values:string)*	AttachmentMetadata contains the metadata of the volume attachment returned by the Vendor CSI driver		
`dpu` *AttachmentStatusDPU*	Details about the DPU attachment		
`message` *string*	The last error encountered during the attach operation, if any		
`conditions` *Condition array*	Conditions defines current service state.		

Field	Description	Default	Validation
`observedGeneration` *integer*	ObservedGeneration records the Generation observed on the object the last time it was patched.		

DPUVolumeList

DPUVolumeList contains a list of DPUVolume

Field	Description	Default	Validation
`apiVersion` *string*	`storage.dpu.nvidia.com/v1alpha1`		
`kind` *string*	`DPUVolumeList`		
`metadata` *ListMeta*	Refer to Kubernetes API documentation for fields of `metadata`.		
`items` *DPUVolume array*			

DPUVolumePhase

Underlying type:string

Appears in: - DPUVolumeStatus

Field	Description
`Pending`	used for DPUVolume that are not yet bound to a volume in the DPU cluster
`Bound`	used for DPUVolume that are bound to a volume in the DPU cluster

DPUVolumeSpec

DPUVolumeSpec defines the desired state of DPUVolume

Appears in: - DPUVolume

Field	Description	Default	Validation
`dpuStoragePolicyName` *string*	Name of the DPUStoragePolicyName object that will be used to create the volume.		MinLength: 1
`parameters` *object (keys:string, values:string)*	Additional parameters for the volume, these parameters are merged with the values from the DPUStoragePolicy object.	`{}`	
`accessModes` *PersistentVolumeAccessMode array*	Access modes define how the volume can be mounted. These modes are directly passed to the PersistentVolumeClaim created for the Vendor CSI Plugin selected by the DPUStoragePolicy.		MaxItems: 3 MinItems: 1
`resources` *VolumeResourceRequirements*	Resources represents the storage resources requested for the volume. These resource requirements are directly passed to the PersistentVolumeClaim created for the Vendor CSI Plugin selectedby the DPUStoragePolicy. Since volume resizing is not supported, modifications to the resource request are prohibited.		
`volumeMode` *PersistentVolumeMode*	Volume mode defines how the volume should be mounted and used. This value is directly passed to the PersistentVolumeClaim created for the Vendor CSI Plugin selected by the DPUStoragePolicy.	Filesystem	Enum: [Filesystem Block]

DPUVolumeState

DPUVolumeState defines the state of the volume.

Appears in: - DPUVolumeStatus

Field	Description	Default	Valida
`dpuCluster` *ObjectReference*	DPUCluster contains the reference to the DPUCluster object that was selected for volume creation.		
`parameters` *object (keys:string, values:string)*	Parameters contains the final set of parameters for volume creation, computed by merging the parameters from the DPUStoragePolicy object with user-provided parameters.		
`selectedDPUStorageVendorName` *string*	SelectedDPUStorageVendorName contains the name of the DPUStorageVendor object that was selected for volume creation.		
`storageVendorPluginName` *string*	StorageVendorPluginName contains the name of the storage vendor plugin deployed on the DPU cluster that was selected for volume creation.		
`storageClassName` *string*	StorageClassName contains the name of the storage class in the DPU cluster that was selected for volume creation.		
`csiDriverName` *string*	CSIDriverName contains the name of the CSI driver in the DPU cluster that was selected for volume creation.		
`persistentVolumeClaimRef` *ObjectReference*	PersistentVolumeClaimRef contains the reference to the PersistentVolumeClaim object in the DPU cluster that was created for the volume.		

Field	Description	Default	Valida
`volumeInfo` *VolumeInfo*	VolumeInfo contains a subset of fields from the PersistentVolume object created in the DPU cluster		

DPUVolumeStatus

DPUVolumeStatus defines the observed state of DPUVolume

Appears in: - DPUVolume

Field	Description	Default	Validation
`phase` *DPUVolumePhase*	Phase of the volume		Enum: [Pending Bound]
`state` *DPUVolumeState*	State of the volume. This field is managed by the controller. User usually do not need to set fields from this struct.		
`conditions` *Condition array*	Conditions defines current service state.		
`observedGeneration` *integer*	ObservedGeneration records the Generation observed on the object the last time it was patched.		

FSdevAttrs

FSdevAttrs represents the attributes of the underlying filesystem device

Appears in: - VolumeAttachmentStatusDPU

Field	Description	Default	Validation
`filesystemTag` *string*	Filesystem tag identified by SNAP on the host (used for the mount). Relevant for volume of type filesystem		

FunctionType

Underlying type:string

Appears in: - DPUVolumeAttachmentSpec - FunctionTypeConfig - VolumeAttachmentSpec

Field	Description
`pf`	FunctionTypePF is the PF function type
`vf`	FunctionTypeVF is the VF function type

FunctionTypeConfig

FunctionTypeConfig is the configuration for the emulated function that should be used to attach the volume

Appears in: - DPUVolumeAttachmentSpec - VolumeAttachmentSpec

Field	Description	Default	Validation
`functionType` *FunctionType*	FunctionType is the type of the emulated function that should be used to attach the volume		Enum: [pf vf]
`hotplugFunction` *boolean*	HotplugFunction is a boolean flag that indicates if the emulated function should be hotplugged		

NVMEAttrs

NVMEAttrs represents the attributes of the NVME emulated function

Appears in: - AttachmentStatusDPU

Field	Description	Default	Validation
`namespaceID` *integer*	The namespace ID within the NVME controller		
`namespaceUUID` *string*	The NVMe namespace UUID		

ObjectRef

ObjectRef reference to the object

Appears in: - CSIReference - VolumeAttachmentSpec - VolumeSource - VolumeSpec

Field	Description	Default	Validation
`kind` *string*			
`apiVersion` *string*			
`name` *string*			
`namespace` *string*			

ObjectReference

ObjectReference represents a reference to a Kubernetes object.

Appears in: - DPUStorageVendorStatus - DPUVolumeState

Field	Description	Default	Validation
`name` *string*	Name specifies the name of the referenced object		MinLength: 1
`namespace` *string*	Namespace specifies the namespace where the referenced object exists		MinLength: 1

SVVolumeAttachment

SVVolumeAttachment captures the intent to attach/detach the specified Volume to/from the specified node.

Appears in: - SVVolumeAttachmentList

Field	Description	Default	Validatior
`apiVersion` *string*	`storage.dpu.nvidia.com/v1alpha1`		
`kind` *string*	`SVVolumeAttachment`		
`metadata` *ObjectMeta*	Refer to Kubernetes API documentation for fields of `metadata`.		
`spec` *VolumeAttachmentSpec*			
`status` *VolumeAttachmentStatus*			

SVVolumeAttachmentList

SVVolumeAttachmentList contains a list of SVVolumeAttachment

Field	Description	Default	Validation
`apiVersion` *string*	`storage.dpu.nvidia.com/v1alpha1`		
`kind` *string*	`SVVolumeAttachmentList`		
`metadata` *ListMeta*	Refer to Kubernetes API documentation for fields of `metadata`.		
`items` *SVVolumeAttachment array*			

SelectionAlgorithm

Underlying type:string

SelectionAlgorithm represents the storage selection algorithm type

Appears in: - DPUStoragePolicySpec

Field	Description
`Random`	Random selection across the vendors defined in the StoragePolicy list.
`NumberVolumes`	Load-balancing on the number of volumes belonging to the StoragePolicy. The vendor (in the DPUStoragePolicy list) with the minimal number of volumes should be selected.

VirtioFSAttrs

VirtioFSAttrs represents the attributes of the VirtioFS emulated function

Appears in: - AttachmentStatusDPU

Field	Description	Default	Validation
`filesystemTag` *string*	Filesystem tag identified by SNAP on the host (used for the mount). Relevant for volume of type filesystem		

Volume

Volume represents a persistent volume on the DPU cluster. It maps between the tenant K8S persistent volume (PV) object on the tenant cluster into the actual volume on the DPU cluster. Volume is an internal API, it is not intended to be used by users.

Appears in: - VolumeList

Field	Description	Default	Validation
`apiVersion` *string*	`storage.dpu.nvidia.com/v1alpha1`		
`kind` *string*	`Volume`		

Field	Description	Default	Validation
`metadata` *ObjectMeta*	Refer to Kubernetes API documentation for fields of `metadata`.		
`spec` *VolumeSpec*			
`status` *VolumeStatus*			

VolumeAttachment

VolumeAttachment captures the intent to attach/detach the specified NV-Volume to/from the specified node. VolumeAttachment is an internal API, it is not intended to be used by users.

Appears in: - VolumeAttachmentList

Field	Description	Default	Validation
`apiVersion` *string*	`storage.dpu.nvidia.com/v1alpha1`		
`kind` *string*	`VolumeAttachment`		
`metadata` *ObjectMeta*	Refer to Kubernetes API documentation for fields of `metadata`.		
`spec` *VolumeAttachmentSpec*			
`status` *VolumeAttachmentStatus*			

VolumeAttachmentList

VolumeAttachmentList contains a list of VolumeAttachment

Field	Description	Default	Validation
`apiVersion` *string*	`storage.dpu.nvidia.com/v1alpha1`		

Field	Description	Default	Validation
`kind` *string*	`VolumeAttachmentList`		
`metadata` *ListMeta*	Refer to Kubernetes API documentation for fields of `metadata`.		
`items` *VolumeAttachment array*			

VolumeAttachmentSpec

VolumeAttachmentSpec defines the desired state of VolumeAttachment

Appears in: - VolumeAttachment

Field	Description	Default	Validation
`nodeName` *string*	The name of the node that the volume should be attached to		
`source` *VolumeSource*	Reference to the NV-Volume object		
`volumeAttachmentRef` *ObjectRef*	Reference to the SV-VolumeAttachment object		
`parameters` *object (keys:string, values:string)*	Opaque static publish properties of the volume returned by the plugin		
`functionType` *FunctionType*	FunctionType is the type of the emulated function that should be used to attach the volume		Enum: [pf vf]
`hotplugFunction` *boolean*	HotplugFunction is a boolean flag that indicates if the emulated function should be hotplugged		

VolumeAttachmentStatus

VolumeAttachmentStatus defines the observed state of VolumeAttachment

Appears in: - VolumeAttachment

Field	Description	Default	Validation
`storageAttached` *boolean*	Indicates the volume is successfully attached to the target storage system		
`message` *string*	The last error encountered during the attach operation, if any		
`dpu` *VolumeAttachmentStatusDPU*	Details about the DPU attachment		

VolumeAttachmentStatusDPU

VolumeAttachmentStatusDPU describe the information of DPU volume

Appears in: - VolumeAttachmentStatus

Field	Description	Default	Validation
`attached` *boolean*	Indicates the volume is successfully attached to the DPU node		
`pciDeviceAddress` *string*	PCI device address in the following format: (bus:device.function)		
`deviceName` *string*	The name of the device that was created by the storage vendor plugin		
`bdevAttrs` *BdevAttrs*	The attributes of the underlying block device		
`fsdevAttrs` *FSdevAttrs*	The attributes of the underlying filesystem device		

VolumeInfo

VolumeInfo represents a subset of fields from the PersistentVolume object that was created in the DPU cluster. This struct is used to track and expose key volume information without carrying the full PersistentVolume object.

Appears in: - DPUVolumeState

Field	Description	Default	Validation
`volumeName` *string*	VolumeName contains the name of the PersistentVolume object in the DPU cluster		
`capacity` *ResourceList*	Actual capacity of the volume in the DPU cluster		
`accessModes` *PersistentVolumeAccessMode array*	Actual access modes of the volume in the DPU cluster		
`volumeMode` *PersistentVolumeMode*	Actual volume mode of the volume in the DPU cluster		
`volumeAttributes` *object (keys:string, values:string)*	VolumeAttributes from the PersistentVolume object in the DPU cluster This field usually contains parameters returned by the Vendor CSI plugin on volume creation.		

VolumeList

VolumeList contains a list of Volume

Field	Description	Default	Validation
`apiVersion` *string*	`storage.dpu.nvidia.com/v1alpha1`		
`kind` *string*	`VolumeList`		
`metadata` *ListMeta*	Refer to Kubernetes API documentation for fields of `metadata`.		
`items` *Volume array*			

VolumeRequest

VolumeRequest represents the volume's requirements

Appears in: - VolumeSpec

Field	Description	Default	Validation
`capacityRange` *CapacityRange*	The capacity of the required storage space in bytes		
`accessModes` *PersistentVolumeAccessMode array*	Contains the types of access modes required		
`volumeMode` *PersistentVolumeMode*	volumeMode defines what type of volume is required by the claim. Value of Filesystem is implied when not included in claim spec.		

VolumeSource

VolumeSource references to the NV-Volume object

Appears in: - VolumeAttachmentSpec

Field	Description	Default	Validation
`volumeRef` *ObjectRef*	Reference to the NV-Volume object		

VolumeSpec

VolumeSpec defines the desired state of Volume

Appears in: - Volume

Field	Description	Default	Validation
`storageParameters` *object (keys:string, values:string)*	List of storage parameters supported by the policy, values are string only		
`request` *VolumeRequest*	The capacity of the required storage space in bytes		
`storagePolicyRef` *ObjectRef*	Reference to the StoragePolicy object		
`storagePolicyParameters` *object (keys:string, values:string)*	List of storage parameters supported by the policy, values are string only		
`volume` *VolumeSpecDPU*	Describe volume information in DPU cluster		

VolumeSpecDPU

VolumeSpecDPU describe volume information in DPU cluster

Appears in: - VolumeSpec

Field	Description	Default	Validation
`id` *string*			
`capacity` *Quantity*			
`accessModes` *PersistentVolumeAccessMode array*			
`reclaimPolicy` *PersistentVolumeReclaimPolicy*			Enum: [Delete Retain]
`storageVendorName` *string*			
`storageVendorPluginName` *string*			
`volumeAttributes` *object (keys:string, values:string)*			

Field	Description	Default	Validation
`csiReference` *CSIReference*			

VolumeState

Underlying type:string

VolumeState represents the state of volume

Validation: - Enum: [InProgress Available]

Appears in: - VolumeStatus

Field	Description
`InProgress`	InProgress means the some of related resource is still in progress
`Available`	Available means that all related resources are created

VolumeStatus

VolumeStatus defines the observed state of Volume

Appears in: - Volume

Field	Description	Default	Validation
`state` *VolumeState*	The state of a Volume object		Enum: [InProgress Available]

svc.dpu.nvidia.com/v1alpha1

Package v1alpha1 contains API Schema definitions for the svc.dpf v1alpha1 API group

Package v1alpha1 contains API Schema definitions for the sfc v1alpha1 API group

Resource Types

- DPUDeployment
- DPUDeploymentList
- DPUService
- DPUServiceChain
- DPUServiceChainList
- DPUServiceConfiguration
- DPUServiceConfigurationList
- DPUServiceCredentialRequest
- DPUServiceCredentialRequestList
- DPUServiceIPAM
- DPUServiceIPAMList
- DPUServiceInterface
- DPUServiceInterfaceList
- DPUServiceList
- DPUServiceNAD
- DPUServiceNADList
- DPUServiceTemplate
- DPUServiceTemplateList
- ServiceChain
- ServiceChainList
- ServiceChainSet
- ServiceChainSetList

- ServiceInterface
- ServiceInterfaceList
- ServiceInterfaceSet
- ServiceInterfaceSetList

ApplicationSource

ApplicationSource specifies the source of the Helm chart.

Appears in: - HelmChart

Field	Description	Default	Validation
`repoURL` *string*	RepoURL specifies the URL to the repository that contains the application Helm chart. The URL must begin with either 'oci://' or 'https://', ensuring it points to a validOCI registry or a web-based repository.		Pattern: `^(oci://\\|https://).+$`
`path` *string*	Path is the location of the chart inside the repo.		
`version` *string*	Version is a semver tag for the Chart's version.		MinLength: 1
`chart` *string*	Chart is the name of the helm chart.		
`releaseName` *string*	ReleaseName is the name to give to the release generate from the DPUService.		

CNIPlugin

CNIPlugin defines a CNI plugin to be used in a chained CNI configuration. When multiple CNI plugins are specified in ChainedCNIs, they are executed in order after the base OVS CNI plugin to provide additional network functionality.

Appears in: - DPUServiceNADSpec

Field	Description	Default	Validation
`type` *string*	Type specifies the CNI plugin type to be used in the chain. Currently only "rdma" is supported, which enables RDMA capabilities for the network interface.		Enum: [rdma]
`config` *RawExtension*	Config contains optional plugin-specific configuration as raw JSON. The configuration is merged into the CNI plugin configuration.		

ConfigPort

ConfigPort defines the configuration of a single port within a DPUService. Each port must have a unique name within the service.

Appears in: - ConfigPorts - DPUServiceStatus

Field	Description	Default	Validation
`name` *string*	Name is a unique identifier for the port within the DPUService. This name is used for reference inside the service.		MinLength: 1 Pattern: `^[a-z0-9-]+$`
`port` *integer*	Port is the port number that will be exposed by the service. Must be within the valid range of TCP/UDP ports (1-65535).		
`protocol` *Protocol*	Protocol specifies the transport protocol used by the port. Supported values: TCP, UDP		Enum: [TCP UDP]

Field	Description	Default	Validation
`nodePort` *integer*	NodePort is the external port assigned on each node in the cluster. If not set, Kubernetes will automatically allocate a NodePort.Constraints:- Can only be set when ServiceType is "NodePort".- Must be within the clusters valid NodePort range (Kubernetes default is 30000-32767).		

ConfigPorts

ConfigPorts defines the desired state of port configurations for a DPUService. This struct determines how ports are exposed from the DPU to the host cluster. A DPUService can only have a single ServiceType across all ports.

Validation: - If any port has a NodePort assigned, ServiceType **must** be "NodePort".

Appears in: - DPUServiceSpec - ServiceConfiguration

Field	Description	Default	Validation
`serviceType` *ServiceType*	ServiceType specifies the type of Kubernetes Service to create. All ports within this ConfigPorts will have the same ServiceType.The value is immutable and cannot be changed after creation.Supported values:- "NodePort": Exposes ports externally on a node.- "ClusterIP": Exposes ports internally within the cluster.- "None": Internal-only service with no cluster IP.Default: "NodePort"	NodePort	Enum: [NodePort ClusterIP None]
`ports` *ConfigPort array*	Ports defines the list of port configurations that will be exposed by the DPUService. Each port must specify a name, port number, and protocol.Constraints:- If ServiceType is "NodePort", ports may optionally specify a NodePort.- If ServiceType is "None" or "ClusterIP", ports **cannot** specify a NodePort.		

DPUDeployment

DPUDeployment is the Schema for the dpudeployments API. This object connects DPUServices with specific BFBs and DPUServiceChains.

Appears in: - DPUDeploymentList

Field	Description	Default	Validation
`apiVersion` *string*	`svc.dpu.nvidia.com/v1alpha1`		
`kind` *string*	`DPUDeployment`		
`metadata` *ObjectMeta*	Refer to Kubernetes API documentation for fields of `metadata`.		
`spec` *DPUDeploymentSpec*			
`status` *DPUDeploymentStatus*			

DPUDeploymentList

DPUDeploymentList contains a list of DPUDeployment

Field	Description	Default	Validation
`apiVersion` *string*	`svc.dpu.nvidia.com/v1alpha1`		
`kind` *string*	`DPUDeploymentList`		
`metadata` *ListMeta*	Refer to Kubernetes API documentation for fields of `metadata`.		
`items` *DPUDeployment array*			

DPUDeploymentPort

DPUDeploymentPort defines how a port can be configured

Appears in: - DPUDeploymentSwitch

Field	Description	Default	Validation
`service` *DPUDeploymentService*	Service holds configuration that helps configure the Service Function Chain and identify a port associated with a DPUService		
`serviceInterface` *ServiceIfc*	ServiceInterface holds configuration that helps configure the Service Function Chain and identify a user defined port		

DPUDeploymentService

DPUDeploymentService is the struct used for referencing an interface.

Appears in: - DPUDeploymentPort

Field	Description	Default	Validation
`name` *string*	Name is the name of the service as defined in the DPUDeployment Spec		MaxLength: 28 MinLength: 1
`interface` *string*	Interface name is the name of the interface as defined in the DPUServiceConfiguration		MaxLength: 28 MinLength: 1
`ipam` *IPAM*	IPAM defines the IPAM configuration that is configured in the Service Function Chain		

DPUDeploymentServiceConfiguration

DPUDeploymentServiceConfiguration describes the configuration of a particular Service

Appears in: - DPUDeploymentSpec

Field	Description	Default	Validation
`serviceTemplate` *string*	ServiceTemplate is the name of the DPUServiceTemplate object to be used for this Service. It must be in the same namespace as the DPUDeployment.		
`serviceConfiguration` *string*	ServiceConfiguration is the name of the DPUServiceConfiguration object to be used for this Service. It must be in the same namespace as the DPUDeployment.		
`dependsOn` *LocalObjectDependency array*	DependsOn is a list of local object dependencies that are required for this Service.		MinItems: 1

DPUDeploymentSpec

DPUDeploymentSpec defines the desired state of DPUDeployment

Appears in: - DPUDeployment

Field	Description	Default	Validat
`dpus` *DPUs*	DPUs contains the DPU related configuration		
`services` *object (keys:string, values:DPUDeploymentServiceConfiguration)*	Services contains the DPUDeploymentService related configuration. The key is the deploymentServiceName and the value is its configuration. All underlying objects must specify the same deploymentServiceName in order to be able to be consumed by theDPUDeployment.		MaxPro 50 MinPro 1

Field	Description	Default	Validat
`serviceChains` *ServiceChains*	ServiceChains contains the configuration related to the DPUServiceChains that the DPUDeployment creates.		
`revisionHistoryLimit` *integer*	The maximum number of revisions that can be retained during upgrades. Defaults to 10.	10	

DPUDeploymentStatus

DPUDeploymentStatus defines the observed state of DPUDeployment

Appears in: - DPUDeployment

Field	Description	Default	Validation
`conditions` *Condition array*	Conditions reflect the status of the object		
`observedGeneration` *integer*	ObservedGeneration records the Generation observed on the object the last time it was patched.		

DPUDeploymentSwitch

DPUDeploymentSwitch holds the ports that are connected in switch topology

Appears in: - ServiceChains

Field	Description	Default	Validation
`ports` *DPUDeploymentPort array*	Ports contains the ports of the switch		MaxItems: 50 MinItems: 1

Field	Description	Default	Validation
`serviceMTU` *integer*	ServiceMTU of the switch The default is 1500.	1500	Maximum: 9216 Minimum: 1280

DPUService

DPUService is the Schema for the dpuservices API

Appears in: - DPUServiceList

Field	Description	Default	Validation
`apiVersion` *string*	`svc.dpu.nvidia.com/v1alpha1`		
`kind` *string*	`DPUService`		
`metadata` *ObjectMeta*	Refer to Kubernetes API documentation for fields of `metadata`.		
`spec` *DPUServiceSpec*			
`status` *DPUServiceStatus*			

DPUServiceChain

DPUServiceChain is the Schema for the DPUServiceChain API

Appears in: - DPUServiceChainList

Field	Description	Default	Validation
`apiVersion` *string*	`svc.dpu.nvidia.com/v1alpha1`		
`kind` *string*	`DPUServiceChain`		
`metadata` *ObjectMeta*	Refer to Kubernetes API documentation for fields of `metadata`.		

Field	Description	Default	Validation
`spec` *DPUServiceChainSpec*			
`status` *DPUServiceChainStatus*			

DPUServiceChainList

DPUServiceChainList contains a list of DPUServiceChain

Field	Description	Default	Validation
`apiVersion` *string*	`svc.dpu.nvidia.com/v1alpha1`		
`kind` *string*	`DPUServiceChainList`		
`metadata` *ListMeta*	Refer to Kubernetes API documentation for fields of `metadata`.		
`items` *DPUServiceChain array*			

DPUServiceChainSpec

DPUServiceChainSpec defines the desired state of DPUServiceChainSpec

Appears in: - DPUServiceChain

Field	Description	Default	Validation
`clusterSelector` *LabelSelector*	Select the Clusters with specific labels, ServiceChainSet CRs will be created only for these Clusters		
`template` *ServiceChainSetSpecTemplate*	Template describes the ServiceChainSet that will be created for each selected Cluster.		

DPUServiceChainStatus

DPUServiceChainStatus defines the observed state of DPUServiceChain

Appears in: - DPUServiceChain

Field	Description	Default	Validation
`conditions` *Condition array*	Conditions reflect the status of the object		
`observedGeneration` *integer*	ObservedGeneration records the Generation observed on the object the last time it was patched.		

DPUServiceConfiguration

DPUServiceConfiguration is the Schema for the dpuserviceconfigurations API. This object is intended to be used in conjunction with a DPUDeployment object. This object is the template from which the DPUService will be created. It contains all configuration options from the user to be provided to the service itself via the helm chart values. This object doesn't allow configuration of nodeSelector and resources in purpose as these are delegated to the DPUDeployment and DPUServiceTemplate accordingly.

Appears in: - DPUServiceConfigurationList

Field	Description	Default	Validatio
`apiVersion` *string*	`svc.dpu.nvidia.com/v1alpha1`		
`kind` *string*	`DPUServiceConfiguration`		
`metadata` *ObjectMeta*	Refer to Kubernetes API documentation for fields of `metadata`.		
`spec` *DPUServiceConfigurationSpec*			
`status` *DPUServiceConfigurationStatus*			

DPUServiceConfigurationList

DPUServiceConfigurationList contains a list of DPUServiceConfiguration

Field	Description	Default	Validation
`apiVersion` *string*	`svc.dpu.nvidia.com/v1alpha1`		
`kind` *string*	`DPUServiceConfigurationList`		
`metadata` *ListMeta*	Refer to Kubernetes API documentation for fields of `metadata`.		
`items` *DPUServiceConfiguration array*			

DPUServiceConfigurationServiceDaemonSetValues

DPUServiceConfigurationServiceDaemonSetValues reflects the Helm related configuration

Appears in: - ServiceConfiguration

Field	Description	Default	Validation
`updateStrategy` *DaemonSetUpdateStrategy*	UpdateStrategy specifies the DeaemonSet update strategy for the ServiceDaemonset.		
`labels` *object (keys:string, values:string)*	Labels specifies labels which are added to the ServiceDaemonSet.		MaxProperties: 50
`annotations` *object (keys:string, values:string)*	Annotations specifies annotations which are added to the ServiceDaemonSet.		MaxProperties: 50
`resources` *ResourceList*	Resources specifies resources which are added to the ServiceDaemonSet.		

DPUServiceConfigurationSpec

DPUServiceConfigurationSpec defines the desired state of DPUServiceConfiguration

Appears in: - DPUServiceConfiguration

Field	Description	Default	Validation
`deploymentServiceName` *string*	DeploymentServiceName is the name of the DPU service this configuration refers to. It must match .spec.deploymentServiceName of a DPUServiceTemplate object and one of the keys in .spec.services of aDPUDeployment object.		MaxLength: 28 MinLength: 1
`serviceConfiguration` *ServiceConfiguration*	ServiceConfiguration contains fields that are configured on the generated DPUService.		
`interfaces` *ServiceInterfaceTemplate array*	Interfaces specifies the DPUServiceInterface to be generated for the generated DPUService.		MaxItems: 50 MinItems: 1
`upgradePolicy` *UpgradePolicy*	UpgradePolicy contains the configuration for the upgrade process	{ }	

DPUServiceConfigurationStatus

DPUServiceConfigurationStatus defines the observed state of DPUServiceConfiguration

Appears in: - DPUServiceConfiguration

DPUServiceCredentialRequest

DPUServiceCredentialRequest is the Schema for the dpuserviceCredentialRequests API

Appears in: - DPUServiceCredentialRequestList

Field	Description	Default	Vali
`apiVersion` *string*	`svc.dpu.nvidia.com/v1alpha1`		
`kind` *string*	`DPUServiceCredentialRequest`		
`metadata` *ObjectMeta*	Refer to Kubernetes API documentation for fields of `metadata`.		
`spec` *DPUServiceCredentialRequestSpec*			
`status` *DPUServiceCredentialRequestStatus*			

DPUServiceCredentialRequestList

DPUServiceCredentialRequestList contains a list of DPUServiceCredentialRequest

Field	Description	Default	Valida
`apiVersion` *string*	`svc.dpu.nvidia.com/v1alpha1`		
`kind` *string*	`DPUServiceCredentialRequestList`		
`metadata` *ListMeta*	Refer to Kubernetes API documentation for fields of `metadata`.		
`items` *DPUServiceCredentialRequest array*			

DPUServiceCredentialRequestSpec

DPUServiceCredentialRequestSpec defines the desired state of DPUServiceCredentialRequest

Appears in: - DPUServiceCredentialRequest

Field	Description	Default	Validation
`serviceAccount` *NamespacedName*	ServiceAccount defines the needed information to create the service account.		
`duration` *Duration*	Duration is the duration for which the token will be valid. Value must be in units accepted by Go time.ParseDuration https://golang.org/pkg/time/#ParseDuration.e.g. "1h", "1m", "1s", "1ms", "1.5h", "2h45m".Value duration must not be less than 10 minutes.**Note:** The maximum TTL for a token is 24 hours, after which the tokenwill be rotated.		Pattern: `^([0-9]-9]+)?(ms|s\` Type: strin
`targetCluster` *NamespacedName*	TargetCluster defines the target cluster where the service account will be created, and where a token for that service account will be requested.If not provided, the token will be requested for the same cluster wherethe DPUServiceCredentialRequest object is created.		
`type` *string*	Type is the type of the secret that will be created. The supported types are `kubeconfig` and `tokenFile`.If `kubeconfig` is selected, the secret will contain a kubeconfig file,that can be used to access the cluster.If `tokenFile` is selected, the secret will contain a token file and severalenvironment variables that can be used to access the cluster. It can be usedwith https://github.com/kubernetes/client-go/blob/v11.0.0/rest/config.go#L52to create a client that will handle file rotation.		Enum: [kul tokenFile]
`secret` *NamespacedName*	Secret defines the needed information to create the secret. The secret will be of the type specified in the `spec.type` field.		
`metadata` *ObjectMeta*	Refer to Kubernetes API documentation for fields of `metadata`.		

DPUServiceCredentialRequestStatus

DPUServiceCredentialRequestStatus defines the observed state of DPUServiceCredentialRequest

Appears in: - DPUServiceCredentialRequest

Field	Description	Default	Validation
`conditions` *Condition array*	Conditions defines current service state.		
`serviceAccount` *string*	ServiceAccount is the namespaced name of the ServiceAccount resource created by the controller for the DPUServiceCredentialRequest.		
`targetCluster` *string*	TargetCluster is the cluster where the service account was created. It has to be persisted in the status to be able to delete the service accountwhen the DPUServiceCredentialRequest is updated.		
`expirationTimestamp` *Time*	ExpirationTimestamp is the time when the token will expire.		
`issuedAt` *Time*	IssuedAt is the time when the token was issued.		
`secret` *string*	Sercet is the namespaced name of the Secret resource created by the controller for the DPUServiceCredentialRequest.		

DPUServiceIPAM

DPUServiceIPAM is the Schema for the dpuserviceipams API

Appears in: - DPUServiceIPAMList

Field	Description	Default	Validation
`apiVersion` *string*	`svc.dpu.nvidia.com/v1alpha1`		
`kind` *string*	`DPUServiceIPAM`		
`metadata` *ObjectMeta*	Refer to Kubernetes API documentation for fields of `metadata`.		
`spec` *DPUServiceIPAMSpec*			
`status` *DPUServiceIPAMStatus*			

DPUServiceIPAMList

DPUServiceIPAMList contains a list of DPUServiceIPAM

Field	Description	Default	Validation
`apiVersion` *string*	`svc.dpu.nvidia.com/v1alpha1`		
`kind` *string*	`DPUServiceIPAMList`		
`metadata` *ListMeta*	Refer to Kubernetes API documentation for fields of `metadata`.		
`items` *DPUServiceIPAM array*			

DPUServiceIPAMSpec

DPUServiceIPAMSpec defines the desired state of DPUServiceIPAM

Appears in: - DPUServiceIPAM

Field	Description	Default	Validation
`metadata` *ObjectMeta*	Refer to Kubernetes API documentation for fields of `metadata`.		
`ipv4Network` *IPV4Network*	IPV4Network is the configuration related to splitting a network into subnets per node, each with their own gateway.		
`ipv4Subnet` *IPV4Subnet*	IPV4Subnet is the configuration related to splitting a subnet into blocks per node. In this setup, there is a single gateway.		
`clusterSelector` *LabelSelector*	ClusterSelector determines in which clusters the DPUServiceIPAM controller should apply the configuration.		
`nodeSelector` *NodeSelector*	NodeSelector determines in which DPU nodes the DPUServiceIPAM controller should apply the configuration.		

DPUServiceIPAMStatus

DPUServiceIPAMStatus defines the observed state of DPUServiceIPAM

Appears in: - DPUServiceIPAM

Field	Description	Default	Validation
`conditions` *Condition array*	Conditions reflect the status of the object		
`observedGeneration` *integer*	ObservedGeneration records the Generation observed on the object the last time it was patched.		

DPUServiceInterface

DPUServiceInterface is the Schema for the DPUServiceInterface API

Appears in: - DPUServiceInterfaceList

Field	Description	Default	Validation
`apiVersion` *string*	`svc.dpu.nvidia.com/v1alpha1`		
`kind` *string*	`DPUServiceInterface`		
`metadata` *ObjectMeta*	Refer to Kubernetes API documentation for fields of `metadata`.		
`spec` *DPUServiceInterfaceSpec*			
`status` *DPUServiceInterfaceStatus*			

DPUServiceInterfaceList

DPUServiceInterfaceList contains a list of DPUServiceInterface

Field	Description	Default	Validation
`apiVersion` *string*	`svc.dpu.nvidia.com/v1alpha1`		
`kind` *string*	`DPUServiceInterfaceList`		
`metadata` *ListMeta*	Refer to Kubernetes API documentation for fields of `metadata`.		
`items` *DPUServiceInterface array*			

DPUServiceInterfaceSpec

DPUServiceInterfaceSpec defines the desired state of DPUServiceInterfaceSpec

Appears in: - DPUServiceInterface

Field	Description	Default	Validation
`clusterSelector` *LabelSelector*	Select the Clusters with specific labels, ServiceInterfaceSet CRs will be created only for these Clusters		
`template` *ServiceInterfaceSetSpecTemplate*	Template describes the ServiceInterfaceSet that will be created for each selected Cluster.		

DPUServiceInterfaceStatus

DPUServiceInterfaceStatus defines the observed state of DPUServiceInterface

Appears in: - DPUServiceInterface

Field	Description	Default	Validation
`conditions` *Condition array*	Conditions defines current service state.		
`observedGeneration` *integer*	ObservedGeneration records the Generation observed on the object the last time it was patched.		

DPUServiceList

DPUServiceList contains a list of DPUService

Field	Description	Default	Validation
`apiVersion` *string*	`svc.dpu.nvidia.com/v1alpha1`		
`kind` *string*	`DPUServiceList`		
`metadata` *ListMeta*	Refer to Kubernetes API documentation for fields of `metadata`.		

Field	Description	Default	Validation
`items` *DPUService array*			

DPUServiceNAD

DPUServiceNAD is the Schema for the dpuservicenads API.

Appears in: - DPUServiceNADList

Field	Description	Default	Validation
`apiVersion` *string*	`svc.dpu.nvidia.com/v1alpha1`		
`kind` *string*	`DPUServiceNAD`		
`metadata` *ObjectMeta*	Refer to Kubernetes API documentation for fields of `metadata`.		
`spec` *DPUServiceNADSpec*			
`status` *DPUServiceNADStatus*			

DPUServiceNADList

DPUServiceNADList contains a list of DPUServiceNAD.

Field	Description	Default	Validation
`apiVersion` *string*	`svc.dpu.nvidia.com/v1alpha1`		
`kind` *string*	`DPUServiceNADList`		
`metadata` *ListMeta*	Refer to Kubernetes API documentation for fields of `metadata`.		
`items` *DPUServiceNAD array*			

DPUServiceNADSpec

DPUServiceNADSpec defines the desired state of DPUServiceNAD.

Appears in: - DPUServiceNAD

Field	Description	Default	Validation
`metadata` *ObjectMeta*	Refer to Kubernetes API documentation for fields of `metadata`.		
`resourceType` *string*	ResourceType specifies the type of network resource to allocate for pods using this NAD. - "vf": Virtual Function (SR-IOV VF) from the DPU's physical ports- "sf": Scalable Function from the DPU (maps to nvidia.com/bf_sf or nvidia.com/bf_sf_trusted)- "veth": Virtual Ethernet pair (no device plugin resource required)The resource type determines which SR-IOV device plugin resource will be requested.		Enum: [vf sf veth]
`bridge` *string*	Bridge specifies the name of the OVS bridge to which the network interface will be connected. This bridge name is used in the CNI configuration for the OVS plugin.		
`serviceMTU` *integer*	ServiceMTU specifies the MTU size in bytes for the network interface. This value is passed to the OVS CNI plugin and determines the maximum packet size.If there is a DPUServiceChain that references an interface that is part of this network,then the MTU that is defined in the DPUServiceChain takes precedence.The default is 1500.	1500	Maximum: 9216 Minimum: 1280

Field	Description	Default	Validation
`ipam` *boolean*	IPAM enables IP Address Management for the network interfaces attached to this network When set to true, a DPUServiceChain that references the DPUServiceInterface that hasrequested this network must be created and include the relevant IPAM information. SeeDPUServiceChain documentation for more.When set to false, the network interfaces attached to this network will not get an IP		
`chainedCNIs` *CNIPlugin array*	ChainedCNIs specifies additional CNI plugins to be chained after the base OVS plugin. When specified, the NAD will use the CNI chaining format with the OVS plugin as thefirst plugin, followed by the plugins defined in this list.This allows adding capabilities like RDMA support on top of the base network interface.If empty, the NAD uses a single OVS plugin configuration (backward compatible format).		

DPUServiceNADStatus

DPUServiceNADStatus defines the observed state of DPUServiceNAD.

Appears in: - DPUServiceNAD

Field	Description	Default	Validation
`conditions` *Condition array*	Conditions reflect the status of the object		

DPUServiceSpec

DPUServiceSpec defines the desired state of DPUService

Appears in: - DPUService

Field	Description	Default	Validation
`helmChart` *HelmChart*	HelmChart reflects the Helm related configuration		
`serviceID` *string*	ServiceID is the ID of the service that the DPUService is associated with.		
`serviceDaemonSet` *ServiceDaemonSetValues*	ServiceDaemonSet specifies the configuration for the ServiceDaemonSet.		
`deployInCluster` *boolean*	DeployInCluster indicates if the DPUService Helm Chart will be deployed on the Host cluster. Default to false.		
`interfaces` *string array*	Interfaces specifies the DPUServiceInterface names that the DPUService uses in the same namespace.		MaxItems: 50 MinItems: 1
`paused` *boolean*	Paused indicates that the DPUService is paused. Underlying resources are also paused when this is set to true.No deletion of resources will occur when this is set to true.		
`configPorts` *ConfigPorts*	ConfigPorts defines the desired state of port configurations for a DPUService. This struct determines how ports are exposed from the DPU to the host cluster.A DPUService can only have a single ServiceType across all ports.		

DPUServiceStatus

DPUServiceStatus defines the observed state of DPUService

Appears in: - DPUService

Field	Description	Default	Validation
`conditions` *Condition array*	Conditions defines current service state.		
`observedGeneration` *integer*	ObservedGeneration records the Generation observed on the object the last time it was patched.		
`configPorts` *object (keys:string, values:ConfigPort)*	ConfigPorts defines the observed state of the config ports. It contains the actual port numbers that are exposed on the DPUService per cluster.		
`serviceID` *string*	ServiceID is the ID of the service that the DPUService is associated with. This is set when the DPUService is created.		

DPUServiceTemplate

DPUServiceTemplate is the Schema for the DPUServiceTemplate API. This object is intended to be used in conjunction with a DPUDeployment object. This object is the template from which the DPUService will be created. It contains configuration options related to resources required by the service to be deployed. The rest of the configuration options must be defined in a DPUServiceConfiguration object.

Appears in: - DPUServiceTemplateList

Field	Description	Default	Validation
`apiVersion` *string*	`svc.dpu.nvidia.com/v1alpha1`		
`kind` *string*	`DPUServiceTemplate`		
`metadata` *ObjectMeta*	Refer to Kubernetes API documentation for fields of `metadata`.		
`spec` *DPUServiceTemplateSpec*			

Field	Description	Default	Validation
`status` *DPUServiceTemplateStatus*			

DPUServiceTemplateList

DPUServiceTemplateList contains a list of DPUServiceTemplate

Field	Description	Default	Validation
`apiVersion` *string*	`svc.dpu.nvidia.com/v1alpha1`		
`kind` *string*	`DPUServiceTemplateList`		
`metadata` *ListMeta*	Refer to Kubernetes API documentation for fields of `metadata`.		
`items` *DPUServiceTemplate array*			

DPUServiceTemplateSpec

DPUServiceTemplateSpec defines the desired state of DPUServiceTemplate

Appears in: - DPUServiceTemplate

Field	Description	Default	Validation
`deploymentServiceName` *string*	DeploymentServiceName is the name of the DPU service this configuration refers to. It must match .spec.deploymentServiceName of a DPUServiceConfiguration object and one of the keys in .spec.services of aDPUDeployment object.		MaxLength: 28 MinLength: 1

Field	Description	Default	Validation
`helmChart` *HelmChart*	HelmChart reflects the Helm related configuration. The user is supposed to configure the values that are static across any DPUServiceConfiguration used with this DPUServiceTemplate in a DPUDeployment. These values act as abaseline and are merged with values specified in the DPUServiceConfiguration. In case of conflict, theDPUServiceConfiguration values take precedence.		
`resourceRequirements` *ResourceList*	ResourceRequirements contains the overall resources required by this particular service to run on a single node		

DPUServiceTemplateStatus

DPUServiceTemplateStatus defines the observed state of DPUServiceTemplate

Appears in: - DPUServiceTemplate

Field	Description	Default	Validation
`conditions` *Condition array*	Conditions reflect the status of the object		
`observedGeneration` *integer*	ObservedGeneration records the Generation observed on the object the last time it was patched.		
`versions` *object (keys:string, values:string)*	Versions reflects the required versions the generated DPUService needs in order to function correctly.		

DPUSet

DPUSet contains configuration for the DPUSet to be created by the DPUDeployment

Appears in: - DPUs

Field	Description	Default	Validation
`nameSuffix` *string*	NameSuffix is the suffix to be added to the name of the DPUSet object created by the DPUDeployment.		MaxLength: 24 MinLength: 1
`nodeSelector` *LabelSelector*	NodeSelector defines the nodes that the DPUSet should target		
`dpuSelector` *object (keys:string, values:string)*	DPUSelector defines the DPUs that the DPUSet should target		
`dpuAnnotations` *object (keys:string, values:string)*	DPUAnnotations is the annotations to be added to the DPU object created by the DPUSet.		MaxProperties: 50

DPUs

DPUs contains the DPU related configuration

Appears in: - DPUDeploymentSpec

Field	Description	Default	Validation
`bfb` *string*	BFB is the name of the BFB object to be used in this DPUDeployment. It must be in the same namespace as the DPUDeployment.		
`flavor` *string*	Flavor is the name of the DPUFlavor object to be used in this DPUDeployment. It must be in the same namespace as the DPUDeployment.		
`dpuSets` *DPUSet array*	DPUSets contains configuration for each DPUSet that is going to be created by the DPUDeployment		MaxItems: 50 MinItems: 1

Field	Description	Default	Validation
`nodeEffect` *Action*	NodeEffect is the effect the DPU has on Nodes during provisioning.		

HelmChart

HelmChart reflects the helm related configuration

Appears in: - DPUServiceSpec - DPUServiceTemplateSpec

Field	Description	Default	Validation
`source` *ApplicationSource*	Source specifies information about the Helm chart		
`values` *RawExtension*	Values specifies Helm values to be passed to Helm template, defined as a map. This takes precedence over Values.		

IPV4Network

IPV4Network describes the configuration relevant to splitting a network into subnet per node (i.e. different gateway and broadcast IP per node).

Appears in: - DPUServiceIPAMSpec

Field	Description	Default	Validation
`network` *string*	Network is the CIDR from which subnets should be allocated per node.		
`gatewayIndex` *integer*	GatewayIndex determines which IP in the subnet extracted from the CIDR should be the gateway IP. For point to point networks (/31), one needs to leave this empty to make use of both the IPs.		
`prefixSize` *integer*	PrefixSize is the size of the subnet that should be allocated per node.		

Field	Description	Default	Validation
`exclusions` *string array*	Exclusions is a list of IPs that should be excluded when splitting the CIDR into subnets per node.		
`allocations` *object (keys:string, values:string)*	Allocations describes the subnets that should be assigned in each DPU node.		
`defaultGateway` *boolean*	DefaultGateway adds gateway as default gateway in the routes list if true.		
`routes` *Route array*	Routes is the static routes list using the gateway specified in the spec.		

IPV4Subnet

IPV4Subnet describes the configuration relevant to splitting a subnet to a subnet block per node (i.e. same gateway and broadcast IP across all nodes).

Appears in: - DPUServiceIPAMSpec

Field	Description	Default	Validation
`subnet` *string*	Subnet is the CIDR from which blocks should be allocated per node		
`gateway` *string*	Gateway is the IP in the subnet that should be the gateway of the subnet.		
`perNodeIPCount` *integer*	PerNodeIPCount is the number of IPs that should be allocated per node.		
`defaultGateway` *boolean*	if true, add gateway as default gateway in the routes list DefaultGateway adds gateway as default gateway in the routes list if true.		
`routes` *Route array*	Routes is the static routes list using the gateway specified in the spec.		

LocalObjectDependency

LocalObjectDependency is a list of local object dependencies that are required for this Service. The object must be part of the dpuDeployment `spec.services` list.

Appears in: - DPUDeploymentServiceConfiguration

Field	Description	Default	Validation
`name` *string*	Name is the name of the object		

NamespacedName

NamespacedName contains enough information to locate the referenced Kubernetes resource object in any namespace.

Appears in: - DPUServiceCredentialRequestSpec

Field	Description	Default	Validation
`name` *string*	Name of the object.		
`namespace` *string*	Namespace of the object, if not provided the object will be looked up in the same namespace as the referring object		

OVN

OVN defines the configuration for OVN interface type

Appears in: - ServiceInterfaceSpec

Field	Description	Default	Validation
`externalBridge` *string*	ExternalBridge is the name of the OVN bridge	br-ovn	

ObjectMeta

ObjectMeta holds metadata like labels and annotations.

Appears in: - DPUServiceCredentialRequestSpec - DPUServiceIPAMSpec - DPUServiceNADSpec - ServiceChainSetSpecTemplate - ServiceChainSpecTemplate - ServiceInterfaceSetSpecTemplate - ServiceInterfaceSpecTemplate

Field	Description	Default	Validation
`labels` *object (keys:string, values:string)*	Labels is a map of string keys and values.		
`annotations` *object (keys:string, values:string)*	Annotations is a map of string keys and values.		

PF

PF defines the PF configuration

Appears in: - ServiceInterfaceSpec

Field	Description	Default	Validation
`pfID` *integer*	The PF ID		
`virtualNetwork` *string*	VirtualNetwork is the VirtualNetwork name in the same namespace		

Physical

Physical Identifies a physical interface

Appears in: - ServiceInterfaceSpec

Field	Description	Default	Validation
`interfaceName` *string*	The interface name		

Port

Port defines the port configuration

Appears in: - Switch

Field	Description	Default	Validation
`serviceInterface` *ServiceIfc*			

Route

Route contains static route parameters

Appears in: - IPV4Network - IPV4Subnet

Field	Description	Default	Validation
`dst` *string*	The destination of the route, in CIDR notation		

ServiceChain

ServiceChain is the Schema for the servicechains API

Appears in: - ServiceChainList

Field	Description	Default	Validation
`apiVersion` *string*	`svc.dpu.nvidia.com/v1alpha1`		
`kind` *string*	`ServiceChain`		
`metadata` *ObjectMeta*	Refer to Kubernetes API documentation for fields of `metadata`.		
`spec` *ServiceChainSpec*			
`status` *ServiceChainStatus*			

ServiceChainList

ServiceChainList contains a list of ServiceChain

Field	Description	Default	Validation
`apiVersion` *string*	`svc.dpu.nvidia.com/v1alpha1`		
`kind` *string*	`ServiceChainList`		
`metadata` *ListMeta*	Refer to Kubernetes API documentation for fields of `metadata`.		
`items` *ServiceChain array*			

ServiceChainSet

ServiceChainSet is the Schema for the servicechainsets API

Appears in: - ServiceChainSetList

Field	Description	Default	Validation
`apiVersion` *string*	`svc.dpu.nvidia.com/v1alpha1`		
`kind` *string*	`ServiceChainSet`		
`metadata` *ObjectMeta*	Refer to Kubernetes API documentation for fields of `metadata`.		
`spec` *ServiceChainSetSpec*			
`status` *ServiceChainSetStatus*			

ServiceChainSetList

ServiceChainSetList contains a list of ServiceChainSet

Field	Description	Default	Validation
`apiVersion` *string*	`svc.dpu.nvidia.com/v1alpha1`		
`kind` *string*	`ServiceChainSetList`		
`metadata` *ListMeta*	Refer to Kubernetes API documentation for fields of `metadata`.		
`items` *ServiceChainSet array*			

ServiceChainSetSpec

ServiceChainSetSpec defines the desired state of ServiceChainSet

Appears in: - ServiceChainSet - ServiceChainSetSpecTemplate

Field	Description	Default	Validation
`nodeSelector` *LabelSelector*	Select the Nodes with specific labels, ServiceChain CRs will be created only for these Nodes		
`template` *ServiceChainSpecTemplate*	ServiceChainSpecTemplate holds the template for the ServiceChainSpec		

ServiceChainSetSpecTemplate

ServiceChainSetSpecTemplate describes the data a ServiceChainSet should have when created from a template.

Appears in: - DPUServiceChainSpec

Field	Description	Default	Validation
`spec` *ServiceChainSetSpec*			

Field	Description	Default	Validation
`metadata` *ObjectMeta*	Refer to Kubernetes API documentation for fields of `metadata`.		

ServiceChainSetStatus

ServiceChainSetStatus defines the observed state of ServiceChainSet

Appears in: - ServiceChainSet

Field	Description	Default	Validation
`conditions` *Condition array*	Conditions reflect the status of the object		
`observedGeneration` *integer*	ObservedGeneration records the Generation observed on the object the last time it was patched.		
`numberApplied` *integer*	The number of nodes where the service chain is applied and is supposed to be applied.		
`numberReady` *integer*	The number of nodes where the service chain is applied and ready.		

ServiceChainSpec

ServiceChainSpec defines the desired state of ServiceChain

Appears in: - ServiceChain - ServiceChainSpecTemplate

Field	Description	Default	Validation
`node` *string*	Node where this ServiceChain applies to		
`switches` *Switch array*	The switches of the ServiceChain, order is significant		MaxItems: 50 MinItems: 1

ServiceChainSpecTemplate

ServiceChainSpecTemplate defines the template from which ServiceChainSpecs are created

Appears in: - ServiceChainSetSpec

Field	Description	Default	Validation
`spec` *ServiceChainSpec*	ServiceChainSpec is the spec for the ServiceChainSpec		
`metadata` *ObjectMeta*	Refer to Kubernetes API documentation for fields of `metadata`.		

ServiceChainStatus

ServiceChainStatus defines the observed state of ServiceChain

Appears in: - ServiceChain

Field	Description	Default	Validation
`conditions` *Condition array*	Conditions reflect the status of the object		
`observedGeneration` *integer*	ObservedGeneration records the Generation observed on the object the last time it was patched.		

ServiceChains

Appears in: - DPUDeploymentSpec

Field	Description	Default	Validation
`upgradePolicy` *UpgradePolicy*	UpgradePolicy contains the configuration for the upgrade process	{ }	

Field	Description	Default	Validation
`switches` *DPUDeploymentSwitch array*	Switches is the list of switches that form the service chain		MaxItems: 50 MinItems: 1

ServiceConfiguration

ServiceConfiguration contains fields that are configured on the generated DPUService.

Appears in: - DPUServiceConfigurationSpec

Field	Description	Default
`helmChart` *ServiceConfigurationHelmChart*	HelmChart reflects the Helm related configuration. The user is supposed to configure values specific to that DPUServiceConfiguration used in a DPUDeployment and should not specify values that could be shared across multipleDPUDeployments using different DPUServiceConfigurations. These values are merged with values specified in theDPUServiceTemplate. In case of conflict, the DPUServiceConfiguration values take precedence.	
`serviceDaemonSet` *DPUServiceConfigurationServiceDaemonSetValues*	ServiceDaemonSet contains settings related to the underlying DaemonSet that is part of the Helm chart	

Field	Description	Default
`deployInCluster` *boolean*	DeployInCluster indicates if the DPUService Helm Chart will be deployed on the Host cluster. Default to false.	
`configPorts` *ConfigPorts*	ConfigPorts defines the desired state of port configurations for a DPUService. This struct determines how ports are exposed from the DPU to the host cluster.A DPUService can only have a single ServiceType across all ports.	

ServiceConfigurationHelmChart

ServiceConfigurationHelmChart reflects the helm related configuration

Appears in: - ServiceConfiguration

Field	Description	Default	Validation
`values` *RawExtension*	Values specifies Helm values to be passed to Helm template, defined as a map. This takes precedence over Values.		

ServiceDaemonSetValues

ServiceDaemonSetValues specifies the configuration for the ServiceDaemonSet.

Appears in: - DPUServiceSpec

Field	Description	Default	Validation
`nodeSelector` *NodeSelector*	NodeSelector specifies which Nodes to deploy the ServiceDaemonSet to.		
`updateStrategy` *DaemonSetUpdateStrategy*	UpdateStrategy specifies the DeaemonSet update strategy for the ServiceDaemonset.		
`labels` *object (keys:string, values:string)*	Labels specifies labels which are added to the ServiceDaemonSet.		
`annotations` *object (keys:string, values:string)*	Annotations specifies annotations which are added to the ServiceDaemonSet.		
`resources` *ResourceList*	Resources specifies resources which are added to the ServiceDaemonSet.		

ServiceDef

ServiceDef Identifies the service and network for the ServiceInterface

Appears in: - ServiceInterfaceSpec

Field	Description	Default	Validation
`serviceID` *string*	ServiceID is the DPU Service Identifier		
`network` *string*	Network is the Network Attachment Definition in the form of "namespace/name" or just "name" if the namespace is the same as the ServiceInterface.		
`interfaceName` *string*	The interface name		
`virtualNetwork` *string*	VirtualNetwork is the VirtualNetwork name in the same namespace		

ServiceIfc

ServiceIfc defines the service interface configuration

Appears in: - DPUDeploymentPort - Port

Field	Description	Default	Validation
`matchLabels` *object (keys:string, values:string)*	Labels matching service interface		MaxProperties: 50 MinProperties: 1
`ipam` *IPAM*	IPAM defines the IPAM configuration when referencing a serviceInterface of type 'service'		

ServiceInterface

ServiceInterface is the Schema for the serviceinterfaces API

Appears in: - ServiceInterfaceList

Field	Description	Default	Validation
`apiVersion` *string*	`svc.dpu.nvidia.com/v1alpha1`		
`kind` *string*	`ServiceInterface`		
`metadata` *ObjectMeta*	Refer to Kubernetes API documentation for fields of `metadata`.		
`spec` *ServiceInterfaceSpec*			
`status` *ServiceInterfaceStatus*			

ServiceInterfaceList

ServiceInterfaceList contains a list of ServiceInterface

Field	Description	Default	Validation
`apiVersion` *string*	`svc.dpu.nvidia.com/v1alpha1`		
`kind` *string*	`ServiceInterfaceList`		
`metadata` *ListMeta*	Refer to Kubernetes API documentation for fields of `metadata`.		
`items` *ServiceInterface array*			

ServiceInterfaceSet

ServiceInterfaceSet is the Schema for the serviceinterfacesets API

Appears in: - ServiceInterfaceSetList

Field	Description	Default	Validation
`apiVersion` *string*	`svc.dpu.nvidia.com/v1alpha1`		
`kind` *string*	`ServiceInterfaceSet`		
`metadata` *ObjectMeta*	Refer to Kubernetes API documentation for fields of `metadata`.		
`spec` *ServiceInterfaceSetSpec*			
`status` *ServiceInterfaceSetStatus*			

ServiceInterfaceSetList

ServiceInterfaceSetList contains a list of ServiceInterfaceSet

Field	Description	Default	Validation
`apiVersion` *string*	`svc.dpu.nvidia.com/v1alpha1`		
`kind` *string*	`ServiceInterfaceSetList`		
`metadata` *ListMeta*	Refer to Kubernetes API documentation for fields of `metadata`.		
`items` *ServiceInterfaceSet array*			

ServiceInterfaceSetSpec

ServiceInterfaceSetSpec defines the desired state of ServiceInterfaceSet

Appears in: - ServiceInterfaceSet - ServiceInterfaceSetSpecTemplate

Field	Description	Default	Validation
`nodeSelector` *LabelSelector*	Select the Nodes with specific labels, ServiceInterface CRs will be created only for these Nodes		
`template` *ServiceInterfaceSpecTemplate*	Template holds the template for the serviceInterfaceSpec		

ServiceInterfaceSetSpecTemplate

ServiceInterfaceSetSpecTemplate describes the data a ServiceInterfaceSet should have when created from a template.

Appears in: - DPUServiceInterfaceSpec

Field	Description	Default	Validation
`spec` *ServiceInterfaceSetSpec*			

Field	Description	Default	Validation
`metadata` *ObjectMeta*	Refer to Kubernetes API documentation for fields of `metadata`.		

ServiceInterfaceSetStatus

ServiceInterfaceSetStatus defines the observed state of ServiceInterfaceSet

Appears in: - ServiceInterfaceSet

Field	Description	Default	Validation
`conditions` *Condition array*	Conditions reflect the status of the object		
`observedGeneration` *integer*	ObservedGeneration records the Generation observed on the object the last time it was patched.		
`numberApplied` *integer*	The number of nodes where the service chain is applied and is supposed to be applied.		
`numberReady` *integer*	The number of nodes where the service chain is applied and ready.		

ServiceInterfaceSpec

ServiceInterfaceSpec defines the desired state of ServiceInterface

Appears in: - ServiceInterface - ServiceInterfaceSpecTemplate

Field	Description	Default	Validation
`node` *string*	Node where this interface exists		
`interfaceType` *string*	The interface type ("vlan", "physical", "pf", "vf", "ovn", "service")		Enum: [vlan physical pf vf ovn service]

Field	Description	Default	Validation
`physical` *Physical*	The physical interface definition		
`vlan` *VLAN*	The VLAN definition		
`vf` *VF*	The VF definition		
`pf` *PF*	The PF definition		
`service` *ServiceDef*	The Service definition		
`ovn` *OVN*	The OVN definition		

ServiceInterfaceSpecTemplate

ServiceInterfaceSpecTemplate defines the template from which ServiceInterfaceSpecs are created

Appears in: - ServiceInterfaceSetSpec

Field	Description	Default	Validation
`spec` *ServiceInterfaceSpec*	ServiceInterfaceSpec is the spec for the ServiceInterfaceSpec		
`metadata` *ObjectMeta*	Refer to Kubernetes API documentation for fields of `metadata`.		

ServiceInterfaceStatus

ServiceInterfaceStatus defines the observed state of ServiceInterface

Appears in: - ServiceInterface

Field	Description	Default	Validation
`conditions` *Condition array*	Conditions reflect the status of the object		

Field	Description	Default	Validation
`observedGeneration` *integer*	ObservedGeneration records the Generation observed on the object the last time it was patched.		

ServiceInterfaceTemplate

ServiceInterfaceTemplate contains the information related to an interface of the DPUService

Appears in: - DPUServiceConfigurationSpec

Field	Description	Default	Validation
`name` *string*	Name is the name of the interface		MaxLength: 28 MinLength: 1
`network` *string*	Network is the Network Attachment Definition in the form of "namespace/name" or just "name" if the namespace is the same as the namespace the pod is running.		
`virtualNetwork` *string*	VirtualNetwork is the VirtualNetwork name in the same namespace		

Switch

Switch defines the switch configuration

Appears in: - ServiceChainSpec

Field	Description	Default	Validation
`ports` *Port array*	Ports of the switch		MaxItems: 50 MinItems: 1

Field	Description	Default	Validation
`serviceMTU` *integer*	ServiceMTU of the switch The default is 1500.	1500	Maximum: 9216 Minimum: 1280

UpgradePolicy

Appears in: - DPUServiceConfigurationSpec - ServiceChains

Field	Description	Default	Validation
`applyNodeEffect` *boolean*	ApplyNodeEffect specifies if the node effect should be applied during the upgrade. It signals the reconciler that this object upgrade is disruptive.Hence a new revision of the object should be created and node effect shouldbe applied.	true	

VF

VF defines the VF configuration

Appears in: - ServiceInterfaceSpec

Field	Description	Default	Validation
`vfID` *integer*	The VF ID		
`pfID` *integer*	The PF ID		
`parentInterfaceRef` *string*	The parent interface reference TODO: Figure out what this field is supposed to be		
`virtualNetwork` *string*	VirtualNetwork is the VirtualNetwork name in the same namespace		

VLAN

VLAN defines the VLAN configuration

Appears in: - ServiceInterfaceSpec

Field	Description	Default	Validation
`vlanID` *integer*	The VLAN ID		
`parentInterfaceRef` *string*	The parent interface reference TODO: Figure out what this field is supposed to be		

vpc.dpu.nvidia.com/v1alpha1

Package v1alpha1 contains API Schema definitions for the storage v1alpha1 API group

Resource Types

- DPUVPC
- DPUVPCList
- DPUVirtualNetwork
- DPUVirtualNetworkList
- IsolationClass
- IsolationClassList

BridgedNetworkIPAMIPv4Spec

BridgedNetworkIPAMIPv4Spec contains IPv4 IPAM configuration for bridged network

Appears in: - BridgedNetworkIPAMSpec

Field	Description	Default	Validation
`dhcp` *boolean*	DHCP if set, enables DHCP for the network		

Field	Description	Default	Validation
`subnet` *string*	Subnet is the network subnet in CIDR format to use for DHCP. the first IP in the subnet is the gateway.		
`excludeIPs` *ExcludeIPsEntry array*	ExcludeIPs are the IPs to exclude from DHCP allocation.		

BridgedNetworkIPAMSpec

BridgedNetworkIPAMSpec contains IPAM configuration for bridged network

Appears in: - BridgedNetworkSpec

Field	Description	Default	Validation
`ipv4` *BridgedNetworkIPAMIPv4Spec*	IPv4 contains the IPv4 IPAM configuration		

BridgedNetworkSpec

BridgedNetworkSpec contains configuration for bridged network

Appears in: - DPUVirtualNetworkSpec

Field	Description	Default	Validation
`ipam` *BridgedNetworkIPAMSpec*	IPAM contains the IPAM configuration for the bridged network		

DPUVPC

DPUVPC is the Schema for the dpuvpc API

Appears in: - DPUVPCList

Field	Description	Default	Validation
`apiVersion` *string*	`vpc.dpu.nvidia.com/v1alpha1`		
`kind` *string*	`DPUVPC`		
`metadata` *ObjectMeta*	Refer to Kubernetes API documentation for fields of `metadata`.		
`spec` *DPUVPCSpec*			
`status` *DPUVPCStatus*			

DPUVPCList

DPUVPCList contains a list of DPUVPC

Field	Description	Default	Validation
`apiVersion` *string*	`vpc.dpu.nvidia.com/v1alpha1`		
`kind` *string*	`DPUVPCList`		
`metadata` *ListMeta*	Refer to Kubernetes API documentation for fields of `metadata`.		
`items` *DPUVPC array*			

DPUVPCSpec

DPUVPCSpec defines the desired state of DPUVPCSpec

Appears in: - DPUVPC

Field	Description	Default	Validation
`tenant` *string*	Tenant which owns the VPC.		MinLength: 1
`nodeSelector` *LabelSelector*	NodeSelector Selects the DPU Nodes with specific labels which belong to this VPC.		
`isolationClassName` *string*	IsolationClassName is the name of the isolation class to use for the VPC		MinLength: 1
`interNetworkAccess` *boolean*	InterNetworkAccess defines if virtual networks within the VPC are routed or not. if set to false, communication between virtual networks is not allowed.		

DPUVPCStatus

DPUVPCStatus defines the observed state of DPUVPC

Appears in: - DPUVPC

Field	Description	Default	Validation
`virtualNetworks` *VirtualNetworkStatus array*	VirtualNetworks contains the virtual networks that belong to this VPC		
`conditions` *Condition array*	Conditions reflect the status of the object		
`observedGeneration` *integer*	ObservedGeneration records the Generation observed on the object the last time it was patched.		

DPUVirtualNetwork

DPUVirtualNetwork is the Schema for the dpuvirtualnetwork API

Appears in: - DPUVirtualNetworkList

Field	Description	Default	Validation
`apiVersion` *string*	`vpc.dpu.nvidia.com/v1alpha1`		
`kind` *string*	`DPUVirtualNetwork`		
`metadata` *ObjectMeta*	Refer to Kubernetes API documentation for fields of `metadata`.		
`spec` *DPUVirtualNetworkSpec*			
`status` *DPUVirtualNetworkStatus*			

DPUVirtualNetworkList

DPUVirtualNetworkList contains a list of DPUVirtualNetwork

Field	Description	Default	Validation
`apiVersion` *string*	`vpc.dpu.nvidia.com/v1alpha1`		
`kind` *string*	`DPUVirtualNetworkList`		
`metadata` *ListMeta*	Refer to Kubernetes API documentation for fields of `metadata`.		
`items` *DPUVirtualNetwork array*			

DPUVirtualNetworkSpec

DPUVirtualNetworkSpec defines the desired state of DPUVirtualNetworkSpec

Appears in: - DPUVirtualNetwork

Field	Description	Default	Validation
`nodeSelector` *LabelSelector*	NodeSelector Selects the DPU Nodes with specific labels which can belong to the virtual network.		
`vpcName` *string*	vpcName is the name of the DPUVPC the virtual network belongs within the same namespace.		
`type` *NetworkType*	Type of the virtual network		Enum: [Bridged]
`externallyRouted` *boolean*	ExternallyRouted defines if the virtual network can be routed externally		
`masquerade` *boolean*	Masquerade defines if the virtual network should masquerade the traffic before egressing to external networks. valid only if ExternallyRouted is true	true	
`bridgedNetwork` *BridgedNetworkSpec*	BridgedNetwork contains the bridged network configuration		

DPUVirtualNetworkStatus

DPUVirtualNetworkStatus defines the observed state of DPUVirtualNetwork

Appears in: - DPUVirtualNetwork

Field	Description	Default	Validation
`conditions` *Condition array*	Conditions reflect the status of the object		
`observedGeneration` *integer*	ObservedGeneration records the Generation observed on the object the last time it was patched.		

ExcludeIPsEntry

Appears in: - BridgedNetworkIPAMIPv4Spec

Field	Description	Default	Validation
`ip` *string*	IP is the IP address to exclude from DHCP allocation. must be part for the virtual network subnet.		
`range` *RangeEntry*	Range is the range of IP addresses to exclude from DHCP allocation. must be part for the virtual network subnet.		

IsolationClass

IsolationClass is the Schema for the isolationclass API

Appears in: - IsolationClassList

Field	Description	Default	Validation
`apiVersion` *string*	`vpc.dpu.nvidia.com/v1alpha1`		
`kind` *string*	`IsolationClass`		
`metadata` *ObjectMeta*	Refer to Kubernetes API documentation for fields of `metadata`.		
`spec` *IsolationClassSpec*			
`status` *IsolationClassStatus*			

IsolationClassList

IsolationClassList contains a list of IsolationClass

Field	Description	Default	Validation
`apiVersion` *string*	`vpc.dpu.nvidia.com/v1alpha1`		

Field	Description	Default	Validation
`kind` *string*	`IsolationClassList`		
`metadata` *ListMeta*	Refer to Kubernetes API documentation for fields of `metadata`.		
`items` *IsolationClass array*			

IsolationClassSpec

IsolationClassSpec defines the configuration of IsolationClass

Appears in: - IsolationClass

Field	Description	Default	Validation
`provisioner` *string*	Provisioner indicates the type of the provisioner.		
`parameters` *object (keys:string, values:string)*	Parameters holds the parameters for the provisioner		

IsolationClassStatus

IsolationClassStatus defines the status of IsolationClass

Appears in: - IsolationClass

NetworkType

Underlying type:string

NetworkType represents the type of the virtual network

Validation: - Enum: [Bridged]

Appears in: - DPUVirtualNetworkSpec

Field	Description
`Bridged`	BridgedVirtualNetworkType represents a bridged virtual network

RangeEntry

Underlying type:struct{Start string `"json:\"start\""`; End string `"json:\"end\""`}

RangeEntry contains a range of IP addresses

Appears in: - ExcludeIPsEntry

VirtualNetworkStatus

VirtualNetworkStatus is the status of a virtual network

Appears in: - DPUVPCStatus

Field	Description	Default	Validation
`name` *string*	the name of the virtual network		

DPFOperatorConfig

Overview

The `DPFOperatorConfig` controls how DPF operates in your Kubernetes cluster. This guide explains the major configuration options. When the config is applied, the DPF Operator will deploy all necessary components and configure them according to the configuration.

Basic Configuration Example

This basic config example enables the Kamaji cluster manager and sets the PVC name of the BFB PVC which is necessary for the provisioning controller to download the bf-bundle.

> **Note**
>
> In the current implementation the `DPFOperatorConfig` resource is a singleton. This means that only one instance of this resource can exist in the cluster. If you try to create a second instance, the controllers will not work as expected.

You can find the full API documentation in the API Reference.

```
apiVersion: operator.doca-platform.nvidia.com/v1alpha1
kind: DPFOperatorConfig
metadata:
  name: dpf-operator-config
spec:
  staticClusterManager:
    disable: true
  kamajiClusterManager:
    disable: false
  provisioningController:
    bfbPVCName: bfb-pvc
```

We can verify if the configuration is applied correctly by checking the status of the `DPFOperatorConfig` resource.

```
$ kubectl -n dpf-operator-system get dpfoperatorconfig
NAME                 READY   PHASE     AGE
dpfoperatorconfig    True    Success   1h
```

or via `dpfctl`

```
$ kubectl -n dpf-operator-system exec deployment/dpf-operator-
controller-manager -- /dpfctl describe all
NAME                                        NAMESPACE
STATUS  REASON   SINCE  MESSAGE
DPFOperatorConfig/dpfoperatorconfig         dpf-operator-system
             Ready
True    Success  1h
             ImagePullSecretsReconciled
True    Success  1h
             SystemComponentsReady
True    Success  1h
             SystemComponentsReconciled
True    Success  1h
```

Configuration Options

Networking

There are networking options that can be configured. The MTU for the control plane and high-speed interfaces can be configured. The default value is set to 1500, however it can be adjusted if required.

```
spec:
  networking:
    controlPlaneMTU: 1500     # Management network MTU (range: 1280-
9216, default: 1500)
    highSpeedMTU: 1500        # High-speed interface MTU (range: 1280-
9216, default: 1500)
```

Image Pull Secrets

Specify secrets for pulling container images. This is only necessary if your container registry requires authentication. If you are using the public GHCR registry, which is the default, you don't need to configure this.

```
spec:
  imagePullSecrets:
    - "my-registry-secret"
    - "another-secret"
```

Resources

All system components deployed by the DPF Operator support standard Kubernetes resource requests and limits. Resources can be configured per component at the container level. Components may have multiple containers with different resource requirements that can be configured independently.

Below is an example of configuring resources for the SFC Controller component:

```
spec:
  sfcController:
    controller:
      resources:
        requests:
          cpu: 6
          memory: 2Gi
        limits:
          cpu: 8
          memory: 4Gi
```

This pattern applies to all components listed in the Optional Component Configurations section below.

For production deployments, it's recommended to set appropriate resource limits based on your cluster's workload.

Optional Component Configurations

The following components can be configured to enable/disable features or specify a different container image.

By default, all components are enabled with preconfigured images, and changes are usually only needed for development, testing, or specific deployments.

```
spec:
  cniInstaller: { }
  dpuDetector: { }
  dpuServiceController: { }
  flannel: { }
  kamajiClusterManager: { }
  multus: { }
  nvipam: { }
  ovsCNI: { }
  provisioningController: { }
  serviceSetController: { }
  sfcController: { }
  sriovDevicePlugin: { }
  staticClusterManager: { }
```

To disable a component or override its container image, use the following configuration:

```
spec:
  sriovDevicePlugin:
    disable: true
  dpuDetector:
    daemon:
      image: "my-registry/my-dpu-detector:latest"
```

> ⚠ **Warning**
>
> **Deprecated:** Setting the image at component level (e.g., `spec.dpuDetector.image`) is deprecated. Use the sub-component specific image field instead (e.g., `spec.dpuDetector.daemon.image`).

For a detailed description of each component and its available configuration options, see the API Reference.

DPU Service Controller Configuration options

- `spec.dpuServiceController.disableDPUReadyTaints`: When set to true, disables the automatic tainting of DPU nodes when they're not ready.

```
spec:
  dpuServiceController:
    disableDPUReadyTaints: true
```

Flannel Configuration Options

- `spec.flannel.podCIDR`: CIDR range for pod networking when using Flannel CNI.

```
spec:
  flannel:
    podCIDR: "10.244.0.0/16"
```

Component Deployment Configuration

Several components support additional deployment configuration options:

- `helmChart`: Override the Helm chart repository/version for the component

```
spec:
  multus:
    helmChart: "custom-repo/multus:v1.0.0"
```

SFC Controller Configuration Options

- `spec.sfcController.SecureFlowDeletionTimeout`: Used to control the secure flow deletion feature.

  The default value is 0, which means that the feature is disabled.

  When set with a valid duration value, indicating the API server unavailability threshold, SFC controller will delete all openflow flows to prevent unintended packet leaks, if API server is unavailable for more than the specified duration.

  Value must be in units accepted by Go time.ParseDuration https://golang.org/pkg/time/#ParseDuration.

```
spec:
  sfcController:
    SecureFlowDeletionTimeout: 5m
```

Provisioning Controller Configuration Options

- `spec.provisioningController.bfbPVCName`: **(Required)** Name of the PVC containing the BFB (BF Bundle) for provisioning DPUs.

- `spec.provisioningController.maxDPUParallelInstallations`: Controls the maximum number of DPUs that can be provisioned concurrently. The default value is 50. The value must be at least 1.

- `spec.provisioningController.maxUnavailableDPUNodes`: Maximum number of DPU nodes that can be unavailable during updates. The provisioning controller interacts with the maintenance-operator to implement the drain node effect. The number of nodes that can be applied node effect simultaneously is determined by MaxUnavailableDPUNodes in dpfoperatorconfig and MaxParallelOperations in the NodeMaintenance-operator configuration. NodeMainteanceOperator has higher priority than what is defined in the DPFOperatorConfig. The default value of DPFOperatorConfig.MaxUnavailableDPUNodes is 50. For the default MaintenanceOperatorConfig values see instructions in helm prerequisites.

The maxDPUParallelInstallations and maxUnavailableDPUNodes options can be configured together and can be combined with maxParallelOperations and maxUnavailable in Nvidia NodeMaintenance-operator configuration. Below are some examples to show the expected behaviour.

maxDPUParallelInstallations in DPFOperatorconfig	**maxUnavailableDPUNodes in DPFOperatorconfig**	**maxParallelOperations in Nvidia NodeMaintenanceConfig**	**max Noc**
5	1	10	5
1	5	10	10
5	5	1	5
5	5	10	2

- `spec.provisioningController.bfCFGTemplateConfigMap`: Name of ConfigMap containing bf-cfg template for DPU configuration.

- `spec.provisioningController.customCASecretName`: Name of Secret containing custom CA certificates for secure communication.

- `spec.provisioningController.dmsTimeout`: Timeout in seconds for DMS (DPU Management Service) operations.
- `spec.provisioningController.multiDPUOperationsSyncWaitTime`: Wait time for synchronizing operations across multiple DPUs. Value must be in units accepted by Go time.ParseDuration https://golang.org/pkg/time/#ParseDuration.
- `spec.provisioningController.registry`: Configuration for the container registry used during provisioning.
  - `address`: Registry address
  - `port`: Registry port
- `spec.provisioningController.installInterface`: Method for installing DPU firmware. Choose one:
  - `installViaHostAgent`: Install via host agent
  - `installViaGNOI`: Install via gNOI protocol
  - `installViaRedfish`: Install via Redfish API with additional options:
    - `bfbRegistry.disable`: Disable the BFB registry
    - `bfbRegistry.port`: Port for BFB registry
    - `bfbRegistryAddress`: Address of BFB registry
    - `skipDpuNodeDiscovery`: Skip automatic DPU node discovery

```
spec:
  provisioningController:
    bfbPVCName: bfb-pvc
    maxDPUParallelInstallations: 25  # Limit concurrent
provisioning to 25 DPUs
    maxUnavailableDPUNodes: 5
    dmsTimeout: 600
    multiDPUOperationsSyncWaitTime: 30s
    customCASecretName: my-ca-secret
    registry:
      address: "registry.example.com"
      port: 5000
    installInterface:
      installViaRedfish:
        bfbRegistry:
          port: 8080
        skipDpuNodeDiscovery: false
```

Advanced Overrides

The `overrides` section allows customization of system-level paths and settings. These are typically only needed for non-standard deployments or testing scenarios.

```yaml
spec:
  overrides:
    # Pause reconciliation of the DPFOperatorConfig
    paused: false

    # Kubernetes API server configuration
    kubernetesAPIServerVIP: "192.168.1.100"
    kubernetesAPIServerPort: 6443

    # DPU filesystem paths for CNI
    dpuCNIPath: "/etc/cni/net.d"
    dpuCNIBinPath: "/opt/cni/bin"

    # DPU OpenVSwitch paths
    dpuOpenvSwitchBinPath: "/usr/bin"
    dpuOpenvSwitchRunPath: "/var/run/openvswitch"
    dpuOpenvSwitchSystemSharedPath: "/usr/share/openvswitch"
    dpuOpenvSwitchSystemSharedLib64Path: "/usr/lib64"

    # Flannel-specific overrides
    flannelSkipCNIConfigInstallation: false
```

Override Options

- `paused`: When set to true, pauses reconciliation of the DPFOperatorConfig resource.
- `kubernetesAPIServerVIP`: Override the Kubernetes API server virtual IP address.
- `kubernetesAPIServerPort`: Override the Kubernetes API server port (default: 6443).
- `dpuCNIPath`: Path to CNI configuration directory on DPU nodes.
- `dpuCNIBinPath`: Path to CNI binaries on DPU nodes.

- `dpuOpenvSwitchBinPath`: Path to OpenvSwitch binaries on DPU nodes.
- `dpuOpenvSwitchRunPath`: Path to OpenvSwitch runtime directory on DPU nodes.
- `dpuOpenvSwitchSystemSharedPath`: Path to OpenvSwitch shared directory on DPU nodes.
- `dpuOpenvSwitchSystemSharedLib64Path`: Path to OpenvSwitch 64-bit libraries on DPU nodes.
- `flannelSkipCNIConfigInstallation`: Skip automatic CNI configuration installation for Flannel.

DPUCluster

The DPUCluster is a Kubernetes CRD which managed the control plane of a DPUCluster in DPF. The DPUCluster can be backed by different implementations.

Two implementations are included in this repo:

- Kamaji cluster manager which creates Kamaji TenantControlPlanes to back the DPUCluster
- Static cluster manager which transforms an existing Kubernetes control plane into a DPUCluster control plane

DPUCluster Usage

A DPUCluster is a user API and the usage will differ depending on the implementation.

Using the Static Cluster Manager

The static cluster manager controller should be enabled first. It is enabled by adding staticClusterManager field in the DPUOperatorConfig CR:

```
apiVersion: operator.dpu.nvidia.com/v1alpha1
kind: DPFOperatorConfig
metadata:
  name: dpfoperatorconfig
  namespace: dpf-operator-system
spec:
  provisioningController:
    bfbPVCName: "bfb-pvc"
  staticClusterManager: {}
```

Then create a secret for storing the kubeconfig of the existing Kubernetes control plane. For example, the kubeconfig is under the home directory:

```
TENANT_KUBE_CONFIG=`cat ~/.kube/config | base64 -w 0`

cat <<EOF | kubectl apply -f -
apiVersion: v1
data:
  super-admin.conf: ${TENANT_KUBE_CONFIG}
kind: Secret
metadata:
  name: dpu-cluster-1-admin-kubeconfig
  namespace: dpf-operator-system
type: Opaque
EOF
```

the DPUCluster will look like:

```
apiVersion: provisioning.dpu.nvidia.com/v1alpha1
kind: DPUCluster
metadata:
  name: dpu-cluster-1
  namespace: dpf-operator-system
spec:
  ## type signals which controller implementation should take
responsibility for the DPUCluster.
  type: static
  ## Max nodes is the maximum number of nodes supported by the
DPUCluster implementation.
  maxNodes: 10
  ## Kubeconfig is the name of a secret in the same namespace as
the DPUCluster object.
  ## Note: This field is supplied by the user in the static cluster
manager - but this may not be the case for other implementations.
  kubeconfig: dpu-cluster-1-admin-kubeconfig
```

Using the Kamaji Cluster Manager

The DPUCluster will look like:

```
apiVersion: provisioning.dpu.nvidia.com/v1alpha1
kind: DPUCluster
metadata:
  name: dpu-cluster-1
  namespace: dpf-operator-system
spec:
  ## type signals which controller implementation should take
responsibility for the DPUCluster.
  type: kamaji
  ## Max nodes is the maximum number of nodes supported by the
DPUCluster implementation.
  maxNodes: 10
  ## Cluster endpoint is supplied by the user and provides and IP
and other details to make the APIServer available.
  clusterEndpoint:
    # deploy keepalived instances on the nodes that match the
given nodeSelector.
    keepalived:
      # interface on which keepalived will listen. Should be the oob
interface of the control plane node.
      interface: interface_one
      # vip is the Virtual IP reserved for the DPU Cluster load
balancer. Must not be allocatable by DHCP.
      vip: dpucluster_vip
      # virtualRouterID must be in range [1,255], make sure the
given virtualRouterID does not duplicate with any existing
      # keepalived process running on the host
      virtualRouterID: 126
      # nodeSelector selects which nodes the keepalived pods will
be scheduled to.
      nodeSelector:
        node-role.kubernetes.io/control-plane: ""
```

DPUCluster Implementation

A DPUCluster implementation is a Kubernetes controller which operates on the DPF DPUCluster object. It should:

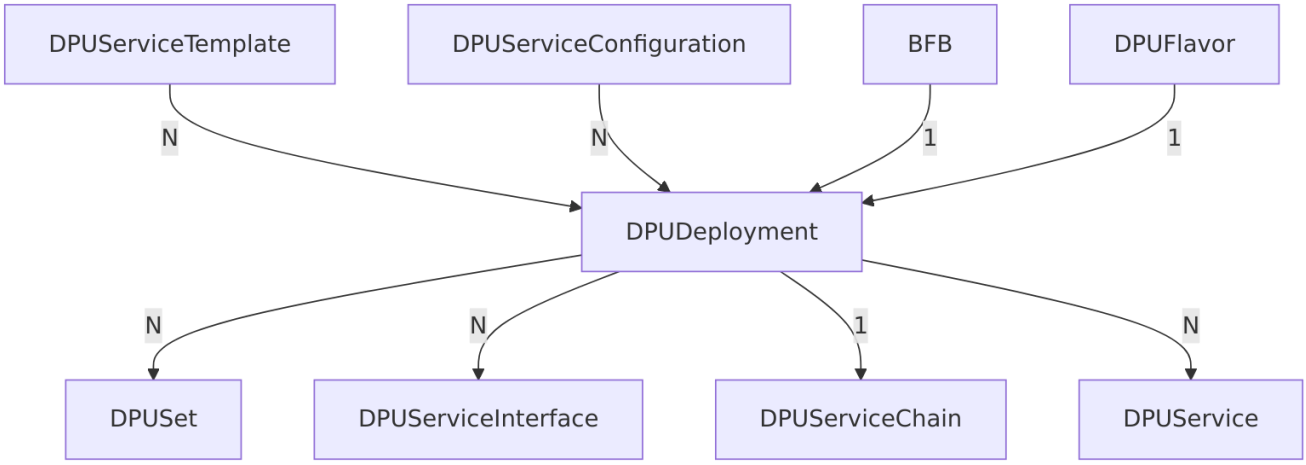
- only operate on a DPUCluster which has a `type` it is responsible for.
- be the only DPUCluster controller implementation in a cluster.
- provide an admin Kubeconfig to a functioning Kubernetes cluster as a Kubernetes Secret.
- ensure the name of that Secret is available in the `.spec.kubeconfig` of the DPUCluster object.

The Kubeconfig provided by the DPUCluster should have the following format:

```
apiVersion: v1
kind: Secret
metadata:
  name: dpu-cluster-1
  namespace: dpf-operator-system
type: Opaque
data:
  super-admin.conf: $KUBECONFIG_DATA
```

DPUDeployment

This document describes how a user can work with the `DPUDeployment` Custom Resource. A `DPUDeployment` describes a set of `DPUServices` and a `DPUServiceChain` that run on a set of DPUs with a given `BFB` and `DPUFlavor`.



Throughout this document, there are examples for the required Custom Resources that end up building a valid `DPUDeployment` Custom Resource. These examples contain comments related to fields that are set for more context. The theoretical example is about 3 services, one of them producing work, another one executing work (producer-consumer problem) and the last one observing the state.

Capabilities

- Validates dependencies to ensure that they are configured correctly and reports errors in the `DPUDeployment` status conditions accordingly.
- Validates that requested `DPUService` resources fit the DPUs they are targeting and report errors in the `DPUDeployment` status conditions accordingly.
- Validates that the version requirements of the `DPUService` fit the versions found in the given `BFB` and reports errors in the `DPUDeployment` status conditions accordingly.
- Gracefully handles synchronized disruptive and non-disruptive updates of the underlying objects.

Created Child Custom Resources

When applying a valid `DPUDeployment` that has all of its dependencies set correctly and available, there will be a couple of objects that are going to be created automatically:

- `DPUSet`: Deploys a given `BFB` with configuration provided by the given `DPUFlavor` to the target `DPUs`. A `DPUDeployment` may create multiple such

objects, depending on what is specified in its `spec`.

- `DPUServiceInterface`: Used to construct a Service Chain on the DPU. A `DPUDeployment` may create multiple such objects, depending on what is specified in the DPUServiceConfiguration.
- `DPUServiceChain`: Used to define a Service Chain on the DPU that references the interfaces created above. A `DPUDeployment` creates a single `DPUServiceChain`.
- `DPUService`: Deploys a service as Pod in each DPU or in the nodes part of the Host cluster. Standard `DPUServices` are deployed on the DPUs, while in-cluster `DPUServices` are deployed on the Host cluster nodes. A `DPUDeployment` may create multiple such objects, depending on what is specified in its `spec`.

Prerequisite Custom Resources With Examples

There are several Custom Resources that are required in order to make use of the `DPUDeployment`. These are:

- DPUServiceTemplate
- DPUServiceConfiguration
- DPUFlavor
- BFB

DPUServiceTemplate

A `DPUServiceTemplate` contains configuration options related to resources required by the `DPUService` to be deployed. This Custom Resource is usually provided by NVIDIA for the supported `DPUServices` that are published. It helps generate the underlying `DPUService`. It is the base configuration for the `DPUService` that is getting merged with configuration provided by the `DPUServiceConfiguration`. If there is a conflict, settings in `DPUServiceConfiguration` take precedence.

A user must create as many `DPUServiceTemplate` Custom Resources as the number of services they aim to deploy using a `DPUDeployment`. In this example, we will need to create 3 of those since we have 3 `DPUServices`.

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceTemplate
metadata:
  name: producer
  namespace: customer-namespace
spec:
  deploymentServiceName: "producer" # Must match the key in the
`dpudeployment.spec.services`
  helmChart:
    source:
      repoURL: https://example.com/charts
      path: producer
      version: v0.0.1
    # The `DPUServiceTemplate` owner should specify requests and
limits for the actual containers. The
    # `DPUServiceTemplate` owner is responsible to ensure that
those resources are not exceeding the specified
    # `resourceRequirements` field. Notice that setting resources
on containers may be different per Helm Chart
    # implementation.
    values:
      container1:
        resources:
          requests:
            cpu: 0.2
            memory: 0.2Gi
            nvidia.com/sf: 1
          limits:
            cpu: 0.4
            memory: 0.4Gi
            nvidia.com/sf: 1
      container2:
        resources:
          limits:
```

```
            cpu: 0.6
            memory: 0.6Gi
  # resourceRequirements contains the overall resources required
by this particular service to run on a single node
  # This is the field that is considered when scheduling a
`DPUDeployment`.
  resourceRequirements:
    cpu: 1
    memory: 1Gi
    nvidia.com/sf: 1
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceTemplate
metadata:
  name: consumer
  namespace: customer-namespace
spec:
  deploymentServiceName: "consumer"
  helmChart:
    source:
      repoURL: https://example.com/charts
      path: consumer
      version: v0.0.1
    values:
      worker:
        parallelization: 5
        resources:
          requests:
            cpu: 1
            memory: 1Gi
            nvidia.com/sf: 1
          limits:
            cpu: 1
            memory: 2Gi
            nvidia.com/sf: 1
  resourceRequirements:
```

```
    cpu: 2
    memory: 4Gi
    nvidia.com/sf: 1
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceTemplate
metadata:
  name: observer
  namespace: customer-namespace
spec:
  deploymentServiceName: "observer"
  helmChart:
    source:
      repoURL: https://example.com/charts
      path: observer
      version: v0.0.1
    values:
      observer:
        resources:
          requests:
            cpu: 0.5
            memory: 0.5Gi
          limits:
            cpu: 0.5
            memory: 1Gi
```

DPUServiceConfiguration

A `DPUServiceConfiguration` contains all configuration options from the user to be provided to the `DPUService` via the Helm values. This Custom Resource is usually crafted by the user according to their environment and intended use of the `DPUService`. It helps generate the underlying `DPUService`. It is a layer on top of the configuration defined in the `DPUServiceTemplate`. This configuration is getting merged with

configuration provided by the `DPUServiceTemplate`. If there is a conflict, settings in `DPUServiceConfiguration` take precedence.

A user must create as many `DPUServiceConfiguration` Custom Resources as the number of services they aim to deploy using a `DPUDeployment`. In this example, we will need to create 3 of those since we have 3 `DPUServices`.

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceConfiguration
metadata:
  name: producer
  namespace: customer-namespace
spec:
  deploymentServiceName: "producer" # Must match the key in the
`dpudeployment.spec.services`
  serviceConfiguration:
    serviceDaemonSet:
      labels:
        sre.nvidia.com/service-tier: "t1"
      annotations:
        sre.nvidia.com/page: "false"
  interfaces:
  - name: app-iface
    network: mynad
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceConfiguration
metadata:
  name: consumer
  namespace: customer-namespace
spec:
  deploymentServiceName: "consumer"
  serviceConfiguration:
    # The `DPUServiceConfiguration` owner may choose to override
some of the settings defined by the
    # `DPUServiceTemplate` or add new. It is not recommended to
change container resources in this resource but rather
    # in DPUServiceTemplate.
    helmChart:
      values:
        worker:
```

```
          parallelization: 10
    serviceDaemonSet:
      labels:
        sre.nvidia.com/service-tier: "t1"
      annotations:
        sre.nvidia.com/page: "true"
  # interfaces describes the network each interface of this
application needs to be attached to so that it can function
  # and be available for use in the Service Chain framework.
  interfaces:
  - name: app-iface
    network: mynad
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceConfiguration
metadata:
  name: observer
  namespace: customer-namespace
spec:
  deploymentServiceName: "observer"
  serviceConfiguration:
    deployInCluster: true # Indicates that the service should be
deployed on the host cluster instead of the DPUCluster
    serviceDaemonSet:
      labels:
        sre.nvidia.com/service-tier: "t2"
      annotations:
        sre.nvidia.com/page: "false"
```

`spec.deploymentServiceName` must match the key in the `spec.services` field of the `DPUDeployment`.

`spec.Interfaces` is a list of interfaces that the `DPUService` should have. They can be referenced in the `spec.serviceChains` of the `DPUDeployment`.

`spec.serviceConfiguration.deployInCluster` is a boolean that indicates whether the service should be deployed in the Host cluster (where the DPF operator is running) rather than on the DPUs. When set to `true`, the service is deployed in the Host cluster.

`spec.upgradePolicy.applyNodeEffect` is a boolean that indicates whether the service update should be disruptive or not. The default is `true`, which means that a new version of the service is created for every new version of the `DPUServiceConfiguration` and the node effect defined in the dpuset is triggered for the relevant nodes for the update to happen. If set to `false`, the service is updated non-disruptively.

DPUFlavor

A `DPUFlavor` describes the configuration to be applied on the DPU during the provisioning. This is a very minimal `DPUFlavor` as the purpose of this document is to demonstrate the capabilities of the `DPUDeployment`. Given that, there are 2 fields set that are related to the `DPUDeployment`.

```
apiVersion: provisioning.dpu.nvidia.com/v1alpha1
kind: DPUFlavor
metadata:
  name: producer-consumer
  namespace: customer-namespace
spec:
  # dpuResources indicates the minimum amount of resources needed
for a BFB with that flavor to be installed on a
  # DPU. Using this field, the controller can understand if that
flavor can be installed on a particular DPU. It
  # should be set to the total amount of resources the system
needs + the resources that should be made available for
  # DPUServices to consume.
  dpuResources:
    cpu: 16
    memory: 16Gi
    nvidia.com/sf: 20
  # systemReservedResources indicates the resources that are
consumed by the system (OS, OVS, DPF system etc) and are
  # not made available for DPUServices to consume. DPUServices can
consume the difference between DPUResources and
  # SystemReservedResources. This field must not be specified if
dpuResources are not specified.
  systemReservedResources:
    cpu: 4
    memory: 4Gi
    nvidia.com/sf: 4
```

The above configuration translates to the following resources being available for the `DPUServices` deployed by the `DPUDeployment`.

```
allocatableResources:
  cpu: 12
  memory: 12Gi
  nvidia.com/sf: 16
```

BFB

A `BFB` describes the BFB to be flashed on the DPU during the provisioning.

```
apiVersion: provisioning.dpu.nvidia.com/v1alpha1
kind: BFB
metadata:
  name: bfb-2.9
  namespace: customer-namespace
spec:
  fileName: "bfb-2.9.0.bfb"
  url: "http://internal-nfs/bf-bundle-2.9.0-33_24.04_ubuntu-22.04_unsigned.bfb"
```

DPUDeployment Example

The following `DPUDeployment` example is based on the Custom Resources found above. It describes a `DPUDeployment` which targets 2 sets of DPUs, provisioned with a specific `DPUFlavor` and `BFB`, and all of them running 3 `DPUServices`.

```
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUDeployment
metadata:
  name: producer-consumer
  namespace: customer-namespace
spec:
  dpus:
    # bfb references the `BFB` object
    bfb: "bfb-2.9"
    # flavor references the `DPUFlavor` Custom Resource
    flavor: "producer-consumer"
    # dpuSets enables the user to select the DPUs this
`DPUDeployment` should deploy to. It's a list so that the user
    # can be as flexible as possible. In this example, we
theoretical target Hosts in 2 different racks, and we target
    # the DPUs that have the specified PCI address.
    dpuSets:
    - nameSuffix: "dpuset1"
      nodeSelector:
        matchLabels:
          datacenter.nvidia.com/rack: "b-100"
      dpuSelector:
        provisioning.dpu.nvidia.com/dpudevice-pciAddress:
"0000:0e:00.0"
    - nameSuffix: "dpuset2"
      nodeSelector:
        matchLabels:
          datacenter.nvidia.com/rack: "b-101"
      dpuSelector:
        provisioning.dpu.nvidia.com/dpudevice-pciAddress:
"0000:1a:00.0"
  # services reflects the `DPUServices` that should be deployed
on those DPUs. For in-cluster `DPUServices` like the
  # observer, the pods will be deployed on the host cluster and
target the nodes that the DPUSet nodeSelectors target.
```

```
  # The key of this map is the service name and the value is
referencing the respective `DPUServiceTemplate` and
  # `DPUServiceConfiguration` for each service.
  services:
    producer:
      serviceTemplate: "producer"
      serviceConfiguration: "producer"
    consumer:
      serviceTemplate: "consumer"
      serviceConfiguration: "consumer"
    observer:
      serviceTemplate: "observer"
      serviceConfiguration: "observer"
  # serviceChains defines the `DPUServiceChain` that should be
created as part of this `DPUDeployment`.
  serviceChains:
    switches:
    - ports:
      - service:
          name: producer # The value must match the key in the
`spec.services`
          interface: app-iface # The value must match the
`dpuserviceconfiguration.spec.interfaces[].name`
      - service:
          name: consumer
          interface: app-iface
     #  Notice that the user can also reference other
DPUServiceInterfaces that may already exist or are created
manually
     #  by the user by specifying an entry like the one that
follows:
     # - serviceInterface:
     #     matchLabels:
     #       svc.dpu.nvidia.com/interface: p0
```

As mentioned in the Created Child Custom Resources section, after applying this manifest, the following objects are created:

```
$ kubectl get dpuset -A
NAMESPACE            NAME                        AGE
customer-namespace   producer-consumer-dpuset1   36m
customer-namespace   producer-consumer-dpuset2   36m

$ kubectl get dpuset -n customer-namespace
NAME                        AGE
producer-consumer-dpuset1   36m
producer-consumer-dpuset2  36m

$ kubectl get dpuserviceinterface -n customer-namespace
NAME                        READY   PHASE     IFTYPE     IFNAME
AGE
consumer-app-iface-w6tgf   True    Success   service    app-iface
36m
producer-app-iface-vqvs4   True    Success   service    app-iface
36m

$ kubectl get dpuservicechain -n customer-namespace
NAME                        READY   PHASE     AGE
producer-consumer-vpn7w   True    Success   36m

$ kubectl get dpuservice -n customer-namespace
NAME            READY   PHASE     AGE
consumer-fjfh8  True    Success   36m
observer-xk9p2  True    Success   36m
producer-ln2kk  True    Success   36m
```

Writing a DPUDeployment Spec

DPUs Configuration

The `spec.dpus` contains the configuration for the DPUs that the `DPUDeployment` should target.

```
spec:
  dpus:
    bfb: "bfb-2.9"
    flavor: "producer-consumer"
    dpuSets:
    - nameSuffix: "dpuset1"
      nodeSelector:
        matchLabels:
          datacenter.nvidia.com/rack: "b-100"
      dpuSelector:
        provisioning.dpu.nvidia.com/dpudevice-pciAddress: "0000:0e:00.0"
    - nameSuffix: "dpuset2"
      nodeSelector:
        matchLabels:
          datacenter.nvidia.com/rack: "b-101"
      dpuSelector:
        provisioning.dpu.nvidia.com/dpudevice-pciAddress: "0000:1a:00.0"
    nodeEffect:
      taint:
        key: "dpu"
        value: "provisioning"
        effect: NoSchedule
```

In the above example, the `DPUDeployment` targets 2 sets of DPUs. The first set targets the DPUs in rack `b-100` with the PCI address `0000:0e:00.0`. The second set targets the DPUs in rack `b-101` with the PCI address `0000:1a:00.0`.

The following fields are available in the `spec.dpus`:

- `bfb`: The `BFB` object to be flashed on the DPUs. It must exist in the same namespace as the `DPUDeployment`.
- `flavor`: The `DPUFlavor` object that describes the configuration to be applied on the DPU during the provisioning. It must exist in the same namespace as the `DPUDeployment`.
- `dpuSets`: A list of `DPUSet` configurations that describe the DPUs to be targeted by the `DPUDeployment`.
  - `nameSuffix`: A suffix to be added to the `DPUSet` name. This is a required field, as the `DPUSet` name must be unique and identifiable.
  - `nodeSelector`: The selector of the DPUNodes to which the DPUs are attached to. See more in DPU Selection. Note that this field maps to the DPUSet field `dpuNodeSelector`.
  - `dpuSelector`: The selector of the DPUDevices that are to be targeted. In this example, the DPUs are selected based on their PCI address. See more in DPU Selection.
  - `dpuAnnotations`: The annotation to be applied on the DPU objects that are created by the `DPUDeployment`.
- `nodeEffect`: The effect to be applied on the nodes to which the DPUs are attached. In this example, a `NoSchedule` taint is applied to the nodes.

See the Prerequisite Custom Resources With Examples section for examples of the `DPUFlavor` and `BFB` Custom Resources.

See the DPUSets document for more information on the `DPUSet` Custom Resource.

Services Configuration

The `spec.services` contains the configuration for the services that the `DPUDeployment` should deploy.

```
spec:
  services:
    producer:
      serviceTemplate: "producer"
      serviceConfiguration: "producer"
    consumer:
      serviceTemplate: "consumer"
      serviceConfiguration: "consumer"
    observer:
      serviceTemplate: "observer"
      serviceConfiguration: "observer"
```

The following fields are available in the `spec.services`:

- `serviceTemplate`: The `DPUServiceTemplate` object that describes the configuration for the service to be deployed.
- `serviceConfiguration`: The `DPUServiceConfiguration` object that describes the configuration for the service to be deployed.

Both `serviceTemplate` and `serviceConfiguration` must be provided for each service that the `DPUDeployment` should deploy and must exist in the same namespace as the `DPUDeployment`.

See the Prerequisite Custom Resources With Examples section for examples of the `DPUServiceTemplate` and `DPUServiceConfiguration` Custom Resources.

Dependencies configuration

The `spec.services.dependsOn` field is used to specify the dependencies between the `DPUServices`. The reconciler will enforce the dependency order during the deployment of the `DPUServices`. The `LocalObjectDependency` object contains the following fields:

- `name`: The name of the dependency. This field is required and must match the name of the `DPUService` that is being depended on, i.e. it must exist in

`spec.services`.

```
spec:
  services:
    producer:
      serviceTemplate: "producer"
      serviceConfiguration: "producer"
    consumer:
      serviceTemplate: "consumer"
      serviceConfiguration: "consumer"
      dependsOn:
      - name: producer
    observer:
      serviceTemplate: "observer"
      serviceConfiguration: "observer"
      dependsOn:
      - name: producer
      - name: consumer
```

In the above example, the `consumer` service depends on the `producer` service, and the `observer` service depends on both `producer` and `consumer` services. The services will not be deployed until their dependencies are deployed.

Templating

The `DPUServiceConfiguration` supports Go templating, allowing you to create dynamic configurations that can be customized based on provided parameters.

Here's a basic example of using templating in a DPUServiceConfiguration:

```
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceConfiguration
metadata:
  name: consumer
  namespace: customer-namespace
spec:
  deploymentServiceName: "consumer"
  serviceConfiguration:
    helmChart:
      values:
        other-service:
          name: {{ .Services.producer.Name }}
```

When accessing values in the template, you can use the `{{ .Services.ServiceName.Field }}` syntax, where `ServiceName` is the name of the service as defined in `spec.services` and `Field` is a field of the `DPUService` object.

It is not possible to access a field with a dash in its name, e.g. `{{ .Services.ServiceName.Field-With-Dash }}` the same way. This is due to the limitations of Go templating. Instead the `index` function can be used to access such fields, e.g. `{{ (index .Services "firefly-dpu").Name }}`.

Available Template Variables

The following variables are available in your templates:

- `.Services`: A map of services in the `DPUDeployment`. In order for a service to be available in the template, it must be referenced as a dependency in the `spec.services.dependsOn` field. A service can then be referenced using the following syntax: `{{ .Services.ServiceName.Field }}`, where `ServiceName` is the name of the service as defined in `spec.services` and `Field` is a field of the `DPUService` object.

At the moment, the following fields are available:

- `.Name`: The name of the `DPUService` generated by the `DPUDeployment` controller.

Template Delimiters

By default, the system uses the standard Go template delimiters `{{` and `}}`. However, you can customize these delimiters using the `svc.dpu.nvidia.com/template-delimiter` annotation on your DPUServiceConfiguration:

```
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceConfiguration
metadata:
  name: consumer
  namespace: customer-namespace
  annotations:
    svc.dpu.nvidia.com/template-delimiter: "{{,}}"  # Default
delimiters
    # OR
    svc.dpu.nvidia.com/template-delimiter: "[[,]]"  # Custom
delimiters
```

Service Chains Configuration

The `spec.serviceChains` contains the configuration for the `DPUServiceChain` that the `DPUDeployment` should create.

```
spec:
  serviceChains:
    switches:
      - ports:
        - service:
            name: producer
            interface: app-iface
        - serviceInterface:
            matchLabels:
              svc.dpu.nvidia.com/interface: p0
```

The following fields are available in the `spec.serviceChains`:

- `upgradePolicy.applyNodeEffect`: A boolean that indicates whether the service chain update should be disruptive or not. The default is `true`, which means that a new version of the service chain is created for every new version of the `DPUService` and the node effect defined in the dpuset is triggered for the relevant nodes for the update to happen. If set to `false`, the service chain is updated non-disruptively. The disruptive operation applies the node effect defined for the `DPUSet` on the nodes.
- `switches`: A list of switches that are part of the service chain.
  - `ports`: A list of ports that are part of the switch.
    - `service`: Holds the configuration for an interface. The service referenced by this field must be defined in the corresponding `DPUServiceConfiguration`. See the Prerequisite Custom Resources With Examples section for examples of the `DPUServiceConfiguration` Custom Resource.
      - `name`: The name of the service. This field must match the service name defined as key `spec.services`.
      - `interface`: The name of the interface. This interface is injected by the cni plugin when the pod is scheduled.

        - `ipam`: The IPAM configuration for the interface. This field is optional and can be used to specify the IPAM configuration for the interface.

    - `serviceInterface`: Holds the configuration for an existing interface.

        - `matchLabels`: The labels to be used to select the interface. This field is required and must match the labels of the `DPUServiceInterface`.

        - `ipam`: The IPAM configuration for the interface. This field is optional and can be used to specify the IPAM configuration for the interface.

The `DPUDeployment` controller creates a single `DPUServiceChain` based on the configuration provided in the `spec.serviceChains`. The `DPUServiceChain` is created in the same namespace as the `DPUDeployment`.

See DPUServiceChain for more information on the `DPUServiceChain` Custom Resource.

Working with DPUDeployment

Waiting for Ready

When a DPUDeployment is created, it may take some time for all the underlying objects to be created and for the DPUs to be provisioned.

It is possible to wait for a DPUDeployment to be ready by using the `kubectl wait`

```
$ kubectl wait --for=condition=Ready dpudeployment/<dpudeployment-name> -n <namespace>
```

DPUDeployment Updates

A `DPUDeployment` can be updated by modifying `.spec` of the custom resource or by changing a referenced object like `DPUServiceTemplate` and `DPUServiceConfiguration`. The update of the underlying objects is specific to each kind:

- `DPUSet` can be updated by modifying `.spec.dpus`. The underlying `dpus` can be reprovisioned if the referenced `bfb` or `DPUFlavor` change.
- `DPUServices` can be updated by modifying `spec.Services`. Changing the referenced `DPUServiceTemplate` or `DPUServiceConfiguration` will update the selected `DPUService`. A differentiation is made for "disruptive DPUServices" which have an impact on the cluster nodes and "non-disruptive" ones that do not.
- `DPUServiceInterface` can be updated by modifying the referenced `DPUServiceConfiguration` `spec.Interfaces`.
- `DPUServiceChain` can be updated by modifying `spec.ServiceChains`.

Note: Users should avoid manually modifying an object owned by a `DPUDeployment`, as doing so can lead to unforeseen consequences that may disrupt the entire setup. The controller does not recognize these manual changes and may or may not overwrite them to reach the desired state.

Non-disruptive DPUService Update

1. Retrieve the reference `DPUServiceConfiguration` or `DPUServiceTemplate`:

```
$ kubectl get dpuserviceconfiguration -n customer-namespace
NAME        AGE
producer    36m
```

2. We should get a valid `DPUServiceConfiguration`:

```
$ kubectl get dpuserviceconfiguration producer -n customer-
namespace -o yaml
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceConfiguration
metadata:
  name: producer
  namespace: customer-namespace
spec:
  deploymentServiceName: "producer"  # Must match the key in the
`dpudeployment.spec.services`
  serviceConfiguration:
    serviceDaemonSet:
      labels:
        sre.nvidia.com/service-tier: "t1"
      annotations:
        sre.nvidia.com/page: "false"
  interfaces:
  - name: app-iface
    network: mynad
```

3. As an example let's update the requested `interface` name. In this case a new `DPUServiceInterface` is expected as this field is part of the `DPUServiceInterface` name. This is the only case where a new `DPUServiceInterface` is expected, otherwise the existing one will be updated:

```
$ kubectl patch dpuserviceconfiguration producer \
  -n customer-namespace \
  --type='json' \
  -p='[{"op": "replace", "path": "/spec/interfaces/0/name", "value":"app-iface2"}]'
```

4. The `DPUService` should be updated by the `DPUDeployment` controller:

```
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUService
metadata:
  annotations:
    svc.dpu.nvidia.com/dpuservice-version: f4295be911
  finalizers:
  - dpu.nvidia.com/dpuservice
  labels:
    svc.dpu.nvidia.com/owned-by-dpudeployment: producer-consumer-
dpudeployment
  name: producer-consumer-producer-2444q
  namespace: customer-namespace
spec:
...
  serviceID: dpudeployment_producer-consumer-
dpudeployment_producer-consumer-producer
  interfaces:
  - producer-app-iface2-748qf # Notice that this field is updated to match the new
DPUServiceInterface
```

5. A new `DPUServiceInterface` is created by the `DPUDeployment` controller:

```
$ kubectl get dpuserviceinterface -n customer-namespace
NAME                        READY   PHASE     IFTYPE    IFNAME
AGE
producer-app-iface2-748qf   True    Success   service   app-iface
5m
```

Disruptive DPUService Update

Updating "disruptive DPUServices" involves creating a new instance for every new version. Both standard `DPUServices` (deployed on DPUs) and in-cluster `DPUServices`

(deployed on Host cluster nodes) are supported. For in-cluster services, the `DPUDeployment` controller manages the lifecycle of labels on the Host cluster nodes to enable proper scheduling and targeting.

In addition, `DPUServiceInterfaces` are created for the new `DPUService` instances. Up to `revisionHistoryLimit` instances can exist at a given time, e.g. when changes are made to the `DPUServiceConfiguration` or `DPUServiceTemplate` while no instance has reached a `ready` state yet.

1. Retrieve the reference `DPUServiceConfiguration` or `DPUServiceTemplate`:

```
$ kubectl get dpuserviceconfiguration -n customer-namespace
NAME       AGE
producer   36m
```

2. We should get a valid `DPUServiceConfiguration`:

```
$ kubectl get dpuserviceconfiguration producer -n customer-
namespace -o yaml
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceConfiguration
metadata:
  name: producer
  namespace: customer-namespace
spec:
  deploymentServiceName: "producer" # Must match the key in the
`dpudeployment.spec.services`
  serviceConfiguration:
    serviceDaemonSet:
      labels:
        sre.nvidia.com/service-tier: "t1"
      annotations:
        sre.nvidia.com/page: "false"
  interfaces:
  - name: app-iface
    network: mynad
```

3. Make the `DPUService` disruptive by changing the `upgradePolicy.applyNodeEffect`:

```
$ kubectl patch dpuserviceconfiguration producer \
  -n customer-namespace \
  --type='json' \
  -p='[{"op": "add", "path": "/spec/upgradePolicy/applyNodeEffect", "value": true }]'
```

4. As an example let's update the requested `interface` name:

```
$ kubectl patch dpuserviceconfiguration producer  \
  -n customer-namespace \
  --type='json'  \
  -p='[{"op": "replace", "path": "/spec/interfaces/0/name", "value":"app-iface2"}]'
```

5. The `DPUService` should be updated by the `DPUDeployment` controller by adding a new version. In addition, a new `DPUServiceInterface` is created for the new `DPUService`.

```
$ kubectl get dpuservices -n customer-namespace
NAME                     READY   PHASE     AGE
producer-consumer-2444q True    Success   27m
producer-consumer-rr45f False   Pending   1m
```

```
$ kubectl get dpuserviceinterface -n customer-namespace
NAME                        READY   PHASE     IFTYPE    IFNAME
AGE
producer-app-iface-vqvs4    True    Success   service   app-iface
27m
producer-app-iface2-s6tb7   True    Success   service   app-iface
1m
```

Once the new version is `ready`, the `DPUDeployment` controller garbage collect the previous versions. In addition, it removes the stale `DPUServiceInterfaces` associated with the old previous versions

```
$ kubectl get dpuservices -n customer-namespace
NAME                     READY   PHASE     AGE
producer-consumer-rr45f True   Success   5m
```

Non-disruptive DPUServiceChain Update

1. Retrieve the `DPUDeployment`:

```
$ kubectl get dpudeployment producer-consumer -n customer-
namespace
NAME                READY   PHASE     AGE
producer-consumer   True    Success   36m
```

2. As an example let's update the first `Switch` port interface:

```
$ kubectl patch dpudeployment producer-consumer \
  -n customer-namespace \
  --type='json' \
  -p='[{"op": "replace", "path": "/spec/serviceChains/switches/0/ports/1/service/interface", "value":"app-
iface2"}]'
```

3. The `DPUServiceChain` should be updated by the `DPUDeployment` controller:

```
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceChain
metadata:
  annotations:
    svc.dpu.nvidia.com/dpuservicechain-version: e4b5c6d5e1
  finalizers:
  - dpu.nvidia.com/dpuservicechain
  labels:
    svc.dpu.nvidia.com/owned-by-dpudeployment: producer-consumer-
dpudeployment
  name: producer-consumer-vpn7w
  namespace: customer-namespace
spec:
...
template:
  spec:
    template:
      spec:
        switches:
          - ports:
            - serviceInterface:
                matchLabels:
                  svc.dpu.nvidia.com/service: producer
                  svc.dpu.nvidia.com/interface: app-iface
            - serviceInterface:
                matchLabels:
                  svc.dpu.nvidia.com/service: consumer
                  svc.dpu.nvidia.com/interface: app-iface2
...
```

Disruptive DPUServiceChain Update

Updating "disruptive DPUServiceChains" involves creating a new instance for every new version. Up to `revisionHistoryLimit` instances can exist at a given time, e.g. when changes are made to the `DPUDeployment.spec.serviceChains` while no instance has reached a `ready` state yet.

1. Retrieve the `DPUDeployment`:

```
$ kubectl get dpudeployment producer-consumer -n customer-
namespace
NAME                READY   PHASE     AGE
producer-consumer   True    Success   36m
```

2. Make the `DPUServiceChain` disruptive by changing the `upgradePolicy.applyNodeEffect`:

```
$ kubectl patch dpudeployment producer-consumer \
  -n customer-namespace \
  --type='json' \
  -p='[{"op": "replace", "path": "/spec/serviceChains/upgradePolicy/applyNodeEffect", "value": true }]'
```

3. As an example let's update the first `Switch` port interface:

```
$ kubectl patch dpudeployment producer-consumer \
  -n customer-namespace \
  --type='json' \
  -p='[{"op": "replace", "path": "/spec/serviceChains/switches/0/ports/1/service/interface", "value":"app-
iface2"}]'
```

4. The `DPUServiceChain` should be updated by the `DPUDeployment` controller by adding a new version:

```
$ kubectl get dpuservicechains -n customer-namespace
NAME                      READY   PHASE     AGE
producer-consumer-vpn7w True    Success   25m
producer-consumer-rwe67 False   Pending   1m
```

Once the new version is `ready`, the `DPUDeployment` controller garbage collect the previous versions.

```
$ kubectl get dpuservicechains -n customer-namespace
NAME                      READY   PHASE     AGE
producer-consumer-rwe67 True   Success   5m
```

DPUService and BFB version matching

`DPUDeployment` has the capability of checking if the version constraints defined by the `DPUService` resources are satisfied against the `BFB`. A relevant condition in the `DPUDeployment` reflects whether the user provided `BFB` and `DPUServices` is a valid combination that can work. Below is an example of the condition when a mismatched combination is configured:

```
- lastTransitionTime: "2025-02-10T07:59:58Z"
  message: 'Error occurred: version constraint for "dpu.nvidia.com/doca-version"'
    found in DPUServiceTemplate "producer" is not satisfied by the
version
    "2.9.1" found in the given BFB'
  observedGeneration: 1
  reason: Error
  status: "False"
  type: VersionMatchingReady
```

Debugging DPUDeployments

General guideline

There are several ways to debug `DPUDeployments` in DPF. The recommended way is to use the dpfctl command line tool to describe the `DPUDeployment` and its underlying objects. The `dpfctl` tool provides a detailed description of the `DPUDeployment` and its underlying objects, including the status of the objects.

```
$ dpfctl describe dpudeployments
NAME                                      NAMESPACE
STATUS       REASON    SINCE  MESSAGE
DPFOperatorConfig/dpfoperatorconfig    dpf-operator-system
Ready: True  Success   28h
  DPUDeployments
    DPUDeployment/vpc-ovn              dpf-operator-system
Ready: True  Success   28h
      DPUServiceChains
        DPUServiceChain/vpc-ovn-trsq6  dpf-operator-system
Ready: True  Success   28h
      DPUSets
        DPUSet/vpc-ovn-dpuset1         dpf-operator-system
          BFB/bf-bundle                dpf-operator-system
Ready: True  Ready     3d23h  File: bf-bundle-3.2.1-
34_25.11_ubuntu-24.04_64k_prod.bfb, DOCA: 3.2.1
          DPUs
            4 DPUs...                  dpf-operator-system
Ready: True  DPUReady  3d22h  See dpu-node-mt2310xz03lr-
mt2310xz03lr, dpu-node-mt2310xz03m2-mt2310xz03m2,

dpu-node-mt2425601x13-mt2425601x13, dpu-node-mt2425601xqy-
mt2425601xqy
      Services
        DPUServiceTemplates
          4 DPUServiceTemplates...     dpf-operator-system
Ready: True  Success   3d23h  See ovn-central, ovn-controller,
vpc-ovn-controller, vpc-ovn-node
        DPUServices
          4 DPUServices...             dpf-operator-system
Ready: True  Success   3d7h   See ovn-central-9558p, ovn-
controller-v5bkr, vpc-ovn-controller-7sbp6, vpc-ovn-node-r84zn
```

Debugging disruptive upgrades

DPUDeployment is not ready because DPUs are stuck in Node Effect Removal

When DPUs are stuck in the Node Effect Removal phase, it indicates that the system is waiting for certain prerequisites to be met before proceeding with the upgrade. This typically means:

- **DPUService Pods in the DPUCluster are not ready yet**: Verify that the Pod deployed by the DPUService in the DPUCluster for the DPU that is stuck is scheduled and running.
- **ServiceChain object in the DPUCluster is not ready yet**: Verify that the ServiceChain object deployed by the DPUServiceChain in the DPUCluster for the DPU that is stuck is ready.

in-cluster DPUService does not create pod on relevant node

When an in-cluster DPUService fails to create a pod on the expected host node, the most common cause is a node labeling mismatch:

- **Verify node labels**: The node should have labels that match the `spec.serviceDaemonSet.nodeSelector` specified in the generated DPUService object.
- **Check DPUService controller logs**: If the aforementioned node labels are not present on the nodes, check the logs of the `dpuservice-controller-manager` pod for detailed error messages and insights into why those labels are not applied.

DPUDevice

The DPUDevice is a Kubernetes CRD that represents a physical DPU (Data Processing Unit) device that was discovered. The DPUDevice contain all the information required to identify and provision the DPU by the DPU Controller.

Overview

The DPUDevice resource serves as an inventory and management interface for physical DPU devices. It contains device-specific information such as serial numbers, product identifiers, BMC (Base Management Controller) details, and PCI addresses. The DPUDevice is can be created automatically through discovery processes or manually by administrators.

DPUDevice Specification

DPUDeviceSpec

The `spec` section defines the desired configuration for the DPU device:

Field	Type	Required	Description
`serialNumber`	string	Yes	The serial number of the device for inventory management
`psid`	string	No	Product Serial ID (deprecated, use status.psid)
`opn`	string	No	Ordering Part Number (deprecated, use status.opn)
`bmcIp`	string	No	IP address of the BMC for remote management
`bmcPort`	uint32	No	Port number for BMC communication (default: 443)
`numberOfPFs`	int	No	Number of Physical Functions on the device (default: 1)
`pf0Name`	string	No	Name of the first Physical Function

DPUDeviceStatus

The `status` section contains the observed state of the DPU device:

Field	Type	Description
`psid`	string	Product Serial ID discovered from the device
`serialNumber`	string	Serial number discovered from the device
`opn`	string	Ordering Part Number discovered from the device

Field	Type	Description
`bmcIp`	string	BMC IP address discovered from the device
`bmcPort`	uint32	BMC port discovered from the device
`pciAddress`	string	PCI address of the device in the host system
`pf0Mac`	string	MAC address of the first Physical Function
`conditions`	array	Array of condition objects describing device state

Conditions

The DPUDevice resource uses several condition types to track its state:

- **DpuDeviceDiscovered**: Indicates that the DPU has been discovered
- **DpuDeviceNodeAttached**: Indicates that the DPU is attached to a node
- **DpuDeviceInitialized**: Indicates that the DPU interface has been initialized
- **DpuDeviceError**: Indicates that the DPUDevice has an error
- **DpuDeviceReady**: Indicates that the DPUDevice is ready for use

Example Usage

Basic DPUDevice Creation

Determine the serial number of the DPUDevice. In zero-trust mode, serial number will be discovered from the BMC. In trusted mode, run: `lspci -vvs ${pci_address} | grep "SN"`.

```yaml
---
apiVersion: provisioning.dpu.nvidia.com/v1alpha1
kind: DPUDevice
metadata:
  name: MT25066004C7
  namespace: dpf-operator-system
spec:
  serialNumber: "MT25066004C7"
  bmcIp: "10.1.2.3"
  numberOfPFs: 1
  pf0Name: "eth0"
```

Lifecycle Management

Creation

DPUDevice resources are typically created through: * **Automatic Discovery**: * Zero-Trust: Via DPUDiscovery controller scanning IP ranges * Host-Trusted: Via dpudetector daemon on host nodes * **Manual Creation**: By administrators with known device details * **DPU Detection**: Via dpudetector daemon on host nodes

Firmware Update: - In zero-trust mode, BMC firmware will be updated to the latest version.

Updates

Most fields in DPUDevice are immutable once set. Only the following can be updated: - Labels and annotations - Status fields (managed by controllers)

Deletion

DPUDevice resources are protected by a finalizer (`provisioning.dpu.nvidia.com/dpudevice-protection`) to prevent accidental

deletion while the device is in use.

Integration with Other Resources

DPUNode

DPUDevice resources are referenced by DPUNode resources through the `dpus` field by their serial numbers:

```
apiVersion: provisioning.dpu.nvidia.com/v1alpha1
kind: DPUNode
metadata:
  name: dpu-node-001
spec:
  dpus:
  - name: MT25066004C7
  - name: MT25066004C8
```

DPU

DPU resources reference DPUDevice resources through the `dpuDeviceName` field:

```
apiVersion: provisioning.dpu.nvidia.com/v1alpha1
kind: DPU
metadata:
  name: dpu-001
spec:
  dpuDeviceName: MT25066004C7
  dpuNodeName: dpu-node-001
  # ... other fields
```

Monitoring and Troubleshooting

Checking Device Status

```
# Get all DPUDevice resources
kubectl get dpudevices -n dpf-operator-system

# Get detailed information about a specific device
kubectl describe dpudevice MT25066004C7 -n dpf-operator-system

# Check device conditions
kubectl get dpudevice MT25066004C7 -n dpf-operator-system -o jsonpath='{.status.conditions}'
```

Common Issues

- **Device Not Discovered when in Zero Trust setup**: Check if the device is reachable via BMC IP
- **Invalid Serial Number**: Ensure the serial number matches the required pattern
- **BMC Connection Issues**: Verify BMC IP and port configuration
- **PCI Address Not Found**: Check if the device is properly installed in the host

Status Conditions

Monitor the following conditions for device health:

```
# Check if device is ready
kubectl get dpudevice MT25066004C7 -n dpf-operator-system -o
jsonpath='{.status.conditions[?(@.type=="DpuDeviceReady")].status}'

# Check for errors
kubectl get dpudevice MT25066004C7 -n dpf-operator-system -o
jsonpath='{.status.conditions[?(@.type=="DpuDeviceError")]}'
```

Related Resources

- DPUNode - Node-level DPU management
- DPUDiscovery - Automatic DPU discovery
- DPU - DPU provisioning and deployment
- DPUSet - Bulk DPU management

DPUDiscovery

The DPUDiscovery is a Kubernetes CRD that enables automatic discovery of DPU (Data Processing Unit) devices within specified IP ranges in the DOCA Platform Framework (DPF). It provides a scalable and efficient way to scan network ranges for DPU BMCs and automatically create DPUDevice resources.

Overview

The DPUDiscovery resource automates the process of finding and registering DPU devices in your infrastructure. It scans specified IP ranges for DPU BMCs (Base Management Controllers) and creates corresponding DPUDevice resources for discovered devices. This eliminates the need for manual device registration and enables dynamic DPU management.

Key Features

- **Automatic Discovery**: Scans IP ranges for DPU BMCs automatically

- **Configurable Scanning**: Customizable scan intervals and worker counts
- **Scalable**: Supports parallel scanning with configurable workers
- **Redfish Integration**: Uses Redfish protocol for DPU communication
- **Status Tracking**: Provides scan status and discovered device counts
- **Resource Creation**: Automatically creates DPUDevice resources for found devices

DPUDiscovery Specification

DPUDiscoverySpec

The `spec` section defines the discovery configuration:

Field	Type	Required	Description
`ipRangeSpec`	IPRangeValidationSpec	Yes	IP range configuration for scanning
`scanInterval`	Duration	No	How often to perform scans (default: 1h)
`workers`	int	No	Number of workers for parallel scanning

IPRangeValidationSpec

Configuration for IP range validation and scanning:

Field	Type	Required	Description
`ipRange`	IPRange	Yes	IP range to scan for DPU devices

IPRange

Defines the range of IP addresses to scan:

Field	Type	Required	Description
`startIP`	string	Yes	Starting IP address of the range
`endIP`	string	Yes	Ending IP address of the range
`port`	uint32	No	BMC port to scan (default: 443)

DPUDiscoveryStatus

The `status` section contains discovery results and status:

Field	Type	Description
`lastScanTime`	Time	Timestamp of the last successful scan
`foundDPUs`	int	Number of DPU devices discovered

Example Usage

Basic DPUDiscovery

```yaml
---
apiVersion: provisioning.dpu.nvidia.com/v1alpha1
kind: DPUDiscovery
metadata:
  name: dpu-discovery-main
  namespace: dpf-operator-system
spec:
  ipRangeSpec:
    ipRange:
      startIP: "192.168.1.1"
      endIP: "192.168.1.254"
      port: 443
  scanInterval: "30m"
```

DPUDiscovery with Custom Workers

```yaml
---
apiVersion: provisioning.dpu.nvidia.com/v1alpha1
kind: DPUDiscovery
metadata:
  name: dpu-discovery-large-range
  namespace: dpf-operator-system
spec:
  ipRangeSpec:
    ipRange:
      startIP: "10.0.0.1"
      endIP: "10.0.255.254"
      port: 443
  scanInterval: "1h"
  workers: 10
```

Multiple Discovery Ranges

You can create multiple DPUDiscovery resources for different network segments:

```yaml
---
apiVersion: provisioning.dpu.nvidia.com/v1alpha1
kind: DPUDiscovery
metadata:
  name: dpu-discovery-management
  namespace: dpf-operator-system
spec:
  ipRangeSpec:
    ipRange:
      startIP: "192.168.100.1"
      endIP: "192.168.100.254"
      port: 443
  scanInterval: "15m"

---
apiVersion: provisioning.dpu.nvidia.com/v1alpha1
kind: DPUDiscovery
metadata:
  name: dpu-discovery-production
  namespace: dpf-operator-system
spec:
  ipRangeSpec:
    ipRange:
      startIP: "10.10.0.1"
      endIP: "10.10.255.254"
      port: 443
  scanInterval: "1h"
  workers: 20
```

Field Validation

IP Address Validation

- **Format**: Must be valid IPv4 addresses
- **Pattern**: `^((25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)$`
- **Restrictions**:
- Cannot be `0.0.0.0`
- Only IPv4 supported (IPv6 not allowed)
- Both startIP and endIP must be provided

Port Validation

- **Range**: 1 to 65535
- **Default**: 443

Scan Interval

- **Format**: Kubernetes Duration format (e.g., "30m", "1h", "2h30m")
- **Default**: "1h"

Workers

- **Default**: Calculated as 1 worker per 255 IPs in the range
- **Minimum**: 1 worker

- **Maximum**: No explicit limit (limited by cluster resources)

Discovery Process

Scanning Workflow

- **IP Range Calculation**: Determines the number of IPs to scan
- **Worker Allocation**: Allocates workers based on configuration or defaults
- **Parallel Scanning**: Workers scan IP ranges in parallel
- **Redfish Communication**: Uses Redfish protocol to communicate with BMCs
- **Device Detection**: Identifies DPU devices and extracts information
- **Resource Creation**: Creates DPUDevice resources for discovered devices
- **Status Update**: Updates discovery status with results

Worker Scaling

The discovery controller automatically calculates the optimal number of workers:

```
const ipPerWorker = 255
workers = int((end-start)/uint32(ipPerWorker)) + 1
if workers < 1 {
    workers = 1
}
```

This ensures efficient scanning without overwhelming the cluster resources.

Configuration Requirements

DPFOperatorConfig

DPUDiscovery requires specific configuration in the DPFOperatorConfig:

```
apiVersion: provisioning.dpu.nvidia.com/v1alpha1
kind: DPFOperatorConfig
metadata:
  name: dpf-operator-config
  namespace: dpf-operator-system
spec:
  provisioningController:
    installInterface:
      installViaRedfish:
        enabled: true
        skipDPUNodeDiscovery: true  # Set to false to create DPUNode
by DPUDiscovery process
```

Redfish Configuration

The discovery process uses Redfish protocol for BMC communication. Ensure:

- **Redfish is enabled** in DPFOperatorConfig
- **BMC credentials** are properly configured
- **Network connectivity** to BMC IPs is available
- **Firewall rules** allow Redfish traffic (typically port 443)

Monitoring and Troubleshooting

Checking Discovery Status

```
# Get all DPUDiscovery resources
kubectl get dpudiscoveries -n dpf-operator-system

# Get detailed information about discovery
kubectl describe dpudiscovery dpu-discovery-main -n dpf-operator-system

# Check discovery status
kubectl get dpudiscovery dpu-discovery-main -n dpf-operator-system -o jsonpath='{.status}'
```

Monitoring Scan Progress

```
# Check last scan time
kubectl get dpudiscovery dpu-discovery-main -n dpf-operator-system -o jsonpath='{.status.lastScanTime}'

# Check number of found DPUs
kubectl get dpudiscovery dpu-discovery-main -n dpf-operator-system -o jsonpath='{.status.foundDPUs}'

# Watch discovery status
kubectl get dpudiscoveries -n dpf-operator-system -w
```

Common Issues

- **No DPUs Found**:
- Verify IP range configuration
- Check network connectivity to BMCs
- Ensure Redfish is enabled and configured

- **Scan Failures**:
- Check DPFOperatorConfig settings
- Verify Redfish credentials (`bmc-shared-password` secret)
- Review controller logs

Controller Logs

```
# Check discovery controller logs
kubectl logs -n dpf-operator-system deployment/dpf-operator-controller-manager | grep -i discovery
```

Integration with Other Resources

DPUDevice Creation

DPUDiscovery automatically creates DPUDevice resources for discovered devices:

```
apiVersion: provisioning.dpu.nvidia.com/v1alpha1
kind: DPUDevice
metadata:
  name: <discovered-serial>
  namespace: dpf-operator-system
spec:
  serialNumber: "<discovered-serial>"
  # ... other discovered fields
```

DPUNode Integration

When `skipDpuNodeDiscovery` is false, discovery can also create DPUNode resources for discovered devices.

Related Resources

- DPUDevice - Individual DPU device management
- DPUNode - Node-level DPU management
- DPFOperatorConfig - Operator configuration
- DPU - DPU provisioning and deployment

DPUFlavor

Overview

DPUFlavor is a Kubernetes Custom Resource Definition (CRD) that defines configuration templates for DPU system-level settings. It serves as a blueprint that specifies how DPUs should be configured during provisioning, including kernel parameters, firmware settings, OVS configuration, network interfaces, and resource allocation.

API Version

- **API Group**: `provisioning.dpu.nvidia.com`
- **API Version**: `v1alpha1`
- **Kind**: `DPUFlavor`

Key Features

- **Immutable Configuration**: Once created, the DPUFlavor spec cannot be modified to ensure consistency across DPU deployments
- **Comprehensive System Configuration**: Covers all aspects of DPU system configuration from boot parameters to runtime settings

- **Resource Management**: Defines resource requirements and allocation policies
- **Multiple DPU Modes**: Supports both standard DPU mode and zero-trust mode
- **Template Reusability**: Can be applied to multiple DPUs for consistent configuration

API Reference

DPUFlavorSpec

Field	Type	Description
`grub`	DPUFlavorGrub	All the parameters will be set in `GRUB_CMDLINE_LINUX` grub configuration
`sysctl`	DPUFlavorSysctl	Kernel sysctl parameters which will be stored in `/etc/sysctl.d/99-dpf.conf`
`nvconfig`	[]NVConfig	The device configuratior which will be applied by `mlxconfig`
`ovs`	DPUFlavorOVS	Open vSwitch configuration which will be executed via systemc service
`bfcfgParameters`	[]string	Parameters for bf.cfg file
`configFiles`	[]ConfigFile	Custom configuration files. Users can use this configuration to overwrite files in the DPU file system or add content to existing files
`containerdConfig`	ContainerdConfig	ContainerdConfig contains the configuration for containerd

Field	Type	Description
`dpuResources`	ResourceList	Minimum resources needed for BFB installation
`systemReservedResources`	ResourceList	Resources reserved for system use
`dpuMode`	DpuModeType	DPU operation mode (default: `dpu`)
`hostNetworkInterfaceConfigs`	[]NetworkInterfaceConfig	Host-side network interface configuration

DPUFlavorGrub

Field	Type	Description
`kernelParameters`	[]string	Kernel boot parameters to be set in grub configuration

DPUFlavorSysctl

Field	Type	Description
`parameters`	[]string	Sysctl parameters to be applied

NVConfig

Field	Type	Description
`device`	*string	Target device (use `"*"` for all devices)
`parameters`	[]string	Firmware parameters to set
`hostPowerCycleRequired`	*bool	Whether host power cycle is needed after applying config(Deprecated)

IB Mode to Ethernet Mode Configuration

Example for single port DPU:

```
nvconfig:
  - device: '*'
    parameters:
      - LINK_TYPE_P1=ETH
```

Example for dual port DPU:

```
nvconfig:
  - device: '*'
    parameters:
      - LINK_TYPE_P1=ETH
      - LINK_TYPE_P2=ETH
```

DPUFlavorOVS

Field	Type	Description
`rawConfigScript`	string	Raw OVS configuration script

ConfigFile

Field	Type	Description
`path`	string	File path on the DPU
`operation`	DPUFlavorFileOp	File operation type (`override` or `append`)
`raw`	string	File content

Field	Type	Description
`permissions`	string	File permissions (e.g., `"0644"`)

ContainerdConfig

Field	Type	Description
`registryEndpoint`	string	Container registry endpoint

NetworkInterfaceConfig

Field	Type	Description
`mtu`	*int32	MTU value (1280-9216)
`dhcp`	*bool	Enable DHCP configuration
`portNumber`	int32	Port identifier (0 or 1)
`nvconfig`	*NVConfig	Port-specific NVConfig settings

Enumerations

DpuModeType

- `dpu`: Standard DPU mode
- `zero-trust`: Zero-trust security mode

DPUFlavorFileOp

- `override`: Replace file content entirely

- `append`: Append to existing file content

Resource Management

DPU Resources

The `dpuResources` field specifies the minimum resources required for a BFB with this flavor to be installed on a DPU. Using this field, the controller can understand if that flavor can be installed on a particular DPU. It should be set to the total amount of resources the system needs + the resources that should be made available for DPUServices to consume:

```
dpuResources:
  cpu: 16
  memory: 16Gi
  nvidia.com/sf: 20
```

System Reserved Resources

The `systemReservedResources` field indicates resources consumed by the system (OS, OVS, DPF system etc) and are not made available for DPUServices to consume. DPUServices can consume the difference between `dpuResources` and `systemReservedResources`. This field must not be specified if `dpuResources` are not specified.:

```
systemReservedResources:
  cpu: 4
  memory: 4Gi
  nvidia.com/sf: 4
```

The difference between `dpuResources` and `systemReservedResources` is available for DPUServices.

Examples

HBN-OVN DPUFlavor

```yaml
apiVersion: provisioning.dpu.nvidia.com/v1alpha1
kind: DPUFlavor
metadata:
  name: hbn-ovn
  namespace: dpf-operator-system
spec:
  bfcfgParameters:
  - UPDATE_ATF_UEFI=yes
  - UPDATE_DPU_OS=yes
  - WITH_NIC_FW_UPDATE=yes
  configFiles:
  - operation: override
    path: /etc/mellanox/mlnx-bf.conf
    permissions: "0644"
    raw: |
      ALLOW_SHARED_RQ="no"
      IPSEC_FULL_OFFLOAD="no"
      ENABLE_ESWITCH_MULTIPORT="yes"
  - operation: override
    path: /etc/mellanox/mlnx-ovs.conf
    permissions: "0644"
    raw: |
      CREATE_OVS_BRIDGES="no"
      OVS_DOCA="yes"
  - operation: override
    path: /etc/mellanox/mlnx-sf.conf
    permissions: "0644"
    raw: ""
  dpuMode: dpu
  grub:
    kernelParameters:
    - console=hvc0
    - console=ttyAMA0
    - earlycon=pl011,0x13010000
    - fixrttc
```

```
    - net.ifnames=0
    - biosdevname=0
    - iommu.passthrough=1
    - cgroup_no_v1=net_prio,net_cls
    - hugepagesz=2048kB
    - hugepages=3072
  hostNetworkInterfaceConfigs:
  - dhcp: true
    mtu: 1500
    portNumber: 0
  nvconfig:
  - device: '*'
    parameters:
    - PF_BAR2_ENABLE=0
    - PER_PF_NUM_SF=1
    - PF_TOTAL_SF=20
    - PF_SF_BAR_SIZE=10
    - NUM_PF_MSIX_VALID=0
    - PF_NUM_PF_MSIX_VALID=1
    - PF_NUM_PF_MSIX=228
    - INTERNAL_CPU_MODEL=1
    - INTERNAL_CPU_OFFLOAD_ENGINE=0
    - SRIOV_EN=1
    - NUM_OF_VFS=46
    - LAG_RESOURCE_ALLOCATION=1
  ovs:
    rawConfigScript: |
      _ovs-vsctl() {
        ovs-vsctl --no-wait --timeout 15 "$@"
      }

      _ovs-vsctl set Open_vSwitch . other_config:doca-init=true
      _ovs-vsctl set Open_vSwitch . other_config:dpdk-max-
memzones=50000
      _ovs-vsctl set Open_vSwitch . other_config:hw-offload=true
```

```
      _ovs-vsctl set Open_vSwitch . other_config:pmd-quiet-
idle=true
      _ovs-vsctl set Open_vSwitch . other_config:max-idle=20000
      _ovs-vsctl set Open_vSwitch . other_config:max-
revalidator=5000
      _ovs-vsctl set Open_vSwitch . other_config:ctl-pipe-
size=1024
      _ovs-vsctl --if-exists del-br ovsbr1
      _ovs-vsctl --if-exists del-br ovsbr2
      _ovs-vsctl --may-exist add-br br-sfc
      _ovs-vsctl set bridge br-sfc datapath_type=netdev
      _ovs-vsctl set bridge br-sfc fail_mode=secure
      _ovs-vsctl --may-exist add-br br-hbn
      _ovs-vsctl set bridge br-hbn datapath_type=netdev
      _ovs-vsctl set bridge br-hbn fail_mode=secure
      _ovs-vsctl --may-exist add-port br-sfc p0
      _ovs-vsctl set Interface p0 type=dpdk
      _ovs-vsctl set Interface p0 mtu_request=9216
      _ovs-vsctl set Port p0 external_ids:dpf-type=physical

      _ovs-vsctl set Open_vSwitch . external-ids:ovn-bridge-
datapath-type=netdev
      _ovs-vsctl --may-exist add-br br-ovn
      _ovs-vsctl set bridge br-ovn datapath_type=netdev
      _ovs-vsctl br-set-external-id br-ovn bridge-id br-ovn
      _ovs-vsctl br-set-external-id br-ovn bridge-uplink
puplinkbrovntobrsfc
      _ovs-vsctl --may-exist add-port br-ovn pf0hpf
      _ovs-vsctl set Interface pf0hpf type=dpdk
      _ovs-vsctl set Interface pf0hpf mtu_request=9216
```

Best Practices

1. **Resource Planning**: Always specify `dpuResources` and `systemReservedResources` to ensure proper resource allocation

2. **Immutability**: Plan your configuration carefully as DPUFlavor specs cannot be modified after creation

3. **Testing**: Test DPUFlavor configurations in development environments before production deployment

4. **Documentation**: Document custom configurations and their purposes for team understanding

5. **IB Mode Conversion**: For DPUs initially in InfiniBand (IB) mode, always include `LINK_TYPE_P1=2` in nvconfig parameters to convert to Ethernet mode. For dual port DPUs, also add `LINK_TYPE_P2=2`

DPUNode

The DPUNode is a Kubernetes CRD that represents a physical host node containing one or more DPU (Data Processing Unit) devices in the DOCA Platform Framework (DPF). It provides node-level management capabilities for DPU provisioning, reboot control, and integration with Kubernetes clusters.

Overview

The DPUNode resource serves as a bridge between physical host nodes and DPU devices, enabling centralized management of DPU provisioning and host operations. It defines how DPUs should be provisioned on a specific node and how the host should be managed during DPU operations.

Key Features

- **Node-Level Management**: Manages DPU operations at the host node level
- **Reboot Control**: Configurable host reboot methods (gNOI, external, script)
- **DMS Integration**: Integration with Device Management Service (DMS)
- **DPU Association**: Links multiple DPU devices to a single node
- **Kubernetes Integration**: Optional integration with Kubernetes Node objects

DPUNode Specification

DPUNodeSpec

The `spec` section defines the desired configuration for the DPU node:

Field	Type	Required	Description
`nodeRebootMethod`	NodeRebootMethod	No	Method for rebooting the host (default: gNOI)
`nodeDMSAddress`	DMSAddress	No	IP and port for DMS communication
`dpus`	[]DPURef	No	List of DPU devices attached to this node

NodeRebootMethod

Defines how the host should be rebooted during DPU operations:

Field	Type	Required	Description
`gNOI`	GNOI	No	Use DPU's DMS interface to reboot the host
`external`	External	No	Reboot via external means (not controlled by DPU controller)
`script`	Script	No	Reboot by executing a custom script

DMSAddress

Configuration for Device Management Service communication:

Field	Type	Required	Description
`ip`	string	Yes	IP address in IPv4 format
`port`	uint16	Yes	Port number (minimum: 1)

DPURef

Reference to a DPU device:

Field	Type	Required	Description
`name`	string	Yes	Name of the DPU device

DPUNodeStatus

The `status` section contains the observed state of the DPU node:

Field	Type	Description
`conditions`	array	Array of condition objects describing node state
`dpuInstallInterface`	string	Interface used for DPU installation (gNOI or redfish)
`kubeNodeRef`	string	Name of the Kubernetes Node object (immutable)
`rebootInProgress`	bool	Indicates if the node is currently rebooting

Conditions

The DPUNode resource uses several condition types to track its state:

- **Ready**: The DPU node is ready for operations
- **InvalidDPUDetails**: The DPU details provided are invalid
- **DPUNodeRebootInProgress**: The DPUNode is in the process of rebooting
- **DPUUpdateInProgress**: The DPU is being updated
- **NeedHostAgentUpgrade**: The host agent needs to be upgraded
- **OOBBridgeConfigured**: The out-of-band bridge (br-dpu) is configured

- **RshimAvailable**: The rshim interface is available

Example Usage

Basic DPUNode with gNOI Reboot

```yaml
---
apiVersion: provisioning.dpu.nvidia.com/v1alpha1
kind: DPUNode
metadata:
  name: dpu-node-001
  namespace: dpf-operator-system
spec:
  nodeRebootMethod:
    gNOI: {}
  nodeDMSAddress:
    ip: "192.168.1.100"
    port: 443
  dpus:
  - name: dpu-device-001
  - name: dpu-device-002
```

DPUNode with External Reboot

```yaml
---
apiVersion: provisioning.dpu.nvidia.com/v1alpha1
kind: DPUNode
metadata:
  name: dpu-node-002
  namespace: dpf-operator-system
spec:
  nodeRebootMethod:
    external: {}
  dpus:
  - name: dpu-device-003
```

DPUNode with Custom Script Reboot

```yaml
---
apiVersion: provisioning.dpu.nvidia.com/v1alpha1
kind: DPUNode
metadata:
  name: dpu-node-003
  namespace: dpf-operator-system
spec:
  nodeRebootMethod:
    script:
      name: custom-reboot-script
  dpus:
  - name: dpu-device-004
```

Custom Reboot Script ConfigMap

```yaml
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: custom-reboot-script
  namespace: dpf-operator-system
data:
  pod-template: |
    apiVersion: v1
    kind: Pod
    metadata:
      name: custom-reboot-pod
      namespace: dpf-operator-system
    spec:
      containers:
      - name: reboot-container
        image: ubuntu:20.04
        command: ["/bin/bash"]
        args:
        - -c
        - |
          echo "Performing custom reboot procedure..."
          # Add your custom reboot logic here
          # For example: IPMI commands, SSH to BMC, etc.
          sleep 10
          exit 0
      restartPolicy: Never
```

Reboot Methods

gNOI (Default)

Uses the DPU's Device Management Service interface to reboot the host. This is the recommended method for most deployments.

Advantages: * Integrated with DPU management * Reliable and consistent * No external dependencies

Requirements: * DMS must be accessible * Valid DMS address configuration

External

Reboots the host via external means not controlled by the DPU controller. This method requires manual intervention or external automation.

Use Cases: * Custom power management systems * IPMI-based reboots * Cloud provider APIs

Requirements: * External reboot mechanism must be available * Manual intervention may be required

Script

Executes a custom script to reboot the host. The script is defined in a ConfigMap and executed as a Kubernetes Job.

Use Cases: * Custom reboot procedures * Integration with existing automation * Complex reboot workflows

Requirements: * ConfigMap with pod template * Script must exit successfully * Proper RBAC permissions

Integration with Kubernetes

Node Association

DPUNode can optionally be associated with a Kubernetes Node object:

```
status:
  kubeNodeRef: "worker-node-001"
```

This association enables: * Node-level operations (draining, tainting) * Integration with Kubernetes scheduling * Resource management alignment

Annotations

DPUNode supports the following annotation for external reboot requirements:

```
metadata:
  annotations:
    provisioning.dpu.nvidia.com/dpunode-external-reboot-required: "true"
```

Lifecycle Management

Creation

DPUNode resources are typically created: * **Manually**: By administrators for known nodes * **Automatically**: Via discovery processes * **Via DPUSet**: As part of bulk node management

Updates

Most fields in DPUNode can be updated, but some restrictions apply: * `kubeNodeRef` is immutable once set * `dpus` list can be modified to add/remove devices

Deletion

DPUNode resources are protected by a finalizer (`provisioning.dpu.nvidia.com/dpunode-protection`) to prevent deletion while DPUs are in use.

Monitoring and Troubleshooting

Checking Node Status

```
# Get all DPUNode resources
kubectl get dpunodes -n dpf-operator-system

# Get detailed information about a specific node
kubectl describe dpunode dpu-node-001 -n dpf-operator-system

# Check node conditions
kubectl get dpunode dpu-node-001 -n dpf-operator-system -o jsonpath='{.status.conditions}'
```

Common Issues

- **Invalid DMS Address**: Verify IP and port configuration
- **DPU Not Found**: Ensure referenced DPUDevice resources exist
- **Reboot Failures**: Check reboot method configuration and permissions
- **Script Execution Errors**: Verify ConfigMap and script syntax

Status Monitoring

```
# Check if node is ready
kubectl get dpunode dpu-node-001 -n dpf-operator-system -o
jsonpath='{.status.conditions[?(@.type=="Ready")].status}'

# Check reboot status
kubectl get dpunode dpu-node-001 -n dpf-operator-system -o
jsonpath='{.status.rebootInProgress}'

# Check install interface
kubectl get dpunode dpu-node-001 -n dpf-operator-system -o
jsonpath='{.status.dpuInstallInterface}'
```

Related Resources

- DPUDevice - Individual DPU device management
- DPU - DPU provisioning and deployment
- DPUSet - Bulk DPU and node management
- DPUDiscovery - Automatic DPU discovery

DPUService

The `DPUService` CRD provides an API for DPF users to deploy applications on DPU nodes using Helm charts.

Example

The following is an example of a `DPUService` YAML for the blueman service:

```yaml
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUService
metadata:
  name: doca-blueman-service
  namespace: dpf-operator-system
spec:
  helmChart:
    source:
      repoURL: https://helm.ngc.nvidia.com/nvidia/doca
      version: 1.0.8
      chart: doca-blueman
  serviceDaemonSet:
    updateStrategy:
      type: RollingUpdate
      rollingUpdate:
        maxUnavailable: 2
    labels:
      dpuservice.dpu.nvidia.com/name: doca-blueman-service
    annotations:
      dpuservice.dpu.nvidia.com/name: doca-blueman-service
```

In the above example:

- A `DPUService` named `doca-blueman-service` is created in the `dpf-operator-system` namespace.
- A HelmChart located at `https://helm.ngc.nvidia.com/nvidia/doca` with version `1.0.5` and chart `doca-blueman` is used to deploy the `DPUService`.
- A `DaemonSet` is deployed on target DPU nodes. We specify the update strategy, labels, and annotations for the `DaemonSet`.

Managing Lifecycle of DPUServices

A DPUDeployment resource is the recommended way to deploy and manage `DPUServices` in DPF. The `DPUDeployment` resource is a higher level abstraction that allows to manage the lifecycle of `DPUServices` in the DPU cluster. It provides a way to deploy, upgrade, validate dependencies and resources requirements for `DPUServices`.

Writing a DPUService Spec

Helm Chart Configuration

The `spec.helmChart` field is used to specify the Helm chart that will be used to deploy the `DPUService`. The `spec.helmChart.source` field is used to specify the Helm chart repository URL, version, and chart name.

```
spec:
  helmChart:
    source:
      repoURL: https://helm.ngc.nvidia.com/nvidia/doca
      version: 1.0.8
      chart: doca-blueman
```

The following fields are used to specify the Helm chart:

- `repoURL` - the URL of the Helm chart repository. `OCI` and `HTTP` repositories are supported.
- `version` - the version of the Helm chart to be used.
- `chart` - the name of the Helm chart.

It is possible to specify the path to the Helm chart in the repository with the `spec.helmChart.source.path` field.

Optionally, the `spec.helmChart.source.releaseName` field can be used to specify the release name of the released Helm chart.

Service DaemonSet Values

The `spec.serviceDaemonSet` field is used to specify the configuration of the DaemonSet. It represents Helm values to be used during the release of the Helm chart. These values take precedence over the values defined in the Helm chart.

```
spec:
  serviceDaemonSet:
    updateStrategy:
      type: RollingUpdate
      rollingUpdate:
        maxUnavailable: 2
    nodeSelector:
      nodeSelectorTerms:
        - matchExpressions:
          - key: "bfb"
            operator: In
            values: ["dk"]
    labels:
      "some-label": "some-value"
    annotations:
      "some-annotation": "some-value"
    resources:
      resources:
        memory: 6Gi
        nvidia.com/bf_sf: 3
```

The following fields are used to specify the DaemonSet configuration:

- `updateStrategy` - the update strategy for the DaemonSet. This should be a valid Kubernetes `DaemonSetUpdateStrategy` object.
- `nodeSelector` - the node selector for the DaemonSet. This field is used to select the nodes where the DaemonSet's Pods will be deployed.

- `labels` - the labels to be applied to the DaemonSet's Pods.
- `annotations` - the annotations to be applied to the DaemonSet's Pods.
- `resources` - the resource requests and limits for the DaemonSet's Pods.

For more information on how to write a Helm chart for a `DPUService`, see the DPUService Development Guide.

Deploy a DPUService in the Host Cluster

It is possible to deploy a `DPUService` in the host cluster, i.e. the cluster where the DPF operator is running. This can be done by setting the `spec.deployInCluster` field in the `DPUService` manifest to `true`.

DPUServiceInterface dependencies

A `DPUService` can have dependencies on `DPUServiceInterface` resources. These resources represent interfaces that are required by the `DPUService` to function properly.

The `DPUServiceInterface` resources must exist for the `DPUService` to be deployed successfully. If the `DPUServiceInterface` resources are not found in the cluster, the `DPUService` will not be deployed and a condition will be set to `false` with a reason on the `status` field of the `DPUService` resource.

To define a dependency `DPUServiceInterface` resources, the `spec.interfaces` field of the `DPUService` resource must be set to the names of the `DPUServiceInterfaces`.

```
spec:
  interfaces:
    - p0-sif
    - p1-if
    - app-if
```

ConfigPorts

A `DPUService` workload can expose multiple ports to the host cluster. The `configPorts` field is used to specify the ports that should be exposed from the DPU nodes to the host cluster.

```
spec:
  configPorts:
    serviceType: NodePort
    ports:
      - name: port1
        protocol: TCP
        nodePort: 30001
      - name: port2
        protocol: TCP
        nodePort: 30000
```

The `spec.configPorts.serviceType` field defines the type of service to be created for the `DPUService`. It can be `NodePort`, `ClusterIP`, or `None` with `NodePort` being the default value.

The `spec.configPorts.ports` field is used to specify the ports that should be exposed from the DPU nodes to the host cluster. Each port should have the following fields:

- `name` - the name of the port. This field is a unique identifier and must reference a port defined in the Helm chart.

- `port` - the port that will be exposed on the host cluster for use by in-cluster workloads.
- `protocol` - the protocol to be used for the port. It can be `TCP` or `UDP`.
- `nodePort` - the external port to be exposed on the host cluster. This field is optional and is used only when the `spec.configPorts.serviceType` field is set to `NodePort`.

Upon reconciliation, the `DPUService` controller will retrieve the list of services that exist in the DPU cluster for this `DPUService`. It will retrieve the ports that are matching the ports defined in the `spec.configPorts.ports` field. It uses the `name` field to match the ports. If the ports are found, the `nodePort` values will be collected. The controller will then make the collected `nodePorts` accessible from the host cluster by creating a new `Service` resource.

For more information on how to write a Helm chart for a `DPUService`, in order to expose ports, see the DPUService development guide.

DPUService Update

The `DPUService` resource can be updated by changing the fields in the `spec` section of the resource. The controller will detect the changes and update the `DPUService` accordingly. The `DPUService` will not be recreated during the update process.

Working with DPUServices

Waiting for Ready

When a `DPUService` is created, it may take some time for the workload to be ready.

It is possible to wait for a `DPUService` to be ready by using the `kubectl wait` command:

```
$ kubectl wait --for=condition=Ready dpuservice/<dpuservice-name>
-n <namespace>
```

Suspending and Resuming

Suspending a DPUService

The need to temporarily suspend a `DPUService` may arise when performing maintenance or troubleshooting. The `DPUService` can be suspended by setting the `spec.paused` field to `true`.

```
spec:
  paused: true
```

Or by using the `kubectl patch` command:

```
$ kubectl patch dpuservice <dpuservice-name> \
-n <namespace> \
--field-manager=dpf-client-side-apply \
--type='json' \
-p='[{"op": "replace", "path": "/spec/paused", "value": true}]'
```

Note: When a `DPUService` is suspended, the controller will not reconcile the resource until the `spec.paused` field is set to `false`. This means that the `DPUService` can neither be updated nor deleted while it is suspended.

Resuming a DPUService

To resume a suspended `DPUService`, set the `spec.paused` field to `false`.

```
spec:
  paused: false
```

Or by using the `kubectl patch` command:

```
$ kubectl patch dpuservice <dpuservice-name> \
-n <namespace> \
--field-manager=dpf-client-side-apply \
--type='json' \
-p='[{"op": "replace", "path": "/spec/paused", "value": false}]'
```

Debugging DPUServices

There are several ways to debug `DPUServices` in DPF. The recommended way is to use the dpfctl command line tool to gather information about the `DPUServices` in the DPU cluster.

```
$ dpfctl describe dpuservices --show-conditions=all
NAME                                                    NAMESPACE
STATUS       REASON   SINCE  MESSAGE
DPFOperatorConfig/dpfoperatorconfig                     dpf-
operator-system
              Ready
True         Success  28h
              ImagePullSecretsReconciled
True         Success  5d1h
              PreUpgradeValidationReady
True         Success  5d1h
              SystemComponentsReady
True         Success  28h
              SystemComponentsReconciled
True         Success  5d1h
  DPUServiceCredentialRequests
    DPUServiceCredentialRequest/servicesetcontroller  dpf-
operator-system
                  Ready
True         Success  5d1h
                  SecretReconciled
True         Success  5d1h
                  ServiceAccountReconciled
True         Success  5d1h
  DPUServices
    4 DPUServices...                                    dpf-
operator-system  Ready: True  Success  3d7h   See ovn-central-
9558p, ovn-controller-v5bkr, vpc-ovn-controller-7sbp6, vpc-ovn-
node-r84zn
  System Components                                     dpf-
operator-system
    DPUServices
      8 DPUServices...                                  dpf-
operator-system  Ready: True  Success  5d1h   See flannel,
multus, nvidia-k8s-ipam, ovs-cni, servicechainset-controller,
```

```
servicechainset-rbac-and-crds, sfc-controller, sriov-device-
plugin
```

Configuring Critical DPUService for DPU Readiness

Overview

In DPF architecture, host worker nodes and DPUs run in separate Kubernetes clusters. Some host workloads rely on corresponding DPU pods (DPUServices). If a critical DPUService is missing or not running on the DPU, the host should stop accepting new workloads to avoid instability.

To enforce this, DPF introduces a mechanism that taints the host node when a critical DPUService is not running.

How to Mark a DPUService as Critical

To mark a DPUService as critical, add the following label:
`svc.dpu.nvidia.com/critical: ""`

Example:
```
kubectl label dpuservice sriov-device-plugin
svc.dpu.nvidia.com/critical="" -n dpf-operator-system
```

To remove the critical label, use:
```
kubectl label dpuservice sriov-device-plugin
svc.dpu.nvidia.com/critical- -n dpf-operator-system
```

Example Manifest Snippet

```
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUService
metadata:
  name: sriov-device-plugin
  namespace: dpf-operator-system
  labels:
    svc.dpu.nvidia.com/critical: ""
```

Taint Behavior

When a pod of a critical DPUService is not in a Running state on the DPU, the corresponding host node is tainted:

```
spec:
  taints:
  - effect: NoSchedule
    key: dpu.nvidia.com/dpu-ready
```

When the pod becomes `Ready` again, the taint is removed.

Recovery Process

Once the failed critical DPUService pod is running again, the following occurs:

- The controller detects this change during its next reconciliation loop.
- The `dpu.nvidia.com/dpu-ready` taint is removed.
- Host node resumes scheduling workloads.

Troubleshooting

To check controller logs, use:

```
kubectl logs dpuservice-controller-manager-<pod> -n dpf-operator-system | grep dpuready
```

Example:

```
I0619 09:39:35.905116 dpuready_controller.go:198] Adding taint to node Node="host-01"
I0619 09:46:01.170252 dpuready_controller.go:216] Removing taint from node Node="host-01"
```

Limitations

- Tainting is based on a 10-minute periodic reconciliation.
- Only pods for critical DPUServices labeled as svc.dpu.nvidia.com/critical are considered.
- Taint is only removed once the pod is Ready.
- Feature currently does not trigger instant reconciliation (planned for future releases).

Quick Reference

Item	Description
Label key	`svc.dpu.nvidia.com/critical`
Taint key	`dpu.nvidia.com/dpu-ready`

Item	Description
Taint effect	`NoSchedule`
Taint behavior	Applied if a critical DPUService pod is not running on the DPU
Recovery	Taint is removed once the pod is running and ready
Check logs with	`kubectl logs | grep dpuready`
Reconciliation interval	Every 10 minutes (will become event-driven in next release)

Final Notes

This feature is essential for ensuring host-side stability when critical DPU infrastructure is missing or unhealthy. Proper labeling and understanding of the DPU-host relationship are required for effective use of this functionality.

Dividing the cluster into several zones (DEPRECATED - Use DPUDeployment instead)

For a better control of maintenance and down-time, the cluster can be logically divided into several "zones". Each zone can have its own set of DPU services, that can be upgraded individually, affecting only the specific zone. The creation of zones for DPU services is done by adding labels on the nodes in the DPU cluster and then using them with the DPU service YAML:

Create a specific DPU Set for worker nodes labeled as "e2e.servers/dk=true", by adding the "cluster -> nodeLabels" section, assign their DPUs the label "bfb=dk" (on the DPU cluster).

```
---
apiVersion: provisioning.dpu.nvidia.com/v1alpha1
kind: DPUSet
metadata:
  name: dpuset-dk
  namespace: dpf-operator-system
spec:
  dpuNodeSelector:
  matchLabels:
    e2e.servers/dk: "true"
  strategy:
  rollingUpdate:
    maxUnavailable: "10%"
  type: RollingUpdate
  dpuTemplate:
  spec:
    dpuFlavor: dpf-provisioning-hbn-ovn
    bfb:
    name: bf-bundle-dk-ga
    nodeEffect:
    taint:
      key: "dpu"
      value: "provisioning"
      effect: NoSchedule
    cluster:
    nodeLabels:
      bfb: "dk"
```

Then use the assigned label to create an HBN DPU Service for these specific nodes (under the "nodeSelector" section):

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUService
metadata:
  name: doca-hbn-dk
  namespace: dpf-operator-system
spec:
  serviceID: doca-hbn
  interfaces:
  - p0-sf-dk
  - p1-sf-dk
  - app-sf-dk
  serviceDaemonSet:
  nodeSelector:
    nodeSelectorTerms:
    - matchExpressions:
    - key: "bfb"
        operator: In
      values: ["dk"]
  annotations:
    k8s.v1.cni.cncf.io/networks: |-
    [
    {"name": "iprequest", "interface": "ip_lo", "cni-args": {"poolNames": ["loopback"],
"poolType": "cidrpool"}},
    {"name": "iprequest", "interface": "ip_pf2dpu3", "cni-args": {"poolNames": ["pool1"],
"poolType": "cidrpool", "allocateDefaultGateway": true}}
        ]
  helmChart:
  source:
    repoURL: https://helm.ngc.nvidia.com/nvidia/doca
    version: 1.0.1
    chart: doca-hbn
  values:
    image:
    repository: nvcr.io/nvidia/doca/doca_hbn
```

```
    tag: 2.4.1-doca2.9.1
    resources:
    memory: 6Gi
    nvidia.com/bf_sf: 3
    configuration:
    perDPUValuesYAML: |
    - hostnamePattern: "*"
      values:
        bgp_autonomous_system: 65111
        bgp_peer_group: hbn
    startupYAMLJ2: |
      - header:
        model: BLUEFIELD
        nvue-api-version: nvue_v1
        rev-id: 1.0
        version: HBN 2.4.0
      - set:
        interface:
        lo:
          ip:
          address:
            {{ ipaddresses.ip_lo.ip }}/32: {}
          type: loopback
        p0_if,p1_if:
          type: swp
          link:
          mtu: 9000
        pf2dpu3_if:
          ip:
          address:
            {{ ipaddresses.ip_pf2dpu3.cidr }}: {}
          type: swp
          link:
          mtu: 9000
        router:
        bgp:
```

```
        autonomous-system: {{ config.bgp_autonomous_system }}
        enable: on
        graceful-restart:
        mode: full
        router-id: {{ ipaddresses.ip_lo.ip }}
      vrf:
      default:
        router:
        bgp:
          address-family:
          ipv4-unicast:
            enable: on
            redistribute:
            connected:
              enable: on
          ipv6-unicast:
            enable: on
            redistribute:
            connected:
              enable: on
          enable: on
          neighbor:
          p0_if:
            peer-group: {{ config.bgp_peer_group }}
            type: unnumbered
          p1_if:
            peer-group: {{ config.bgp_peer_group }}
            type: unnumbered
          path-selection:
          multipath:
            aspath-ignore: on
          peer-group:
          {{ config.bgp_peer_group }}:
            remote-as: external
```

You can do the same for the additional required YAMLs (interfaces and chains):

```yaml
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceInterface
metadata:
  name: app-sf-dk
  namespace: dpf-operator-system
spec:
  template:
  spec:
    nodeSelector:
    matchLabels:
      bfb: "dk"
    template:
    metadata:
      labels:
      svc.dpu.nvidia.com/interface: "app_sf"
      svc.dpu.nvidia.com/service: doca-hbn
    spec:
      interfaceType: service
      service:
      serviceID: doca-hbn
      network: mybrhbn
      interfaceName: pf2dpu3_if
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceInterface
metadata:
  name: p0-sf-dk
  namespace: dpf-operator-system
spec:
  template:
  spec:
    nodeSelector:
    matchLabels:
      bfb: "dk"
```

```
    template:
    metadata:
      labels:
      svc.dpu.nvidia.com/interface: "p0_sf"
      svc.dpu.nvidia.com/service: doca-hbn
    spec:
      interfaceType: service
      service:
      serviceID: doca-hbn
      network: mybrhbn
      interfaceName: p0_if
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceInterface
metadata:
  name: p1-sf-dk
  namespace: dpf-operator-system
spec:
  template:
  spec:
    nodeSelector:
    matchLabels:
      bfb: "dk"
    template:
    metadata:
      labels:
      svc.dpu.nvidia.com/interface: "p1_sf"
      svc.dpu.nvidia.com/service: doca-hbn
    spec:
      interfaceType: service
      service:
      serviceID: doca-hbn
      network: mybrhbn
      interfaceName: p1_if
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
```

```
kind: DPUServiceChain
metadata:
  name: hbn-to-fabric-dk
  namespace: dpf-operator-system
spec:
  template:
  spec:
    nodeSelector:
    matchLabels:
      bfb: "dk"
    template:
    spec:
      switches:
      - ports:
        - serviceInterface:
          matchLabels:
          uplink: p0
        - serviceInterface:
          matchLabels:
          svc.dpu.nvidia.com/service: doca-hbn
          svc.dpu.nvidia.com/interface: "p0_sf"
      - ports:
        - serviceInterface:
          matchLabels:
          uplink: p1
        - serviceInterface:
          matchLabels:
          svc.dpu.nvidia.com/service: doca-hbn
          svc.dpu.nvidia.com/interface: "p1_sf"
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceChain
metadata:
  name: ovn-to-hbn-dk
  namespace: dpf-operator-system
spec:
```

```
  template:
  spec:
    nodeSelector:
    matchLabels:
      bfb: "dk"
    template:
    spec:
      switches:
      - ports:
        - serviceInterface:
          matchLabels:
          svc.dpu.nvidia.com/service: doca-hbn
          svc.dpu.nvidia.com/interface: "app_sf"
        - serviceInterface:
          matchLabels:
          port: ovn
```

DOCA Argus Service

This documentation explains configuration and deployment of DOCA Argus service as DPUService in DPF.

Main Argus concepts are explained in the official DOCA Argus documentation.

The official documentation provides a more comprehensive overview, DPUService users should consult it for detailed explanation of service configuration.

The DOCA Argus usecase in DPF is container threat detection in AI workloads and microservices, utilizing a Bluefield DPU to perform live machine introspection at the hardware level.

Service Components

Argus component runs on the DPU and analyzes specific snippets of volatile memory directly, providing attested insights into the operation of various workloads, whether they are bare-metal, virtualized, or containerized. By default, Argus scans all systems of the host, so for scanning specific systems only refer to official DOCA Argus documentation.

IOMMU Kernel Parameters Requirements

Virtualized environments (running inside a VM)

You must set **both** of the following kernel parameters on the **host**:

- `intel_iommu=on` **or** `amd_iommu=on`
- `iommu=pt`

  Example: `intel_iommu=on iommu=pt`

Bare-metal environments (running directly on hardware)

You have **two valid options**:

1. **Disable IOMMU completely**

   - `intel_iommu=off` **or** `amd_iommu=off`

2. **Enable IOMMU with passthrough**

   - `intel_iommu=on` **or** `amd_iommu=on`

   - `iommu=pt`

For more details, refer to the official NVIDIA DOCA Argus documentation.

Configuration

Configuration files:

DPUServiceConfiguration

```
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceConfiguration
metadata:
  name: argus
  namespace: dpf-operator-system
spec:
  deploymentServiceName: argus
  serviceConfiguration:
    helmChart:
      values:
        config:
          isLocalPath: false
        containerImage: nvcr.io/nvidia/doca/doca_argus:1.1.1-
doca3.2.1
```

DPUServiceTemplate

```
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceTemplate
metadata:
  name: argus
  namespace: dpf-operator-system
spec:
  deploymentServiceName: argus
  helmChart:
    source:
      chart: doca-argus
      repoURL: https://helm.ngc.nvidia.com/nvidia/doca
      version: 1.1.1
```

The general resources are:

DPUFlavor

Defines the DPU flavor for the Argus service.

DPUFlavor

```yaml
apiVersion: provisioning.dpu.nvidia.com/v1alpha1
kind: DPUFlavor
metadata:
  name: dpf-provisioning-argus
  namespace: dpf-operator-system
spec:
  bfcfgParameters:
    - UPDATE_ATF_UEFI=yes
    - UPDATE_DPU_OS=yes
    - WITH_NIC_FW_UPDATE=yes
  configFiles:
    - operation: override
      path: /etc/mellanox/mlnx-bf.conf
      permissions: "0644"
      raw: |
        ALLOW_SHARED_RQ="no"
        IPSEC_FULL_OFFLOAD="no"
        ENABLE_ESWITCH_MULTIPORT="yes"
    - operation: override
      path: /etc/mellanox/mlnx-ovs.conf
      permissions: "0644"
      raw: |
        CREATE_OVS_BRIDGES="no"
        OVS_DOCA="yes"
    - operation: override
      path: /etc/mellanox/mlnx-sf.conf
      permissions: "0644"
      raw: ""
  grub:
    kernelParameters:
      - console=hvc0
      - console=ttyAMA0
      - earlycon=pl011,0x13010000
      - fixrttc
      - net.ifnames=0
```

```
      - biosdevname=0
      - iommu.passthrough=1
      - cgroup_no_v1=net_prio,net_cls
      - hugepagesz=2048kB
      - hugepages=3072
  nvconfig:
    - device: '*'
      parameters:
        - PF_BAR2_ENABLE=0
        - PER_PF_NUM_SF=1
        - PF_TOTAL_SF=20
        - PF_SF_BAR_SIZE=10
        - NUM_PF_MSIX_VALID=0
        - PF_NUM_PF_MSIX_VALID=1
        - PF_NUM_PF_MSIX=228
        - INTERNAL_CPU_MODEL=1
        - INTERNAL_CPU_OFFLOAD_ENGINE=0
        - SRIOV_EN=1
        - NUM_OF_VFS=46
        - LAG_RESOURCE_ALLOCATION=1
        - LINK_TYPE_P1=ETH
        - LINK_TYPE_P2=ETH
  ovs:
    rawConfigScript: |
      _ovs-vsctl() {
        ovs-vsctl --no-wait --timeout 15 "$@"
      }

      _ovs-vsctl set Open_vSwitch . other_config:doca-init=true
      _ovs-vsctl set Open_vSwitch . other_config:dpdk-max-
memzones=50000
      _ovs-vsctl set Open_vSwitch . other_config:hw-offload=true
      _ovs-vsctl set Open_vSwitch . other_config:pmd-quiet-
idle=true
      _ovs-vsctl set Open_vSwitch . other_config:max-idle=20000
```

```
      _ovs-vsctl set Open_vSwitch . other_config:max-
revalidator=5000
      _ovs-vsctl set Open_vSwitch . other_config:ctl-pipe-
size=1024
      _ovs-vsctl --if-exists del-br ovsbr1
      _ovs-vsctl --if-exists del-br ovsbr2
      _ovs-vsctl --may-exist add-br br-sfc
      _ovs-vsctl set bridge br-sfc datapath_type=netdev
      _ovs-vsctl set bridge br-sfc fail_mode=secure
      _ovs-vsctl --may-exist add-port br-sfc p0
      _ovs-vsctl set Interface p0 type=dpdk
      _ovs-vsctl set Port p0 external_ids:dpf-type=physical

      _ovs-vsctl set Open_vSwitch . external-ids:ovn-bridge-
datapath-type=netdev
      _ovs-vsctl --may-exist add-br br-ovn
      _ovs-vsctl set bridge br-ovn datapath_type=netdev
      _ovs-vsctl br-set-external-id br-ovn bridge-id br-ovn
      _ovs-vsctl br-set-external-id br-ovn bridge-uplink
puplinkbrovntobrsfc
      _ovs-vsctl --may-exist add-port br-ovn pf0hpf
      _ovs-vsctl set Interface pf0hpf type=dpdk
```

DPUDeployment

Defines the DPUDeployment for the Argus service.

DPUDeployment

```yaml
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUDeployment
metadata:
  name: argus
  namespace: dpf-operator-system
spec:
  dpus:
    bfb: bf-bundle
    dpuSets:
    - nameSuffix: dpuset-argus
      nodeSelector:
        matchLabels:
          feature.node.kubernetes.io/dpu-enabled: "true"
    flavor: dpf-provisioning-argus
  serviceChains:
    switches:
    - ports:
      - serviceInterface:
          matchLabels:
            uplink: p0
    upgradePolicy:
      applyNodeEffect: true
  services:
    argus:
      serviceConfiguration: argus
      serviceTemplate: argus
```

Configuration

Official Argus documentation explains configuration options.

DPUDeployment

The complete `DPUDeployment` configuration is in DPUDeployment.yaml.

Output

Argus offers multiple ways to get events, that includes logs to stdout, log files and telemtry records in json or syslog formats.

DOCA Blueman Service

Overview

DOCA BlueMan is a standalone web dashboard that runs on the DPU, providing consolidated information about system health, basic information, and telemetry counters through a single interface. All data is gathered from the DOCA Telemetry Service (DTS).

Prerequisites

- DTS service must be running
- DTS Guide

Documentation

- Official Blueman Documentation

Deployment

Starting the Service

To start the BlueMan service, you need to create a `DPUService` resource.

This resource will define the service configuration and deployment details.

Configuration file:

DPUService

```yaml
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUService
metadata:
  name: doca-blueman-service
  namespace: dpf-operator-system
spec:
  helmChart:
    source:
      repoURL: https://helm.ngc.nvidia.com/nvidia/doca
      version: 1.0.8
      chart: doca-blueman
  serviceDaemonSet:
    updateStrategy:
      type: RollingUpdate
      rollingUpdate:
        maxUnavailable: 2
    labels:
      dpuservice.dpu.nvidia.com/name: doca-blueman-service
    annotations:
      dpuservice.dpu.nvidia.com/name: doca-blueman-service
```

```
kubectl apply -f DPUService.yaml
```

Verification

```
kubectl -n dpf-operator-system exec deploy/dpf-operator-
controller-manager -- /dpfctl describe all --show-
resources=dpuservice/doca-blueman-service --show-
conditions=dpuservices
NAME                                                NAMESPACE
STATUS       REASON   SINCE  MESSAGE
DPFOperatorConfig/dpfoperatorconfig                 dpf-operator-
system  Ready: True  Success  85m
  DPUServices
    DPUService/doca-blueman-service                 dpf-operator-
system
                  Ready
True         Success  15m
                  ApplicationPrereqsReconciled
True         Success  17m
                  ApplicationsReady
True         Success  15m
                  ApplicationsReconciled
True         Success  17m
                  ConfigPortsReconciled
True         Success  17m
                  DPUServiceInterfaceReconciled
True         Success  17m
```

Network Configuration

Accessing the UI

By default, the Blueman UI is not accessible from outside the DPF host. To enable external access, configure IP prerouting using the following commands:

```
iptables -t nat -A PREROUTING -p tcp --dport 10000 -j DNAT --to-destination $DPF_BF_IP:10000
iptables -t nat -A PREROUTING -p tcp --dport 443 -j DNAT --to-destination $DPF_BF_IP:443
```

Accessing the Dashboard

Open your browser and navigate to:

```
https://$DPF_HOST_IP/login
```

DOCA Firefly Service

This documentation explains configuration and deployment of DOCA Firefly service as DPUService in DPF.

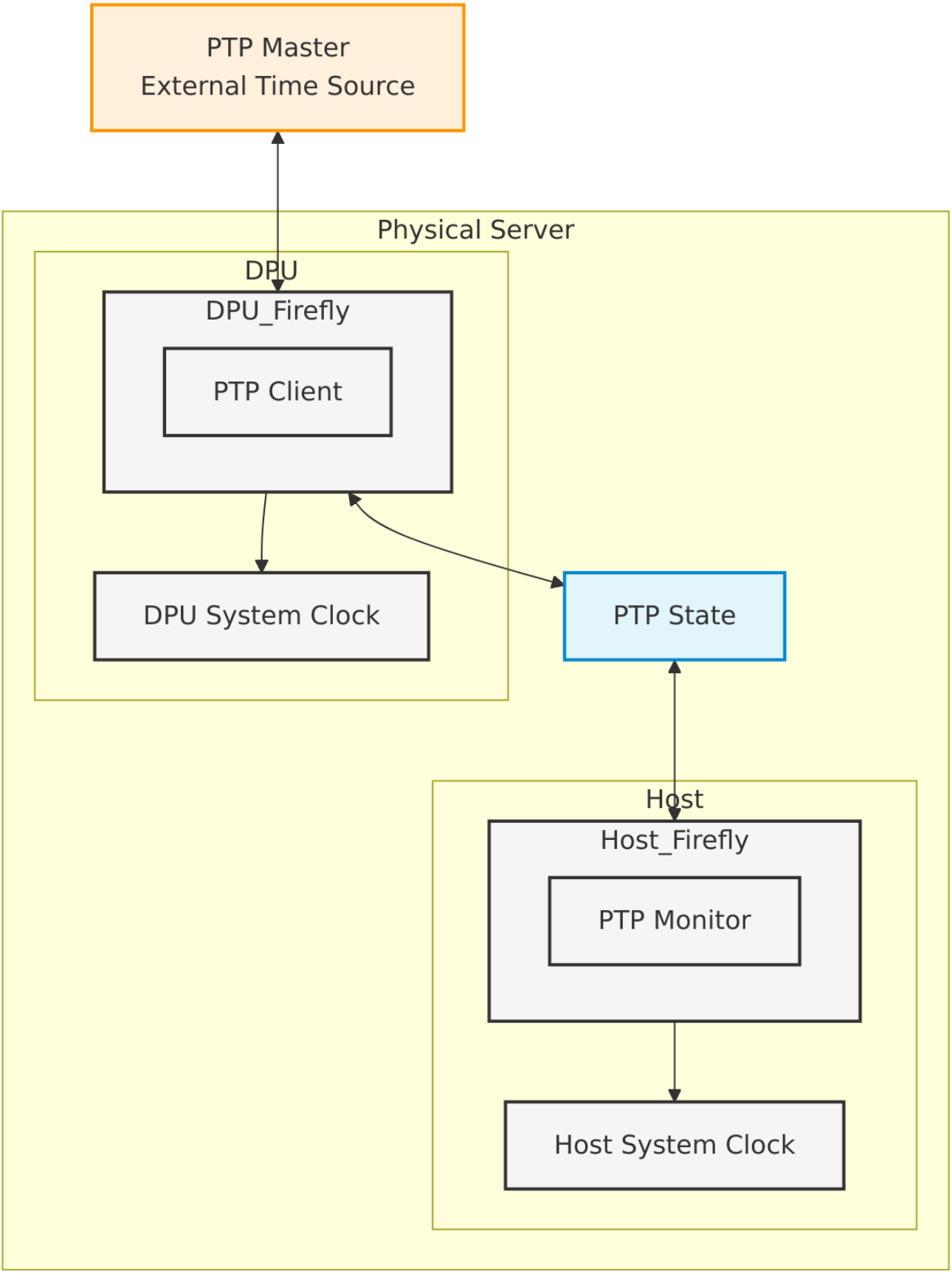
Main Firefly concepts are explained in the official DOCA Firefly documentation.

While the official documentation provides a more comprehensive overview, DPUService users should consult it for detailed explanation of PTP configuration and monitoring options.

The DOCA Firefly usecase in DPF is mainly to provide PTP time synchronization for the host system clocks.

We split the service into two components: one **running on the DPU**, which is running the PTP software stack, and the **other on the host**.

A high-level overview of the **Firefly DPF service architecture** is shown below.


PTP Master
External Time Source
Physical Server
DPU
DPU_Firefly
PTP Client
DPU System Clock
PTP State
Host
Host_Firefly
PTP Monitor
Host System Clock

Service Components

Firefly consists of two main components:

1. DPU Component:

```
* Runs on the DPU
* Acts as the PTP client
* Handles PTP time synchronization
* Sets DPU system clock
```

Configuration files:

DPUServiceConfiguration-dpu

```yaml
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceConfiguration
metadata:
  name: firefly-dpu
  namespace: dpf-operator-system
spec:
  deploymentServiceName: firefly-dpu
  interfaces:
    - name: fireflyiface
      network: mybrsfc-firefly
  serviceConfiguration:
    serviceDaemonSet:
      labels:
        svc.dpu.nvidia.com/custom-flows: firefly
    configPorts:
      ports:
        - name: monitor
          port: 25600
          protocol: TCP
      serviceType: ClusterIP
    helmChart:
      values:
        exposedPorts:
          ports:
            monitor: true
        ptpConfig: ptp.conf
        ptpInterfaces: fireflyiface
        config:
          content:
            ptp.conf: |
              [global]
              domainNumber                    24
              clientOnly                      1
              verbose                         1
              logging_level                   6
```

```
                dataset_comparison              G.8275.x
                G.8275.defaultDS.localPriority  128
                maxStepsRemoved                 255
                logAnnounceInterval             -3
                logSyncInterval                 -4
                logMinDelayReqInterval          -4
                G.8275.portDS.localPriority     128
                ptp_dst_mac                     01:80:C2:00:00:0E
                network_transport               L2
                fault_reset_interval            1
                hybrid_e2e                      0

                [fireflyiface]
```

DPUServiceTemplate-dpu

```
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceTemplate
metadata:
  name: firefly-dpu
  namespace: dpf-operator-system
spec:
  deploymentServiceName: firefly-dpu
  helmChart:
    source:
      chart: doca-firefly
      repoURL: https://helm.ngc.nvidia.com/nvidia/doca
      version: 1.1.8
    values:
      containerImage: nvcr.io/nvidia/doca/doca_firefly:1.7.4-
doca3.2.0
      hostNetwork: false
      enableTXPortTimestampOffloading: true
      monitorState: 0.0.0.0
      phc2sysArgs: -a -r -l 6
      config:
        isLocalPath: false
  resourceRequirements:
    memory: 512Mi
```

2. Host Component:

```
* Runs on the host
* Monitors PTP time synchronization
* Sets host system clock
```

Configuration files:

DPUServiceConfiguration-host.yaml

```
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceConfiguration
metadata:
  name: firefly-host
  namespace: dpf-operator-system
spec:
  deploymentServiceName: firefly-host
  upgradePolicy:
    applyNodeEffect: false
  serviceConfiguration:
    deployInCluster: true
    helmChart:
      values:
        monitorState: '{{ (index .Services "firefly-dpu").Name }}.{{ (index .Services "firefly-dpu").Namespace }}'
```

DPUServiceTemplate-host

```
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceTemplate
metadata:
  name: firefly-host
  namespace: dpf-operator-system
spec:
  deploymentServiceName: firefly-host
  helmChart:
    source:
      chart: doca-firefly
      repoURL: https://helm.ngc.nvidia.com/nvidia/doca
      version: 1.1.8
    values:
      containerImage: nvcr.io/nvidia/doca/doca_firefly:1.7.4-
doca3.2.0-host
      hostNetwork: false
      monitorClientPhc2sysInterface: eth0
      monitorClientType: phc2sys
      phc2sysState: disable
      ppsDevice: disable
      ppsState: do_nothing
      ptpState: disable
      tolerations:
        - effect: NoSchedule
          key: k8s.ovn.org/network-unavailable
          operator: Exists
  resourceRequirements:
    memory: 512Mi
```

The general resources are:

DPUFlavor

Defines the DPU flavor for the Firefly service.

DPUFlavor

```
apiVersion: provisioning.dpu.nvidia.com/v1alpha1
kind: DPUFlavor
metadata:
  annotations:
    provisioning.dpu.nvidia.com/num-of-trusted-sfs: "5"
  name: dpf-provisioning-firefly
  namespace: dpf-operator-system
spec:
  bfcfgParameters:
    - UPDATE_ATF_UEFI=yes
    - UPDATE_DPU_OS=yes
    - WITH_NIC_FW_UPDATE=yes
  configFiles:
    - operation: override
      path: /etc/mellanox/mlnx-bf.conf
      permissions: "0644"
      raw: |
        ALLOW_SHARED_RQ="no"
        IPSEC_FULL_OFFLOAD="no"
        ENABLE_ESWITCH_MULTIPORT="yes"
    - operation: override
      path: /etc/mellanox/mlnx-ovs.conf
      permissions: "0644"
      raw: |
        CREATE_OVS_BRIDGES="no"
        OVS_DOCA="yes"
    - operation: override
      path: /etc/mellanox/mlnx-sf.conf
      permissions: "0644"
      raw: ""
  grub:
    kernelParameters:
      - console=hvc0
      - console=ttyAMA0
      - earlycon=pl011,0x13010000
```

```
      - fixrttc
      - net.ifnames=0
      - biosdevname=0
      - iommu.passthrough=1
      - cgroup_no_v1=net_prio,net_cls
      - hugepagesz=2048kB
      - hugepages=3072
  nvconfig:
    - device: '*'
      parameters:
        - PF_BAR2_ENABLE=0
        - PER_PF_NUM_SF=1
        - PF_TOTAL_SF=20
        - PF_SF_BAR_SIZE=10
        - NUM_PF_MSIX_VALID=0
        - PF_NUM_PF_MSIX_VALID=1
        - PF_NUM_PF_MSIX=228
        - INTERNAL_CPU_MODEL=1
        - INTERNAL_CPU_OFFLOAD_ENGINE=0
        - SRIOV_EN=1
        - NUM_OF_VFS=46
        - LAG_RESOURCE_ALLOCATION=1
        - REAL_TIME_CLOCK_ENABLE=1
        - LINK_TYPE_P1=ETH
        - LINK_TYPE_P2=ETH
  ovs:
    rawConfigScript: |
      _ovs-vsctl() {
        ovs-vsctl --no-wait --timeout 15 "$@"
      }

      _ovs-vsctl set Open_vSwitch . other_config:doca-init=true
      _ovs-vsctl set Open_vSwitch . other_config:dpdk-max-
memzones=50000
      _ovs-vsctl set Open_vSwitch . other_config:hw-offload=true
```

```
      _ovs-vsctl set Open_vSwitch . other_config:pmd-quiet-
idle=true
      _ovs-vsctl set Open_vSwitch . other_config:max-idle=20000
      _ovs-vsctl set Open_vSwitch . other_config:max-
revalidator=5000
      _ovs-vsctl set Open_vSwitch . other_config:ctl-pipe-
size=1024
      _ovs-vsctl --if-exists del-br ovsbr1
      _ovs-vsctl --if-exists del-br ovsbr2
      _ovs-vsctl --may-exist add-br br-sfc
      _ovs-vsctl set bridge br-sfc datapath_type=netdev
      _ovs-vsctl set bridge br-sfc fail_mode=secure
      _ovs-vsctl --may-exist add-port br-sfc p0
      _ovs-vsctl set Interface p0 type=dpdk
      _ovs-vsctl set Port p0 external_ids:dpf-type=physical

      _ovs-vsctl set Open_vSwitch . external-ids:ovn-bridge-
datapath-type=netdev
      _ovs-vsctl --may-exist add-br br-ovn
      _ovs-vsctl set bridge br-ovn datapath_type=netdev
      _ovs-vsctl br-set-external-id br-ovn bridge-id br-ovn
      _ovs-vsctl br-set-external-id br-ovn bridge-uplink
puplinkbrovntobrsfc
      _ovs-vsctl --may-exist add-port br-ovn pf0hpf
      _ovs-vsctl set Interface pf0hpf type=dpdk

      # Disabling DPU NTP. Requires functional PTP setup.
      systemctl disable ntpsec --now
```

DPUServiceNAD

Defines the trusted Scalable Function (SF) for the Firefly service.

DPUServiceNAD

```yaml
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceNAD
metadata:
  name: mybrsfc-firefly
  namespace: dpf-operator-system
  annotations:
    dpuservicenad.svc.dpu.nvidia.com/use-trusted-sfs: ""
spec:
  resourceType: sf
  ipam: false
  bridge: "br-sfc"
  serviceMTU: 1500
```

DPUDeployment

Defines the DPUDeployment for the Firefly service.

DPUDeployment

```yaml
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUDeployment
metadata:
  name: ovn-firefly
  namespace: dpf-operator-system
spec:
  dpus:
    bfb: bf-bundle
    dpuSets:
      - nameSuffix: dpuset1
        nodeSelector:
          matchLabels:
            feature.node.kubernetes.io/dpu-enabled: "true"
    flavor: dpf-provisioning-firefly
  serviceChains:
    switches:
      - ports:
          - serviceInterface:
              matchLabels:
                uplink: p0
          - serviceInterface:
              matchLabels:
                port: ovn
          - service:
              interface: fireflyiface
              name: firefly-dpu
  services:
    firefly-dpu:
      serviceConfiguration: firefly-dpu
      serviceTemplate: firefly-dpu
    firefly-host:
      serviceConfiguration: firefly-host
      serviceTemplate: firefly-host
      dependsOn:
        - name: firefly-dpu
```

```
    ovn:
      serviceConfiguration: ovn
      serviceTemplate: ovn
```

For information about OVN Kubernetes configuration see the OVN Kubernetes user guide

Configuration

Official Firefly documentation explains configuration options.

General note: In the official documentation, all options should be specified in the `ptp.conf` file.

In DPF the same options should be set via the `DPUServiceConfiguration`.

Preconfiguration

Our referenced example `DPUServiceConfigration` includes G.8275.1 PTP Profile configuration. You can set all necessary configurations to your needs.

Network Configuration

The Firefly service requires a **trusted Scalable Function (SF)** to enable secure PTP communication. This interface is added to the SFC bridge using a `DPUServiceNAD` (Network Attachment Definition). The `DPUService` controller takes care of injecting the correct resource to the Pod using that DPUServiceNAD.

DPUDeployment

The complete `DPUDeployment` configuration is in DPUDeployment.

Toleration Configuration

The Firefly service includes a special configuration (toleration) that ensures it always runs on the host system. This is important because:

1. The service needs to run on the host to properly synchronize the system time.
2. Without proper time synchronization, other system components might not work correctly.
3. The toleration prevents the service from being blocked from running on the host.

This configuration is automatically handled in the DPUServiceTemplate and does not require any user configuration.

Check the Status

You can check the status of the Firefly service using the following command:

```
$ kubectl -n dpf-operator-system exec deploy/dpf-operator-
controller-manager -- /dpfctl describe all --grouping=false --show-
conditions=dpuservices
NAME
NAMESPACE            STATUS        REASON             SINCE
MESSAGE
DPFOperatorConfig/dpfoperatorconfig                          dpf-
operator-system  Ready: Trye   Success            31s
  DPUClusters
    DPUCluster/dpu-cplane-tenant1                            dpu-
cplane-tenant1   Ready: True   HealthCheckPassed  27h
  DPUDeployments
    DPUDeployment/firefly                                    dpf-
operator-system  Ready: True   Success            12m
      DPUServices
        DPUService/firefly-dpu-v6pbk                         dpf-
operator-system
                      Ready
True          Success            8m24s
                      ApplicationPrereqsReconciled
True          Success            3h7m
                      ApplicationsReady
True          Success            8m24s
                      ApplicationsReconciled
True          Success            3h7m
                      ConfigPortsReconciled
True          Success            3h7m
                      DPUServiceInterfaceReconciled
True          Success            3h7m
        DPUService/firefly-host-jj98d                        dpf-
operator-system
                      Ready
True          Success            4m46s
                      ApplicationPrereqsReconciled
True          Success            27h
```

```
                        ApplicationsReady
True            Success             4m46s
                        ApplicationsReconciled
True            Success             27h
                        ConfigPortsReconciled
True            Success             27h
                        DPUServiceInterfaceReconciled
True            Success             27h
...
```

DOCA Telemetry Service

This documentation explains configuration and deployment of DOCA Telemetry service (DTS) as DPU service in DPF.

Main DTS concepts are explained in the official DTS documentation.

While the official documentation provides a more comprehensive overview, DPUService users should consult it for its detailed list of telemetry counters and explanation of user options.

Service configuration

Official DTS documentation explains configuration options.

General note: In the official documentation, all options should be specified in the `dts_config.ini` file using the `key=value` format. In DPF the same options should be set via `DPUService.yaml` as `key: value` in yaml format.

To start the DTS service, you need to create a `DPUService` resource.

This resource will define the service configuration and deployment details.

Configuration file:

DPUService

```yaml
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUService
metadata:
  name: doca-telemetry-service
  namespace: dpf-operator-system
spec:
  helmChart:
    source:
      repoURL: https://helm.ngc.nvidia.com/nvidia/doca
      version: 1.23.4
      chart: doca-telemetry
    values:
      configMapData:
        providers: "sysfs,ethtool"
        # aggregator_providers: "fluent_aggr,prometheus_aggr"
        general:
          update: 1000
          sync-time-limit: 10000
          event-buffer-size: 65536
          counter-buffer-size: 65536
        fluent:
          forward:
            enable: 0
            port: 24224
            host: "127.0.0.1"
            tag: "doca_telemetry"
          kafka:
            enable: 0
            brokers: "127.0.0.1:9092"
            topics: "telemetry"
            format: "json"
          influxdb:
            enable: 0
            host: "127.0.0.1"
            port: 8086
```

```
            tag: "dts_measurement"
          elasticsearch:
            enable: 0
            host: "127.0.0.1"
            port: 9200
            index: "dts"
  serviceDaemonSet:
    updateStrategy:
      type: RollingUpdate
      rollingUpdate:
        maxUnavailable: 2
    labels:
      dpuservice.dpu.nvidia.com/name: doca-telemetry-service
    annotations:
      dpuservice.dpu.nvidia.com/name: doca-telemetry-service
  configPorts:
    serviceType: None
    ports:
      - name: httpserverport
        port: 9100
        protocol: TCP
```

```
kubectl apply -f DPUService.yaml
```

Default configuration

The default configuration is set to collect `sysfs` and `ethtool` counters and expose them via Prometheus endpoint.

DPF configuration

User configuration is organized into the following groups:

- providers
- aggregator_providers
- general
- fluent

All user options should be set in DPUService under:

```
spec:
  helmChart:
    values:
      configMapData:
```

The following subsections explain each user option groups

Providers:

Available data providers are listed here: DTS providers.

Note: some of the providers are supported only on host and not on DPU.

To enable data providers in DTS service:

1. Set data providers csv-line under `spec.helmChart.values.configMapData.providers`. E.g.:

```
spec:
  helmChart:
    values:
      configMapData:
        providers: "sysfs,ethtool"
```

2. Optionally, set aggregator providers csv-line under `spec.helmChart.values.configMapData.aggregator_providers`.

General options

Except provider names, official documentation explains various options of `dts_config.ini` file in format of `key=value` pairs.

To set the these options in DPUService.yaml specify them in `key: value` pairs under `spec.helmChart.configMapData.general`.

For instance, DPU service exposes essential general options:

- Sampling interval options:
- `update: 1000` to set the sample interval in milliseconds.
- `sync-time-limit: 10000` buffer rotation time limit in seconds.

Configure Fluent exporter

Additionally Fluent Bit export files can be set via `spec.helmChart.configMapData.fluent.DESTINATION`. Available fluent export destinations are:

- forward
- kafka
- influxdb
- elasticsearch

Default values are exposed for each case in DPUService.yaml. See more details regarding each destination in official documentation.

Run Service

Start service with:

```
kubectl apply -f DPUService.yaml
```

Check the status of DPU service:

```
$ kubectl -n dpf-operator-system exec deploy/dpf-operator-
controller-manager -- /dpfctl describe all --show-
resources=dpuservice/doca-telemetry-service --show-conditions=all
NAME                                                NAMESPACE
READY  REASON   SINCE  MESSAGE
DPFOperatorConfig/dpfoperatorconfig                 dpf-operator-
system  True   Success  15h
             ImagePullSecretsReconciled
True   Success  25d
             SystemComponentsReady
True   Success  15h
             SystemComponentsReconciled
True   Success  25d
  DPUServices
    DPUService/doca-telemetry-service               dpf-operator-
system  True   Success  21d
                ApplicationPrereqsReconciled
True   Success  25d
                ApplicationsReady
True   Success  21d
                ApplicationsReconciled
True   Success  25d
                DPUServiceInterfaceReconciled
True   Success  25d
```

DOCA VPC OVN Service

> **Note**
>
> The OVN VPC service is considered a tech preview and is not recommended for production use.

Overview

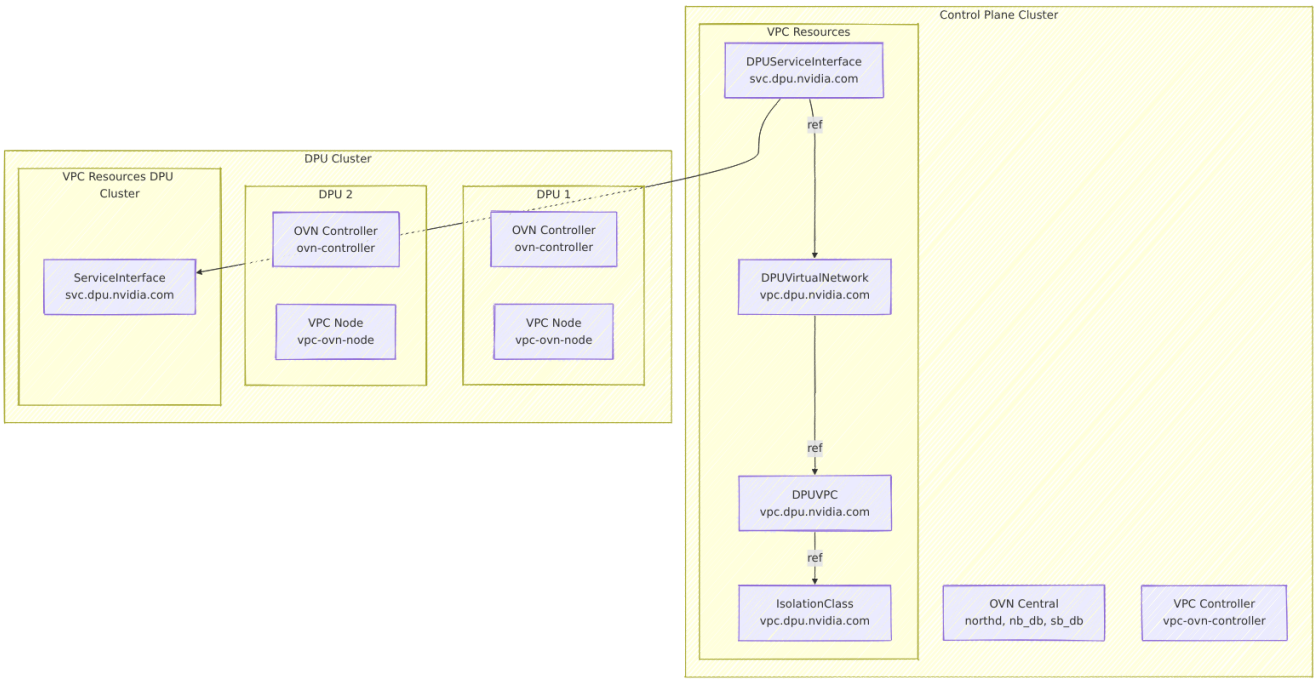
The DOCA VPC OVN Service provides accelerated Virtual Private Cloud (VPC) networking functionality for the DOCA Platform Framework (DPF). Built on top of Open Virtual Network (OVN), this service enables network isolation, virtualization, and advanced SDN capabilities directly on NVIDIA DPUs.

Key Features

- **Multi-tenant Network Isolation**: Create isolated VPCs for different tenants with guaranteed network separationSingle VPC with a single DPUVirtualNetwork and a DPUServiceInterface targeting all DPU Nodes
- **Virtual Network Management**: Support the creation of virtual networks with DHCP and custom IP addressing
- **External Connectivity**: Configurable external routing with NAT/masquerading capabilities
- **Hardware Acceleration**: Leverages DPU hardware acceleration for high-performance networking
- **Flexible Topology**: Support for complex network topologies with inter-network routing controls
- **Kubernetes Integration**: Native Kubernetes resources for declarative VPC management

Architecture

The OVN VPC service consists of several key components that work together:



Component Responsibilities

Control Plane Components

- **OVN Central**: Centralized SDN control plane managing OVN northbound and southbound databases
- **VPC Controller**: Manages VPC-specific Kubernetes resources and coordinates with OVN Central

DPU Cluster Components

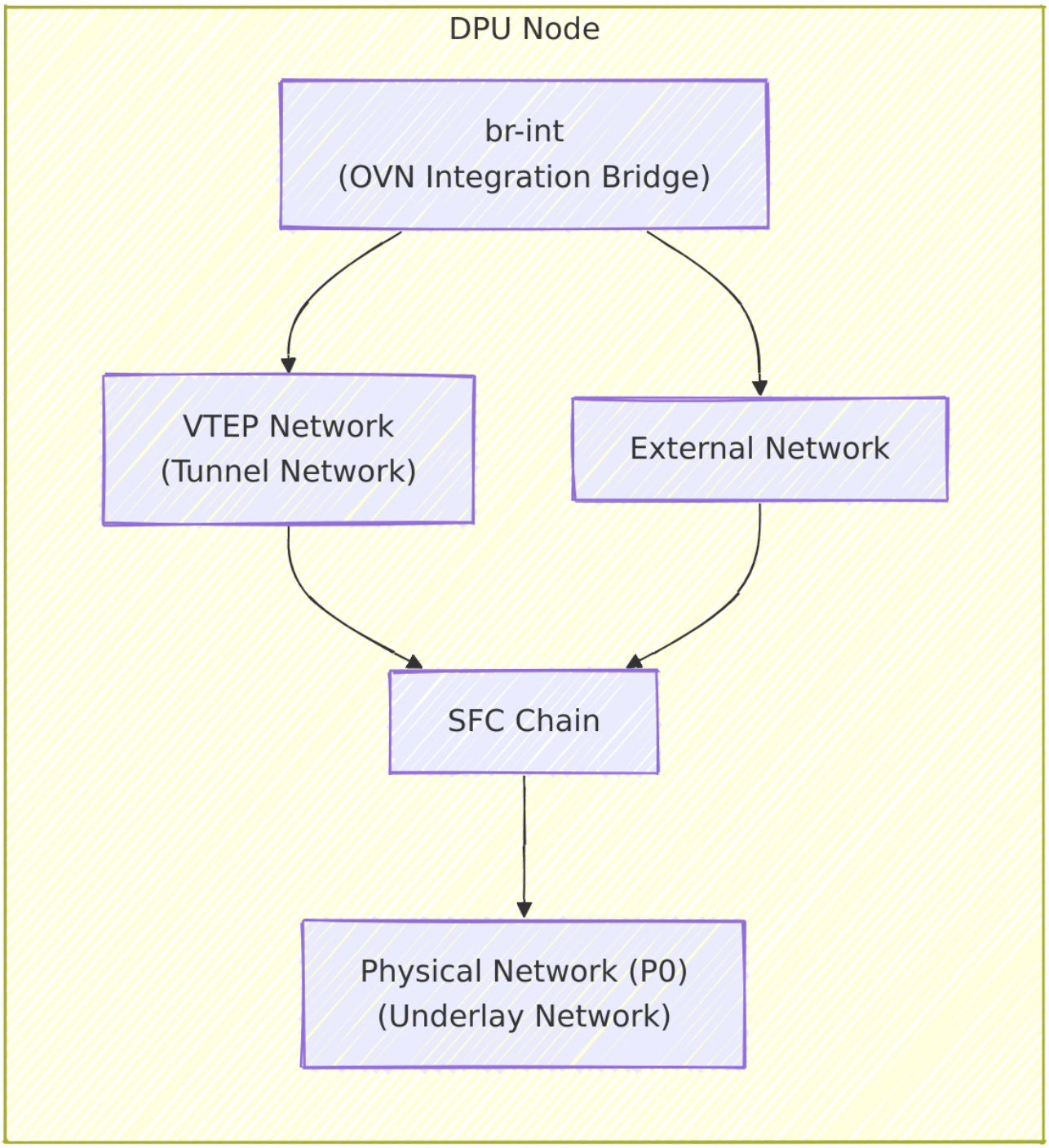
- **OVN Controller**: Local SDN data plane on the DPU, implementing network policies and packet forwarding
- **VPC Node**: Handles local VPC networking operations, including IP allocation and interface management

VPC Kubernetes Resources

- **IsolationClass**: Configuration parameters for VPC implementation
- **DPUVPC**: A Virtual Private Cloud resource that defines an isolated network environment for a tenant, including DPU node selection and isolation configuration
- **DPUVirtualNetwork**: A logical network segment within a VPC
- **DPUServiceInterface**: A control plane resource that defines network interface configuration for virtual networks
- **ServiceInterface**: Physical and virtual network interfaces managed by the DPU that implement network connectivity

DPU Network Configuration

The following diagram depicts the network configuration for a DPU, as facilitated by the OVN VPC service.



- **OVN Integration Bridge**: The main bridge (OVS) that connects interfaces to OVN and handles OVN flows
- **VTEP Network**: The tunnel endpoint interface that enables overlay network connectivity between DPUs
- **External Network**: The interface that provides connectivity to external networks outside the VPC

- **Physical Network**: The primary network interface (typically p0) that connects the DPU to the physical network infrastructure

API Overview

For the full API definition, refer to DPF API Documentation.

IsolationClass

An IsolationClass defines the parameters to be used by the VPC implementation when reconciling a DPUVPC. It specifies:

- **Provisioner**: The VPC provisioner (ovn.vpc.dpu.nvidia.com)
- **Parameters**: Implementation-specific parameters. In this case, OVN-specific configuration such as database endpoints
- **Cluster Scope**: Shared across all VPCs using the same isolation class

Reference Example

```yaml
apiVersion: vpc.dpu.nvidia.com/v1alpha1
kind: IsolationClass
metadata:
  name: ovn.vpc.dpu.nvidia.com
spec:
  # Provisioner type
  provisioner: "ovn.vpc.dpu.nvidia.com"

  # OVN-specific parameters
  parameters:
    # OVN northbound database endpoint
    ovn-nb-endpoint: "tcp:10.1.1.100:30641"
    # Connection retry interval for northbound database (Default: 5 seconds)
    ovn-nb-reconnect-time: "5"
    # OVN southbound database endpoint
    ovn-sb-endpoint: "tcp:10.1.1.100:30642"
    # Connection retry interval for southbound database (Default: 5 seconds)
    ovn-sb-reconnect-time: "5"
```

Note

`ovn-nb-endpoint, ovn-sb-endpoint` should be set to an IP address through which the OVN service is accessible (e.g. the IP of a control plane node).

DPUVPC

A DPUVPC represents a Virtual Private Cloud isolated network environment for a specific tenant. It defines:

- **Tenant Ownership**: Each VPC belongs to a specific tenant
- **Node Selection**: Specifies which DPU nodes participate in the VPC
- **Isolation Class**: The isolation mechanism (OVN-based) to use, referenced by name
- **Inter-Network Access**: Whether virtual networks within the VPC can communicate

Reference Example

```
apiVersion: vpc.dpu.nvidia.com/v1alpha1
kind: DPUVPC
metadata:
  name: example-vpc
  namespace: default
spec:
  # Tenant identifier
  tenant: my-tenant

  # Optional: Select specific DPU nodes (empty = all nodes)
  nodeSelector: {}

  # Isolation class to use, referenced by name
  isolationClassName: "ovn.vpc.dpu.nvidia.com"

  # Whether virtual networks within the DPUVPC can communicate
with each other
  interNetworkAccess: true
```

> **Note**
>
> A DPU node can belong to at most one DPUVPC, as selected by the `spec.nodeSelector` field.

DPUVirtualNetwork

A DPUVirtualNetwork defines a logical network segment within a VPC and supports:

- **Bridged Networks**: Layer 2 bridged networks with DHCP support
- **External Routing**: Optional external connectivity with masquerading
- **IP Address Management**: Automated DHCP with configurable subnets and exclusions
- **Node Selection**: Selective deployment across DPU nodes

Reference Example

```yaml
apiVersion: vpc.dpu.nvidia.com/v1alpha1
kind: DPUVirtualNetwork
metadata:
  name: example-network
  namespace: default
spec:
  # VPC this network belongs to, references a DPUVPC by name in the
same namespace
  vpcName: "example-vpc"

  # Network type (currently only Bridged is supported)
  type: "Bridged"

  # Enable routing to external networks
  externallyRouted: true

  # Enable NAT for external traffic (default: true)
  masquerade: true

  # Bridged network configuration
  bridgedNetwork:
    ipam:
      ipv4:
        # Enable DHCP server
        dhcp: true
        # Network subnet (first IP becomes gateway)
        subnet: "192.178.0.0/16"
        # Optional: Exclude specific IPs from DHCP
        excludeIPs:
          - ip: "192.178.100.10"
          - range:
              start: "192.178.100.100"
              end: "192.178.100.110"
```

DPUServiceInterface

A DPUServiceInterface defines a network interface for DPUs that can be used by workloads running on the DPU or Host. You can reference a DPUVirtualNetwork by name to indicate that the specified DPUServiceInterface should be connected to a specific virtual network.

Supported interface types: `PF`, `VF`, `Service`.

> **Note**
>
> A DPUServiceInterface of type `Service` is intended for advanced use cases where a DPUService network interface needs to be connected to a virtual network. see *Advanced Use-Cases* section below

Reference Example

```
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceInterface
metadata:
  name: vf2-example-network
  namespace: default
spec:
  template:
    spec:
      template:
        spec:
          interfaceType: vf
          vf:
            pfID: 0
            vfID: 2
            parentInterfaceRef: ""
            # DPUVirtualNetwork reference by name in the same
namespace
            virtualNetwork: example-network
```

Deployment & Prerequisites

Please follow the OVN VPC deployment guide to deploy the OVN VPC Service.

OVN VPC Network Features

1. **Intra-VPC Communication**:
   - Traffic between virtual networks within the same VPC is controlled by the `spec.interNetworkAccess` setting in DPUVPC.
   - When enabled, routing is handled by OVN logical routers.
   - When disabled, networks are completely isolated.
2. **External Connectivity**:

- Virtual networks with `spec.externallyRouted: true` can reach external networks.
- Outbound traffic is masqueraded by default unless `spec.masquerade: false`.

3. **DHCP Services**:
   - Each virtual network with `spec.bridgedNetwork.ipam.ipv4.dhcp: true` gets an OVN DHCP server.
   - The first IP in the subnet becomes the gateway.
   - IP exclusions are supported for static assignments.

VPC Network Topology Examples

All examples below assume OVN VPC is deployed together with an `IsolationClass` named `ovn.vpc.dpu.nvidia.com` as described in the deployment guide.

After applying each example, check the status of the resources. Once all resources are ready, you can test connectivity on the host between VF interfaces that correspond to the specified DPUServiceInterface by:

1. Requesting DHCP from OVN (e.g., using `dhclient -v -1 <interface>`).
2. Running traffic between host network interfaces.

1. Single VPC with a single DPUVirtualNetwork and a DPUServiceInterface targeting all DPU Nodes

This basic example allows hosts to communicate with each other (E/W) and to communicate with the external network behind NAT in an isolated manner (e.g., in case the network infrastructure is shared with additional nodes).

The topology consists of:

1. A DPUVPC: `testvpc`
2. A DPUVirtualNetwork: `testnet1`

3. A DPUServiceInterface of type `VF` targeting VF ID `2`: `testnet1-vf2`

VPC Example Topology

```yaml
---
apiVersion: vpc.dpu.nvidia.com/v1alpha1
kind: DPUVPC
metadata:
  name: testvpc
  namespace: default
spec:
  tenant: foo
  isolationClassName: ovn.vpc.dpu.nvidia.com
  interNetworkAccess: true
  nodeSelector: {}
---
apiVersion: vpc.dpu.nvidia.com/v1alpha1
kind: DPUVirtualNetwork
metadata:
  name: testnet1
  namespace: default
spec:
  vpcName: testvpc
  nodeSelector: {}
  type: Bridged
  externallyRouted: true
  masquerade: true
  bridgedNetwork:
    ipam:
      ipv4:
        dhcp: true
        subnet: 192.178.0.0/16
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceInterface
metadata:
  name: testnet1-vf2
  namespace: default
spec:
```

```
  template:
    spec:
      template:
        spec:
          interfaceType: vf
          vf:
            pfID: 0
            vfID: 2
            parentInterfaceRef: ""
            virtualNetwork: testnet1
```

2. Single VPC with Two DPUVirtualNetworks and Two DPUServiceInterfaces targeting separate DPU nodes

This example associates different DPU nodes with different DPUVirtualNetworks within the same VPC.

The topology consists of:

1. A DPUVPC: `testvpc`
2. Two DPUVirtualNetworks: `testnet1`, `testnet2`
3. Two DPUServiceInterfaces targeting different DPU nodes by label: `testnet1-vf2`, `testnet2-vf2`

DPU nodes should be labeled with a distinct label to allow the association of each DPUServiceInterface to a distinct set of nodes.

In this example, we assume two groups of DPU nodes are labeled with `vpc.dpu.nvidia.com/network=testnet1` and `vpc.dpu.nvidia.com/network=testnet2` respectively.

Hosts can communicate with each other (E/W) in an isolated manner as well as reach the external network behind NAT. Nodes will belong to different virtual networks (subnets) according to the specified label.

```
---
apiVersion: vpc.dpu.nvidia.com/v1alpha1
kind: DPUVPC
metadata:
  name: testvpc
  namespace: default
spec:
  tenant: foo
  isolationClassName: ovn.vpc.dpu.nvidia.com
  interNetworkAccess: true
  nodeSelector: {}
---
apiVersion: vpc.dpu.nvidia.com/v1alpha1
kind: DPUVirtualNetwork
metadata:
  name: testnet1
  namespace: default
spec:
  vpcName: testvpc
  nodeSelector: {}
  type: Bridged
  externallyRouted: true
  masquerade: true
  bridgedNetwork:
    ipam:
      ipv4:
        dhcp: true
        subnet: 192.178.0.0/16
---
apiVersion: vpc.dpu.nvidia.com/v1alpha1
kind: DPUVirtualNetwork
metadata:
  name: testnet2
  namespace: default
spec:
```

```
  vpcName: testvpc
  nodeSelector: {}
  type: Bridged
  externallyRouted: true
  masquerade: true
  bridgedNetwork:
    ipam:
      ipv4:
        dhcp: true
        subnet: 192.188.0.0/16
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceInterface
metadata:
  name: testnet1-vf2
  namespace: default
spec:
  template:
    spec:
      nodeSelector:
        matchLabels:
          vpc.dpu.nvidia.com/network: testnet1
      template:
        spec:
          interfaceType: vf
          vf:
            pfID: 0
            vfID: 2
            parentInterfaceRef: ""
            virtualNetwork: testnet1
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceInterface
metadata:
  name: testnet2-vf2
  namespace: default
```

```yaml
spec:
  template:
    spec:
      nodeSelector:
        matchLabels:
          vpc.dpu.nvidia.com/network: testnet2
      template:
        spec:
          interfaceType: vf
          vf:
            pfID: 0
            vfID: 2
            parentInterfaceRef: ""
            virtualNetwork: testnet2
```

3. Two VPCs, each with a Single DPUVirtualNetwork and a Single DPUServiceInterface Targeting Separate DPU Nodes

This example associates two VPCs with different subsets of nodes. Each VPC has a single DPUVirtualNetwork. Each DPUVirtualNetwork is associated with a single DPUServiceInterface. This example corresponds to a multi-tenant environment where different sets of nodes belong to different tenants (VPCs).

The topology consists of:

1. Two DPUVPCs: `redvpc`, `bluevpc`
2. Two DPUVirtualNetworks: `rednet`, `bluenet`
3. Two DPUServiceInterfaces: `rednet-vf2`, `bluenet-vf2`

DPU nodes should be labeled with a distinct label to allow the association to a specific VPC.

In this example, we assume two groups of DPU nodes are labeled with `vpc.dpu.nvidia.com/tenant=red` and `vpc.dpu.nvidia.com/tenant=blue` respectively. Nodes within the same VPC can communicate with each other (E/W) in an isolated manner, as well as reach the external network behind NAT (N/S).

Each VPC and its resources can be created in their own namespace. This example uses the `default` namespace for both.

VPC Example Topology

```
---
apiVersion: vpc.dpu.nvidia.com/v1alpha1
kind: DPUVPC
metadata:
  name: redvpc
  namespace: default
spec:
  tenant: red
  isolationClassName: ovn.vpc.dpu.nvidia.com
  interNetworkAccess: true
  nodeSelector:
    matchLabels:
      vpc.dpu.nvidia.com/tenant: red
---
apiVersion: vpc.dpu.nvidia.com/v1alpha1
kind: DPUVPC
metadata:
  name: bluevpc
  namespace: default
spec:
  tenant: blue
  isolationClassName: ovn.vpc.dpu.nvidia.com
  interNetworkAccess: true
  nodeSelector:
    matchLabels:
      vpc.dpu.nvidia.com/tenant: blue
---
apiVersion: vpc.dpu.nvidia.com/v1alpha1
kind: DPUVirtualNetwork
metadata:
  name: bluenet
  namespace: default
spec:
  vpcName: bluevpc
  nodeSelector:
```

```
    matchLabels:
      vpc.dpu.nvidia.com/tenant: blue
  type: Bridged
  externallyRouted: true
  masquerade: true
  bridgedNetwork:
    ipam:
      ipv4:
        dhcp: true
        subnet: 192.178.0.0/16
---
apiVersion: vpc.dpu.nvidia.com/v1alpha1
kind: DPUVirtualNetwork
metadata:
  name: rednet
  namespace: default
spec:
  vpcName: redvpc
  nodeSelector:
    matchLabels:
      vpc.dpu.nvidia.com/tenant: red
  type: Bridged
  externallyRouted: true
  masquerade: true
  bridgedNetwork:
    ipam:
      ipv4:
        dhcp: true
        subnet: 192.178.0.0/16
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceInterface
metadata:
  name: rednet-vf2
  namespace: default
spec:
```

```
  template:
    spec:
      nodeSelector:
        matchLabels:
          vpc.dpu.nvidia.com/tenant: red
      template:
        spec:
          interfaceType: vf
          vf:
            pfID: 0
            vfID: 2
            parentInterfaceRef: ""
            virtualNetwork: rednet
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceInterface
metadata:
  name: bluenet-vf2
  namespace: default
spec:
  template:
    spec:
      nodeSelector:
        matchLabels:
          vpc.dpu.nvidia.com/tenant: blue
      template:
        spec:
          interfaceType: vf
          vf:
            pfID: 0
            vfID: 2
            parentInterfaceRef: ""
            virtualNetwork: bluenet
```

Monitoring and Troubleshooting

Resource Status

Each VPC resource can be queried for its status, which contains a set of conditions depicting the readiness of each resource.

DPUVPC

```yaml
apiVersion: vpc.dpu.nvidia.com/v1alpha1
kind: DPUVPC
metadata:
  ...
spec:
  ...
status:
  conditions:
  - lastTransitionTime: "2025-06-17T13:51:16Z"
    message: ""
    observedGeneration: 1
    reason: Success
    status: "True"
    type: Ready
  - lastTransitionTime: "2025-06-17T13:51:16Z"
    message: ""
    observedGeneration: 1
    reason: Success
    status: "True"
    type: DPUNodesReconciled
  - lastTransitionTime: "2025-06-17T13:51:16Z"
    message: ""
    observedGeneration: 1
    reason: Success
    status: "True"
    type: DPUVPCReconciled
  - lastTransitionTime: "2025-06-17T13:51:16Z"
    message: ""
    observedGeneration: 1
    reason: Success
    status: "True"
    type: PrerequisitesReady
  - lastTransitionTime: "2025-06-17T13:51:16Z"
    message: ""
    observedGeneration: 1
```

```
    reason: Success
    status: "True"
    type: TopologyReconciled
  observedGeneration: 1
  virtualNetworks:
  - name: rednet
```

The following conditions are used to indicate the status of the VPC:

- `Ready`: Overall readiness of the VPC resource.
- `DPUVPCReconciled`: VPC has been successfully reconciled.
- `TopologyReconciled`: Network topology has been successfully reconciled.
- `DPUNodesReconciled`: DPU nodes have been successfully reconciled with the VPC configuration.
- `PrerequisitesReady`: All required prerequisites for the VPC are in place.

DPUVirtualNetwork

```yaml
apiVersion: vpc.dpu.nvidia.com/v1alpha1
kind: DPUVirtualNetwork
metadata:
  ...
spec:
  ...
status:
  conditions:
  - lastTransitionTime: "2025-06-17T13:51:14Z"
    message: ""
    observedGeneration: 1
    reason: Success
    status: "True"
    type: Ready
  - lastTransitionTime: "2025-06-17T13:51:14Z"
    message: ""
    observedGeneration: 1
    reason: Success
    status: "True"
    type: DPUVirtualNetworkReconciled
  - lastTransitionTime: "2025-06-17T13:51:09Z"
    message: ""
    observedGeneration: 1
    reason: Success
    status: "True"
    type: PrerequisitesReady
  observedGeneration: 1
```

The following conditions are used to indicate the status of the virtual network:

- `Ready`: Overall readiness of the virtual network resource.
- `DPUVirtualNetworkReconciled`: Virtual network has been successfully reconciled.

- `PrerequisitesReady`: All required prerequisites for the virtual network are in place.

DPUServiceInterface

> **Note**
>
> Due to a DPF limitation, the overall readiness of a DPUServiceInterface does not reflect the readiness of the individual ServiceInterfaces for each DPU. Refer to the ServiceInterface section below.

```yaml
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceInterface
  ...
spec:
  ...
status:
  conditions:
  - lastTransitionTime: "2025-06-17T13:51:34Z"
    message: ""
    observedGeneration: 2
    reason: Success
    status: "True"
    type: Ready
  - lastTransitionTime: "2025-06-17T13:51:19Z"
    message: ""
    observedGeneration: 2
    reason: Success
    status: "True"
    type: ServiceInterfacePreReqsReady
  - lastTransitionTime: "2025-06-17T13:51:34Z"
    message: ""
    observedGeneration: 2
    reason: Success
    status: "True"
    type: ServiceInterfaceSetReady
  - lastTransitionTime: "2025-06-17T13:51:19Z"
    message: ""
    observedGeneration: 2
    reason: Success
    status: "True"
    type: ServiceInterfaceSetReconciled
  observedGeneration: 2
```

The following conditions are used to indicate the status of the DPU service interface:

- `Ready`: Overall readiness of the DPU service interface resource.
- `ServiceInterfaceSetReady`: The service interface set in the underlying infrastructure is ready.
- `ServiceInterfaceSetReconciled`: The service interface set in the underlying infrastructure has been successfully reconciled.
- `ServiceInterfacePreReqsReady`: All required prerequisites for the service interface are in place.

ServiceInterface

In some cases, it is necessary to understand the status of the service interface in the underlying infrastructure, for example, when a specific VF/PF is not connected to a virtual network.

In such cases, the user should query the DPU Kubernetes cluster for individual ServiceInterface CRs to understand their overall status.

Relevant fields are shown in the example below. Checking the conditions of the object should generally be sufficient.

```yaml
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: ServiceInterface
metadata:
  annotations:
    ovn.vpc.dpu.nvidia.com/lsp-connected: "true"
    ovn.vpc.dpu.nvidia.com/lsp-mac-address: 5a:19:ad:cf:61:85
spec:
  interfaceType: vf
  node: worker2-0000-c8-00
  vf:
    parentInterfaceRef: -worker2-0000-c8-00
    pfID: 0
    vfID: 2
    virtualNetwork: bluenet
status:
  conditions:
  - lastTransitionTime: "2025-06-17T13:51:40Z"
    message: ""
    observedGeneration: 1
    reason: Success
    status: "True"
    type: Ready
  - lastTransitionTime: "2025-06-17T13:51:40Z"
    message: ""
    observedGeneration: 1
    reason: Success
    status: "True"
    type: ServiceInterfaceReconciled
  observedGeneration: 1
```

The following conditions are used to indicate the status of the service interface:

- `Ready`: Overall readiness of the service interface resource.

- `ServiceInterfaceReconciled`: The service interface has been successfully reconciled.

Check Status of VPC resources using dpfctl

Use `dpfctl` to get a quick view of VPC resources:

```
$ kubectl -n dpf-operator-system exec deploy/dpf-operator-
controller-manager -- /dpfctl describe dpuvpcs --show-
conditions=all
NAME
NAMESPACE            STATUS  REASON   SINCE  MESSAGE
DPFOperatorConfig/dpfoperatorconfig                      dpf-
operator-system
             Ready
True    Success  36m
             ImagePullSecretsReconciled
True    Success  6d
             SystemComponentsReady
True    Success  36m
             SystemComponentsReconciled
True    Success  5d7h
  DPUVPCs
    DPUVPC/bluevpc                                       default
                 Ready
True    Success  25m
                 DPUNodesReconciled
True    Success  25m
                 DPUVPCReconciled
True    Success  25m
                 PrerequisitesReady
True    Success  25m
                 TopologyReconciled
True    Success  25m
      DPUVirtualNetworks
        DPUVirtualNetwork/bluenet                        default
                     Ready
True    Success  25m
                     DPUVirtualNetworkReconciled
True    Success  25m
                     PrerequisitesReady
True    Success  25m
```

```
          DPUServiceInterfaces
            DPUServiceInterface/bluenet-vf2                default
                                Ready
True    Success  24m
                                ServiceInterfacePreReqsReady
True    Success  25m
                                ServiceInterfaceSetReady
True    Success  24m
                                ServiceInterfaceSetReconciled
True    Success  25m
    DPUVPC/redvpc                                          default
                  Ready
True    Success  25m
                  DPUNodesReconciled
True    Success  25m
                  DPUVPCReconciled
True    Success  25m
                  PrerequisitesReady
True    Success  25m
                  TopologyReconciled
True    Success  25m
      DPUVirtualNetworks
        DPUVirtualNetwork/rednet                           default
                      Ready
True    Success  25m
                      DPUVirtualNetworkReconciled
True    Success  25m
                      PrerequisitesReady
True    Success  25m
          DPUServiceInterfaces
            DPUServiceInterface/rednet-vf2                 default
                                Ready
True    Success  24m
                                ServiceInterfacePreReqsReady
True    Success  25m
```

```
                                ServiceInterfaceSetReady
True    Success  24m
                                ServiceInterfaceSetReconciled
True    Success  25m
```

Default route for VPC VirtualNetworks

Problems

- ServiceInterfaces attached to different VirtualNetworks within the same VPC cannot reach each other.
- ServiceInterfaces trying to reach the external network do not use the VPC interface.

In both cases traffic egresses via the primary cluster network instead of the VPC gateway.

Assumptions: The VPC is correctly configured, and the `DPUVPC` resource is configured with `spec.interNetworkAccess: true` and `DPUVirtualNetwork` resource is configured with `spec.externallyRouted: true`.

Cause

The primary network's default route has equal or higher priority than the VPC network's default route (its metric is less than or equal to the VPC route's metric), so traffic prefers the primary network and bypasses the VPC gateway.

Fix

Prefer the VPC default route. The VPC route cannot set a metric and defaults to 0, so raise the primary network's default-route metric above 0 (e.g., 100).

Check

Connect to the worker and check default routes:

```
ip route show

default via 10.100.0.1 dev enp202s0f0np0 # VPC related interface
default via 10.0.110.254 dev eno1 proto static metric 100 # The host's
cluster network interface
10.0.110.0/24 dev eno1 proto kernel scope link src 10.0.110.31
10.100.0.0/16 dev enp202s0f0np0 proto kernel scope link src 10.100.0.2
```

Ensure the VPC default route has default metric value 0 and the primary default route has a higher metric (e.g., 100).

Verify

Verify you have the VPC default route with higher priority as described in the Fix.

- ServiceInterfaces on different VirtualNetworks within the same VPC can reach each other.
- ServiceInterfaces targeting the external network use the VPC interface as their default route.
- Traffic uses the VPC gateway (e.g., traceroute shows the VPC gateway as the first hop).

Common Issues

VPC Not Ready:

- Verify the IsolationClass exists and is properly configured.
- Ensure OVN Central is accessible.

Infrastructure Network Connectivity Issues:

- Check OVN database connectivity (ports 30641, 30642).
- Validate VTEP and external network configuration.
- Verify physical network connectivity between DPU nodes on both VTEP and external networks.
- Validate the ServiceInterface was properly reconciled and is ready.

Network Connectivity Issues for host interfaces connected to VPC:

- Verify the specific ServiceInterface is ready.
- Ensure the target network interface (e.g., VF/PF netdev) has the same MAC address as in the ServiceInterface annotation `vpc.dpu.nvidia.com/lsp-mac-address`.
- Ensure the host interface is up and has an IP address (or assign it one using DHCP/manual).
- Ensure the MTU is set to 9000 for optimal performance.

DHCP Not Working:

- Confirm the virtual network has DHCP enabled (`spec.bridgedNetwork.ipam.ipv4.dhcp: true`).
- Check for IP conflicts in subnet ranges.
- Ensure the target ServiceInterface is ready.

Limitations

- **Production Use**: Not recommended for production use.
- **Network Types**: Only bridged networks are supported.
- **IPv6**: Limited IPv6 support.
- **High Availability**: High availability (HA) for OVN Central is supported, but it is not recommended for production deployments.

- **DPUVirtualNetwork nodeSelector**: The DPUVirtualNetwork `spec.nodeSelector` currently has no effect.
- **OVN Central connection is not secure**: Currently only `tcp` transport is supported.
- **Disabling Masquerade is not supported**: The DPUVirtualNetwork `spec.masquerade` should not be set to `false` as this use-case is not currently supported.

Advanced Use-cases

Service interface with unknown MAC address

There are cases where a service interface needs to have an unknown MAC address so that it can transmit packets with any source MAC (e.g. a network service transmitting packets from arbitrary L2 sources).

To support this case, add the following annotation to your DPUServiceInterface: `ovn.vpc.dpu.nvidia.com/unknown-mac: true`

Example

```
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceInterface
metadata:
  ...
spec:
  template:
    spec:
      template:
        metadata:
          annotations:
            ovn.vpc.dpu.nvidia.com/unknown-mac: "true"
    ...
```

Connecting a DPUServiceInterface of type Service to VPC

Similarly to DPUServiceInterfaces of types `pf` and `vf`, it is possible to connect a DPUServiceInterface of type `service`.

The interface of the DPU Service can then be assigned an IP from the VPC via DHCP. Since the DPU service is ultimately a Kubernetes Pod, this requires the use of the `dhcp` IPAM CNI plugin when setting up the secondary network for the DPU Service interface.

To achieve this, direct access to the DPU Cluster is currently required.

1. Create a custom `NetworkAttachmentDefinition` in the DPU cluster:

```
apiVersion: k8s.cni.cncf.io/v1
kind: NetworkAttachmentDefinition
metadata:
  annotations:
    k8s.v1.cni.cncf.io/resourceName: nvidia.com/bf_sf
  name: mybrint-vpc
  namespace: dpf-operator-system
spec:
  config: |-
    {
      "bridge": "br-int",
      "cniVersion": "0.4.0",
      "interface_type": "dpdk",
      "mtu": 1500,
      "type": "ovs",
      "ipam": {
        "type": "dhcp",
        "daemonSocketPath": "/run/vpc/cni/dhcp.sock"
      }
    }
```

2. Create your DPUService and DPUServiceInterface as usual, providing a reference to this network in `spec.serviceDaemonSet.annotations`:

```yaml
---
apiVersion: "svc.dpu.nvidia.com/v1alpha1"
kind: DPUServiceInterface
metadata:
  name: my-interface
  namespace: my-namespace
spec:
  template:
    spec:
      template:
        spec:
          interfaceType: service
          service:
              interfaceName: my-sf
              serviceID: my-service
              network: dpf-operator-system/mybrint-vpc
              virtualNetwork: testnet1
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUService
metadata:
  name: my-service
  namespace: my-namespace
spec:
  helmChart:
    source:
      ...
    values:
      ...
  serviceID: my-service
  serviceDaemonSet:
    resources:
      nvidia.com/bf_sf: 1
    annotations:
      k8s.v1.cni.cncf.io/networks: |
```

```
[{"name": "mybrint-vpc", "namespace": "dpf-operator-system", "interface": "my_sf"}]
```

Connecting a DPUServiceInterface of type Service to VPC with unknown MAC

Similarly to the previous section, this requires direct access to the DPU Cluster.

1. Create a custom `NetworkAttachmentDefinition` in the DPU cluster:

```
apiVersion: k8s.cni.cncf.io/v1
kind: NetworkAttachmentDefinition
metadata:
  annotations:
    k8s.v1.cni.cncf.io/resourceName: nvidia.com/bf_sf
  name: mybrint-vpc-nodhcp
  namespace: dpf-operator-system
spec:
  config: |-
    {
      "bridge": "br-int",
      "cniVersion": "0.4.0",
      "interface_type": "dpdk",
      "mtu": 1500,
      "type": "ovs"
    }
```

2. Create your DPUService and DPUServiceInterface as usual, providing a reference to this network in `spec.serviceDaemonSet.annotations`:

```yaml
---
apiVersion: "svc.dpu.nvidia.com/v1alpha1"
kind: DPUServiceInterface
metadata:
  name: my-interface
  namespace: my-namespace
spec:
  template:
    spec:
      template:
        metadata:
          annotations:
            ovn.vpc.dpu.nvidia.com/unknown-mac: "true"
        spec:
          interfaceType: service
          service:
              interfaceName: my-sf
              serviceID: my-service
              network: dpf-operator-system/mybrint-vpc-nodhcp
              virtualNetwork: testnet1
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUService
metadata:
  name: my-service
  namespace: my-namespace
spec:
  helmChart:
    source:
      ...
    values:
      resources:
        nvidia.com/bf_sf: 1
  serviceID: my-service
  serviceDaemonSet:
```

```
    annotations:
      k8s.v1.cni.cncf.io/networks: |
        [{"name": "mybrint-vpc", "namespace": "dpf-operator-system", "interface":
"my_sf"}]
```

VPC OVN Upgrade Guide

For zero-downtime VPC OVN upgrade guide refer to VPC OVN upgrade guide

OVN High Availability (HA) guide

> **Note**
>
> The OVN High Availability (HA) service is considered a tech preview and is not recommended for production use.

For OVN High Availability (HA) support guide refer to OVN HA guide

OVN High Availability (HA) guide

To achieve high availability (HA) for OVN, deploy the `ovn-central` DPUService with the `enableHA` option enabled.

0. Prerequisites

Requires at least three schedulable control-plane nodes for quorum.

Use `$HELM_REGISTRY_REPO_URL` and `TAG` to specify the Helm chart registry and your desired version respectively.

```
## The repository URL for the NVIDIA Helm chart registry.
## Usually this is the NVIDIA Helm NGC registry. For development
purposes, this can be set to a different repository.
export HELM_REGISTRY_REPO_URL=https://helm.ngc.nvidia.com/nvidia/doca
## The DPF TAG is the version of the OVN components which will be
deployed in this guide.
export TAG=v25.10.0
```

1. Deploy ovn-central DPUService

```
cat examples/dpuservice-ovn-central.yaml | envsubst | kubectl
apply -f -
```

This will deploy `ovn-central` DPUService with `enableHA` set to `true`, it will create 3 ovn-central pods on the control-plane nodes.

```yaml
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUService
metadata:
  name: ovn-central
  namespace: dpf-operator-system
spec:
  deployInCluster: true
  helmChart:
    source:
      repoURL: $HELM_REGISTRY_REPO_URL
      version: $TAG
      chart: ovn-chart
    values:
      exposedPorts:
        ports:
          ovnnb: true
          ovnsb: true
      management:
        ovnCentral:
          enabled: true
          enableHA: true
          affinity:
            nodeAffinity:
              requiredDuringSchedulingIgnoredDuringExecution:
                nodeSelectorTerms:
                  - matchExpressions:
                      - key: "node-role.kubernetes.io/master"
                        operator: Exists
                  - matchExpressions:
                      - key: "node-role.kubernetes.io/control-plane"
                        operator: Exists
          tolerations:
            - key: node-role.kubernetes.io/master
              operator: Exists
```

```
            effect: NoSchedule
          - key: node-role.kubernetes.io/control-plane
            operator: Exists
            effect: NoSchedule
```

VPC OVN upgrade guide

To achieve a zero-downtime VPC OVN upgrade, upgrade the services in this order: vpc-ovn-node ovn-controller, ovn-central, then vpc-ovn-controller.

Assuming the deployment of all the DPF VPC OVN componenets are done based on the following guide. VPC OVN Deployment

0. Required variables

Use $TARGET_TAG as your desired version.

```
## The TARGET_TAG is the desired version of the DPF VPC OVN
components which will be upgraded.
export TARGET_TAG=v25.10.0
```

1. Upgrade vpc-ovn-node service

Patch the `DPUServiceTemplate` of the vpc-ovn-node service with the upgraded chart version.

```
kubectl patch dpuservicetemplate vpc-ovn-node -n dpf-operator-
system --type=json --patch="[{\"op\": \"replace\", \"path\":
\"/spec/helmChart/source/version\", \"value\": \"$TARGET_TAG\"}]"
```

Ensure the `DPUDeployment` is Ready.

```
kubectl wait --for=condition=ready --namespace dpf-operator-system
dpudeployment vpc-ovn
```

2. Upgrade ovn-controller service

Patch the `DPUServiceTemplate` of the ovn-controller service with the upgraded chart version.

```
kubectl patch dpuservicetemplate ovn-controller -n dpf-operator-
system --type=json --patch="[{\"op\": \"replace\", \"path\":
\"/spec/helmChart/source/version\", \"value\": \"$TARGET_TAG\"}]"
```

Ensure the `DPUDeployment` is Ready.

```
kubectl wait --for=condition=ready --namespace dpf-operator-system
dpudeployment vpc-ovn
```

3. Upgrade ovn-central service

Patch the `DPUServiceTemplate` of the ovn-central service with the upgraded chart version.

```
kubectl patch dpuservicetemplate ovn-central -n dpf-operator-
system --type=json --patch="[{\"op\": \"replace\", \"path\":
\"/spec/helmChart/source/version\", \"value\": \"$TARGET_TAG\"}]"
```

Ensure the `DPUDeployment` is Ready.

```
kubectl wait --for=condition=ready --namespace dpf-operator-system
dpudeployment vpc-ovn
```

4. Upgrade vpc-ovn-controller service

Patch the `DPUServiceTemplate` of the vpc-ovn-controller service with the upgraded chart version.

```
kubectl patch dpuservicetemplate vpc-ovn-controller -n dpf-
operator-system --type=json --patch="[{\"op\": \"replace\", \"path\":
\"/spec/helmChart/source/version\", \"value\": \"$TARGET_TAG\"}]"
```

Ensure the `DPUDeployment` is Ready.

```
kubectl wait --for=condition=ready --namespace dpf-operator-system
dpudeployment vpc-ovn
```

DPUServiceChain

This document describes how to use DPUServiceChain in DPF.

Introduction

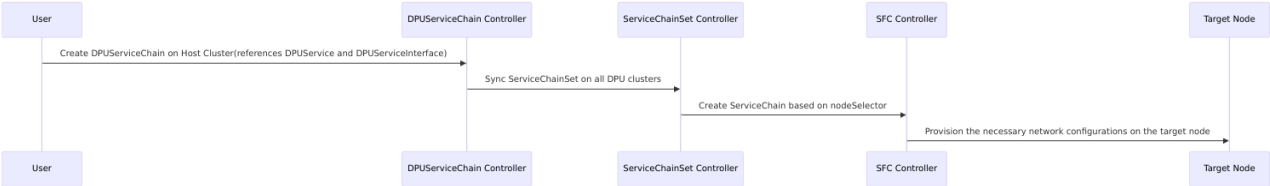
The purpose of DPUServiceChain is to allow user to define how to steer traffic on DPU through DPUServiceInterfaces. The following controllers are used internally to achieve this.

1. User creates DPUServiceChain, servicechaincontroller consumes it on the host cluster.

2. ServiceChainSet is created on DPU clusters

3. ServiceChain is created for individual nodes based on nodeSelector.

4. SFC controller provisions the necessary network configurations on DPU.



How to Use DPUServiceChain

DPUServiceChain example:

The following YAML manifest defines a DPUServiceChain named `example-chain` and refers to one DPUService named `example-service` and 4 DPUServiceInterfaces. DPUServiceChain will define how the traffic will flow through those DPUServiceInterfaces.

`DPUServiceInterfaces`

```
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceInterface
metadata:
  name: p0
  namespace: dpf-operator-system
spec:
  template:
    spec:
      template:
        metadata:
          labels:
            uplink: "p0"
        spec:
          interfaceType: physical
          physical:
            interfaceName: p0
```

```yaml
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceInterface
metadata:
  name: pf0hpf
  namespace: dpf-operator-system
spec:
  template:
    spec:
      nodeSelector:
        matchExpressions:
          - key: kubernetes.io/os
            operator: In
            values:
              - "linux"
      template:
        metadata:
          labels:
            uplink: "pf0hpf"
        spec:
          interfaceType: pf
          pf:
            pfID: 0
```

```yaml
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceInterface
metadata:
  name: eth1
  namespace: dpf-operator-system
spec:
  template:
    spec:
      template:
        metadata:
          labels:
            svc.dpu.nvidia.com/interface: "eth1"
            svc.dpu.nvidia.com/service: example-service
        spec:
          interfaceType: service
          service:
            serviceID: example-service
            network: mybrsfc
            interfaceName: eth1
```

```yaml
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceInterface
metadata:
  name: eth2
  namespace: dpf-operator-system
spec:
  template:
    spec:
      template:
        metadata:
          labels:
            svc.dpu.nvidia.com/interface: "eth2"
            svc.dpu.nvidia.com/service: example-service
        spec:
          interfaceType: service
          service:
            serviceID: example-service
            network: mybrsfc
            interfaceName: eth2
```

`DPUService`

```yaml
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUService
metadata:
  name: example-service
  namespace: dpf-operator-system
spec:
  serviceID: example-service
  interfaces:
    - eth1
    - eth2
  helmChart:
    source:
      repoURL: https://helm.ngc.nvidia.com/nvidia/doca
      version: 1.0.1
      chart: example-service
    values:
      resources:
        memory: 6Gi
```

`DPUServiceChain`

```yaml
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceChain
metadata:
  name: example-chain
  namespace: dpf-operator-system
spec:
  template:
    spec:
      template:
        spec:
          switches:
            - ports:
                - serviceInterface:
                    matchLabels:
                      uplink: p0
                - serviceInterface:
                    matchLabels:
                      svc.dpu.nvidia.com/service: example-service
                      svc.dpu.nvidia.com/interface: eth1
            - ports:
                - serviceInterface:
                    matchLabels:
                      svc.dpu.nvidia.com/service: example-service
                      svc.dpu.nvidia.com/interface: eth2
                - serviceInterface:
                    matchLabels:
                      uplink: pf0hpf
              serviceMTU: 2000
```

Let's break it down step by step.

1. There are 4 DPUServiceInterfaces

    1. uplink port p0

    2. uplink port pf0hpf on host
    3. service interface eth1
    4. service interface eth2
2. There is one DPUService
    1. example-service which has two interfaces eth1 and eth2
3. There is one DPUServiceChain
    1. example-chain `p0 --> eth1 --> eth2 --> pf0hpf`
4. On the switches object there is `serviceMTU` which defines the MTU between services in the DPU cluster
    1. The default value for this MTU is 1500
    2. **NOTE**: This only affects services on the DPU, it should be aligned with the general MTU set for the traffic in the network
    3. **NOTE**: that when changed, it will restart every pod that relates to this switch

In the above example, traffic will flow from uplink port p0 to example DPU service's eth1 iface. From eth1 iface, it will go to eth2 iface(eth1->eth2 is handled by the service itself and not by the chain) and then to uplink port pf0hpf on the host.

DPUServiceCredentialRequest

In order to request credential to access a cluster API Server, users should deploy a `DPUServiceCredentialRequest` custom resource.

Example

The following is an example of a `DPUServiceCredentialRequest` which requests credentials to access a cluster with a duration of 1h.

```yaml
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceCredentialRequest
metadata:
  name: dpu-01-credential-request
  namespace: default
spec:
  serviceAccount:
    name: dpu-01-sa
    namespace: default
  duration: 1h
  targetCluster:
    name: dpu-cplane-tenant1
    namespace: default
  type: kubeconfig
  secret:
    name: dpu-01-credential
    namespace: default
  metadata:
    labels:
      dpuservice: dpu-01
```

In the above example:

- A `DPUServiceCredentialRequest` named `dpu-01-credential-request` is created, that request a `ServiceAccount` to be created in a specific cluster. As part of the request, a token has to be requested for the `ServiceAccount` with a `ttl` of 1h. The output that is expected is a secret of type `kubeconfig`, containing all the needed information in order to access the cluster i.e. the token, API Server endpoint and the cluster CA.

- `targetCluster` is a reference to an existing `dpuCluster` that will provide the needed authentication information to access the API SERVER.

- The controller will create the `ServiceAccount` and request a token with the requested duration. It will generate an output of the requested `.spec.type` and

create a secret with the output pasted as `spec.data`.

- If the `DPUServiceCredentialRequest` does not exist, or is not up-to-date, the controller will try to bring it as close to the desired state as possible.
- The controller will requeue the `DPUServiceCredentialRequest` in order to reconcile it when 80% of the token last time has elapsed. It will then request a new token.

You can run this example by saving it into `dpuservice_credential_request.yaml`

```
$ kubectl apply -f dpuservice_credential_request.yaml
```

How to use the DPUServiceCredentialRequest output to get a client

The `DPUServiceCredentialRequest` creates a secret with the following types:

- `kubeconfig`
- `tokenFile`

Using type kubeconfig

The `kubeconfig` type creates a valid `kubeConfig` file and save it as a kubernetes secret. This file will automatically be updated with valid authentication information. The secret can be mounted in a `pod` and use to create a `client` in order to access to the target cluster API Server.

If using `client.go`, by mounting the secret and setting `KUBECONFIG` to the mounted path, we can use clientcmd to create a `client`. clientcmd also provides the necessary helpers to directly create a `client` using a `kubeConfig` file.

Using type tokenFile

The `tokenFile` type create a secret containing the following keys:

- `TOKEN_FILE`, contains the token attached to the `ServiceAccount`. It can be used as a bearer token.
- `KUBERNETES_SERVICE_HOST`, contains the target cluster API Server host.
- `KUBERNETES_SERVICE_PORT`, contains the target cluster API Server port.
- `KUBERNETES_CA_DATA`, contains the target cluster API Server certificate authority.

These are the needed information in order to access the target cluster API Server. Generally we want to use the `KUBERNETES_*` as environment variables and mount the `TOKEN_FILE`. Using `client.go` it is possible to have a self rotating `client`. For this, the default environment variables and files must be overwritten with the content of the secret. The following example shows how to do so:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: mypod
  namespace: default
spec:
  containers:
  - name: test-container
    image: ubuntu
    command: ["/bin/sh", "-c"]
    args: ["sleep 1000"]
    env:
    - name: KUBERNETES_SERVICE_HOST
      valueFrom:
        secretKeyRef:
          name: my-secret
          key: KUBERNETES_SERVICE_HOST
    - name: KUBERNETES_SERVICE_PORT
      valueFrom:
        secretKeyRef:
          name: my-secret
          key: KUBERNETES_SERVICE_PORT
    volumeMounts:
    - name: tokenfile
      mountPath: "/var/run/secrets/kubernetes.io/serviceaccount"
      readOnly: true
  volumes:
  - name: tokenfile
    projected:
      sources:
      - secret:
          name: dpu-01-credential
          items:
          - key: TOKEN_FILE
            path: token
```

```
      - secret:
          name: dpu-01-credential
          items:
          - key: KUBERNETES_CA_DATA
            path: ca.crt
---
apiVersion: v1
kind: Secret
metadata:
  name: dpu-01-credential
  namespace: default
type: Opaque
data:
  TOKEN_FILE: c29tZSByZWFsIHNlcmlvdXMgdG9rZW4gbm93Cg==
  KUBERNETES_SERVICE_HOST: aHR0cHM6Ly9kZXZlbG9wZXIuY29t
  KUBERNETES_SERVICE_PORT: MTk0NDM=
  KUBERNETES_CA_DATA:
LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSURERENDQWUyQ0NRRHlQa0JnR3
```

We can then call InClusterConfig() and get a self rotating client (This is automatically done with controller-runtime `GetConfigOrDie`). This will free the user from having to watch the mounted secret files.

Note This will overwrite the default values, which are necessary to access the local API Server. If this is not desirable, it is still possible to adapt the `InClusterConfig()` logic.

API Server Consideration

The `DPUServiceCredentialRequest` will request tokens for the provided `ServiceAccount` by setting the `Audience` to the API server audience based on the server config. The specific configuration flag is `--api-audiences`. The service account token authenticator will validate that tokens used against the API server are bound to at least one of these audiences.

DPUServiceIPAM

This document describes how a user can work with the `DPUServiceIPAM` Custom Resource. A `DPUServiceIPAM` object is responsible for doing the IP Address Management in the DPU cluster, IPs which the rest of the DPF objects like `DPUServiceChain` can consume. Under the hood, this object is using nvidia-k8s-ipam.

Modes

The `DPUServiceIPAM` can operate in 2 distinct modes:

- `.spec.ipv4Subnet`: Share a subnet across all the nodes by splitting the subnet in smaller unique chunks per node. In this mode, there is a single gateway for the whole subnet the `DPUServiceIPAM` is managing.
- `.spec.ipv4Network`: Split a CIDR into a unique subnet per node. In this mode, each node has its own gateway.

Created Child Custom Resources

As previously mentioned, the `DPUServiceIPAM` controller leverages the nvidia-k8s-ipam project. The controller creates either a CIDRPool or a IPPool Custom Resource in the underlying DPU cluster depending on the mode the `DPUServiceIPAM` is configured.

Switching from one mode to another leads to the previous Custom Resource being deleted and the new one to be created.

Removing the pool, doesn't affect running Pods. Readding the same pool, assuming that there was no node addition/deletion in the DPU cluster, is a safe operation and will not lead to duplicate IP consumption by Pods.

In order to avoid duplicating docs related to the usage of each field, take a look at the API Documentation. Note that there are additional fields that can be used but in the interest of simplicity, this guide uses only the required ones.

Split subnet into chunk per node

To split a subnet into chunks per node, all of them sharing the same gateway, we have to create a Custom Resource like this:

```
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceIPAM
metadata:
  name: example-pool1
spec:
  metadata:
    labels:
      # Specifying a label here enables us to make use of this
object in DPUServiceChain.
      svc.dpu.nvidia.com/pool: example-pool1
  ipv4Subnet:
    subnet: "192.168.0.0/20"
    gateway: "192.168.0.1"
    perNodeIPCount: 256
```

This configuration creates chunks of `256` available IPs for each node with gateway `192.168.0.1` for all of those nodes.

Split CIDR into subnet per node

To split a CIDR into a subnet per node each having a distinct gateway, we have to create a Custom Resource like this:

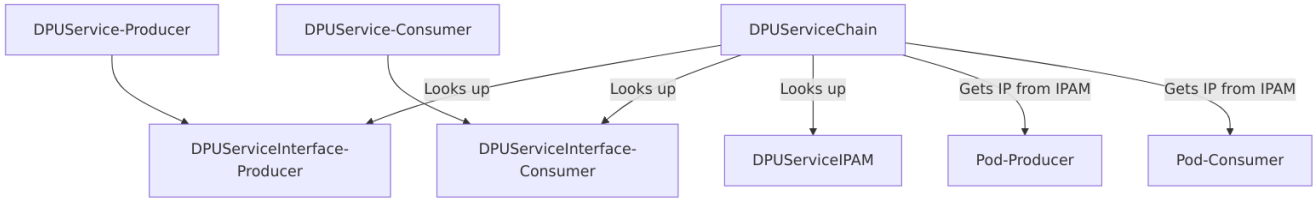
```
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceIPAM
metadata:
  name: example-pool1
spec:
  metadata:
    labels:
      # Specifying a label here enables us to make use of this
object in DPUServiceChain.
      svc.dpu.nvidia.com/pool: example-pool1
  ipv4Network:
    network: "192.168.0.0/20"
    # Optional field that usually makes sense to set, unless we
use a point to point network (/31) and we want to use
    # both IPs.
    gatewayIndex: 1
    prefixSize: 24
```

This configuration creates subnets of size `/24` for each node with gateway the first IP in that subnet.

Consuming IPAM in DPUServiceChain

For our workload to consume an IP from the IPAM we have installed in the cluster, we need to create a `DPUServiceChain` and reference the IPAM object via labels. Below is an example of 2 fake `DPUServices` that are part of the same chain and each of them require an IP address from a particular pool. We won't go into details on the `DPUServiceInterface` or `DPUService` objects here, but we provide reference examples to ensure that the `DPUServiceChain` is understandable.

The following diagram may help illustrate the objects that are involved.



First, we need to create the `DPUServiceIPAM`. We can create the one specified in Split subnet into chunk per node.

Order of applying those resources doesn't matter, but in the interest of making the flow easier to understand, we apply some order.

Then, we apply the `DPUService` and its `DPUServiceInterface` resources for each of the service:

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceInterface
metadata:
  name: producer
spec:
  template:
    spec:
      template:
        metadata:
          labels:
            svc.dpu.nvidia.com/interface: app-iface
            svc.dpu.nvidia.com/service: producer
        spec:
          interfaceType: service
          service:
            interfaceName: app-iface
            network: mybrsfc
            serviceID: producer
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUService
metadata:
  name: producer
spec:
  serviceID: producer
  helmChart:
    source:
      repoURL: https://example.com/charts
      path: producer
      version: v0.0.1
    values:
      container1:
        resources:
          requests:
```

```
            nvidia.com/sf: 1
  interfaces:
  - producer


---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceInterface
metadata:
  name: consumer
spec:
  template:
    spec:
      template:
        metadata:
          labels:
            svc.dpu.nvidia.com/interface: app-iface
            svc.dpu.nvidia.com/service: consumer
        spec:
          interfaceType: service
          service:
            interfaceName: app-iface
            network: mybrsfc
            serviceID: consumer
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUService
metadata:
  name: consumer
spec:
  serviceID: consumer
  helmChart:
    source:
      repoURL: https://example.com/charts
      path: consumer
      version: v0.0.1
```

```
    values:
      worker:
        resources:
          requests:
            nvidia.com/sf: 1
  interfaces:
  - consumer
```

Then we create the `DPUServiceChain`. In this `DPUServiceChain` we request that the 2 interfaces are part of the same switch and each of the interfaces request an IP from the pool we have created above:

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceChain
metadata:
  name: producer-consumer
spec:
  template:
    spec:
      template:
        spec:
          switches:
            - ports:
              - serviceInterface:
                  matchLabels:
                    svc.dpu.nvidia.com/interface: app-iface
                    svc.dpu.nvidia.com/service: producer
                  ipam:
                    # Reference to the label we have specified in
the DPUServiceIPAM
                    matchLabels:
                      svc.dpu.nvidia.com/pool: pool1
              - serviceInterface:
                  matchLabels:
                    svc.dpu.nvidia.com/interface: app-iface
                    svc.dpu.nvidia.com/service: consumer
                  ipam:
                    # Reference to the label we have specified in
the DPUServiceIPAM
                    matchLabels:
                      svc.dpu.nvidia.com/pool: pool1
```

Assuming that the Helm Charts above are valid and the respect the DPF Contract, the outcome of the above resources will be that each node on the DPU cluster will have 2 Pods, one consumer and one producer. These Pods will have 2 interfaces each, one

primary and a secondary using a Scalable Function (SF). The SFs will have an IP configured on them from the pool defined above, from the chunk that is allocated on the node.

Limitations

- `DPUServiceIPAM` CRs must be created in `dpf-operator-system` namespace to take effect.

DPUServiceNAD

Overview

The `DPUServiceNAD` Custom Resource Definition (CRD) allows you to create custom NetworkAttachmentDefinitions (NADs) for your DPU services. These NADs define how secondary network interfaces are configured for pods running on DPUs. NetworkAttachmentDefinition is a CRD defined by multus and used for setting up the secondary interface for the pod. Pod spec provides which NAD to use as a configuration in annotation `k8s.v1.cni.cncf.io/networks: <nadName>`. Multus CNI consumes the annotation, fetches the NAD configuration, and invokes the relevant CNI binary with the configuration given in the NAD to facilitate the creation of a secondary interface on the pod. The following pages are recommended (as a 101 guide) if the user is not familiar with NAD or Multus.

https://github.com/k8snetworkplumbingwg/multus-cni/blob/master/docs/how-to-use.md

https://github.com/k8snetworkplumbingwg/multus-cni/blob/master/docs/how-to-use.md#create-network-attachment-definition

https://github.com/k8snetworkplumbingwg/network-resources-injector?tab=readme-ov-file#network-resources-injection-example

Installation

DPUServiceNAD is automatically installed as part of the DPF operator installation. No additional installation steps are required.

Usage

Creating a DPUServiceNAD

Here's an example of creating a basic DPUServiceNAD with SF (Scalable Function):

```
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceNAD
metadata:
  name: mynad
  namespace: my-namespace
  labels:
    labelTest: labelTestValue
  annotations:
    annotTest: annotTestValue
spec:
  resourceType: "sf"           # Resource type (sf, vf, or veth)
  bridge: "test-ovsbridge"     # OVS bridge name
  mtu: 1500                     # MTU size
  ipam: true                    # Enable IPAM
```

The DPUServiceNAD controller underneath takes care of converting this DPUServiceNAD CRD into a NAD CRD object and applying it to all the DPU Clusters. Users can choose their preferred resource (sf, vf, or veth), custom OVS bridge, MTU, and IPAM. It is mandatory for OVS bridge to exist beforehand, our controllers do not create the ovs bridge. This CRD allows users to choose their own OVS bridge along with their preferred MTU/resource type for their workload's secondary interface.

Using DPUServiceNAD with DPUServiceInterface

To use your DPUServiceNAD in a DPUServiceInterface:

```
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceInterface
metadata:
  name: dpu-service-interface-with-custom-nad
  namespace: my-namespace
spec:
  template:
    spec:
      template:
        spec:
          interfaceType: service
          service:
            serviceID: "service-one"
            network: "mynad"            # Reference to your
DPUServiceNAD
            interfaceName: "net1"
```

Dynamic Resource Injection

When you use a DPUServiceNAD with a DPUServiceInterface in a DPUService, the system automatically injects the appropriate resources based on the resource type:

- **1.** For `sf` (service function) type:
  - If we have the `dpuservicenad.svc.dpu.nvidia.com/use-trusted-sfs` annotation in DPUServiceNAD, then we inject the trusted_sf `nvidia.com/bf_sf_trusted` resource.
  - If the above annotation is not present, we inject the default sf `nvidia.com/bf_sf` resource.
- **2.** For `vf` (virtual function) type:
  - Injects `nvidia.com/bf_vf` resource

- **3.** For `veth` type:
    - No resource injection

Example of an ArgoApplication(which is created from the DPUService which is using the NAD):

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  finalizers:
  - resources-finalizer.argocd.argoproj.io
  name: dpu-cluster-1-example-grb87
  namespace: dpf-operator-system
spec:
  destination:
    name: dpu-cluster-1
    namespace: dpf-operator-system
  project: doca-platform-project-dpu
  source:
    chart: hello-world
    helm:
      valuesObject:
        serviceDaemonSet:
          annotations:
            annKey: annVal
            k8s.v1.cni.cncf.io/networks: '[{"name":"mynad","interface":"dpu-service-interface-with-custom-nad","cni-args":null}]'
          labels:
            labelKey: labelVal
          resources:
            nvidia.com/bf_sf: "1"
```

`nvidia.com/bf_sf` resource is dynamically injected by the controller based on the DPUServiceNAD and DPUServiceInterface configuration that dpuService is using. The

system will automatically merge the injected resources with any existing resources in the ArgoCD application (which is created from DPUService) object's serviceDaemonSet as shown in the above example.

Chained CNIs

DPUServiceNAD allows chaining CNIs to combine the DPF OVS CNI with additional CNI plugins to provide enhanced networking capabilities. This is particularly useful when you need to chain RDMA (Remote Direct Memory Access) CNI with the DPF OVS CNI.

Here's how to setup a DPUServiceNAD that provides SFs and RDMA devices on the Pods and connects the representors to the `test-ovsbridge`:

```
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceNAD
metadata:
  name: rdma-chained-nad
  namespace: my-namespace
spec:
  resourceType: "sf"
  bridge: "test-ovsbridge"
  mtu: 1500
  chainedCNIs:
  - type: rdma
```

DPUSet

The DPUSet is a Kubernetes CRD which managed the DPU CRs in DPF.

Updating the DPUSet

An update to the DPUSet can be done for upgrading the BFB or modifying provisioning parameters.

> **Note**
>
> This operation will result in a network disruption and also a host reboot. A rolling update can be configured to control the number of nodes that will be out-of-service in parallel (Please see the DPUSet YAML example below). The cluster can also be divided into several DPU-Sets, please refer to the section "Using several DPU Sets"

These are the required steps for upgrading the BFB on a set of DPUs (The BFB is specified as part of the DPU Set CRD):

1. Create a BFB YAML that includes the required BFB file and also assigns a distinct name for the object (Different from the currently used BFB objects). After applying the YAML, the BFB will be pulled from the specified URL to the shared storage:

```
---
apiVersion: provisioning.dpu.nvidia.com/v1alpha1
kind: BFB
metadata:
  name: bf-bundle-new
  namespace: dpf-operator-system
spec:
  url: https://content.mellanox.com/BlueField/BFBs/Ubuntu24.04/bf-bundle-3.2.1-34_25.11_ubuntu-24.04_64k_prod.bfb
```

2. Update the DPUSet YAML to point to the new BFB object:

```
---
apiVersion: provisioning.dpu.nvidia.com/v1alpha1
kind: DPUSet
metadata:
  name: dpuset
  namespace: dpf-operator-system
spec:
  dpuNodeSelector:
    matchLabels:
      feature.node.kubernetes.io/dpu-enabled: "true"
  strategy:
    rollingUpdate:
      maxUnavailable: "10%"
    type: RollingUpdate
  dpuTemplate:
    spec:
      dpuFlavor: passthrough
      bfb:
        name: bf-bundle-new
      nodeEffect:
        taint:
          key: "dpu"
          value: "provisioning"
          effect: NoSchedule
```

3. Then delete the DPU objects of the relevant DPUs.

This will initiate a provisioning cycle for the DPUs using the new BFB image:

```
kubectl delete dpu -n dpf-operator-system worker1-0000-2b-00
worker2-0000-2b-00
```

4. You can later delete the previous BFB object:

```
kubectl delete bfb -n dpf-operator-system bf-bundle
```

Using several DPU Sets

There's an option to create several DPU-Set objects, and assign them to different groups of worker nodes. This is done by adding relevant labels to the node selector in the DPUSet object YAML. Each DPU Set can use a different BFB object, can have a different DPU flavor, a different rolling update strategy, etc.

For example:

```yaml
---
apiVersion: provisioning.dpu.nvidia.com/v1alpha1
kind: DPUSet
metadata:
  name: dpuset-dk
  namespace: dpf-operator-system
spec:
  dpuNodeSelector:
    matchLabels:
      e2e.servers/dk: "true"
  strategy:
    rollingUpdate:
      maxUnavailable: "10%"
    type: RollingUpdate
  dpuTemplate:
    spec:
      dpuFlavor: passthrough
      bfb:
      name: bf-bundle-dk-ga
      nodeEffect:
      taint:
        key: "dpu"
        value: "provisioning"
        effect: NoSchedule
```

Cluster Configuration

The `spec.dpuTemplate.spec.cluster` section in the DPUSet allows you to specify configuration for the Kubernetes cluster that the DPU will join.

nodeLabels

The `spec.cluster.nodeLabels` field specifies custom labels to add to the Kubernetes node when the DPU joins the cluster. This is a map of string key-value pairs. The system automatically adds the following labels in addition to any custom labels you specify: - `provisioning.dpu.nvidia.com/host` - The hostname of the host machine - `operator.dpu.nvidia.com/dpf-version` - The DPF version used

Example usage with custom labels:

```
spec:
  dpuTemplate:
    spec:
      cluster:
        nodeLabels:
          environment: production
          workload-type: ai-training
          rack-id: "rack-42"
```

DPU Selection

The DPUSet provides two complementary mechanisms for selecting which DPUs should be managed: `dpuNodeSelector` and `dpuSelector`.

dpuNodeSelector

The `dpuNodeSelector` field is used to select DPUNodes based on their labels. It uses the standard Kubernetes LabelSelector format, which supports both `matchLabels` and `matchExpressions`.

For example, to select nodes with a specific label:

```
spec:
  dpuNodeSelector:
    matchLabels:
      feature.node.kubernetes.io/dpu-enabled: "true"
```

Or using match expressions for more complex selection:

```
spec:
  dpuNodeSelector:
    matchExpressions:
      - key: environment
        operator: In
        values:
          - production
          - staging
```

dpuSelector

The `dpuSelector` field is used to further filter DPUDevices based on their labels. This is applied after the `dpuNodeSelector` and allows you to select specific DPUDevices on the selected DPUNodes.

The `dpuSelector` uses a simple map of label key-value pairs, where all specified labels must match (AND logic).

For example, to select only specific DPU models or configurations:

```
spec:
  dpuNodeSelector:
    matchLabels:
      feature.node.kubernetes.io/dpu-enabled: "true"
  dpuSelector:
    provisioning.dpu.nvidia.com/dpudevice-pciAddress: "0000:1a:00.0"
```

In Host Trusted model, the automatically created DPUDevice objects have the following labels:

- `provisioning.dpu.nvidia.com/dpudevice-num-of-pfs`: The number of PFs on the DPU device
- `provisioning.dpu.nvidia.com/dpudevice-pciAddress`: The PCI address of the DPU device
- `provisioning.dpu.nvidia.com/dpudevice-pf0-name`: The name of PF0 on the DPU device
- `provisioning.dpu.nvidia.com/dpunode-name`: The name of the DPUNode the DPU is part of

Selection Flow

Host Power-cycle in DPU provisioning

If the version of running BFB is lower than 2.7 before DPU provisioning, the BlueField firmware upgrades and mlxconfig parameter changes require a host power-cycle. Once the version of BFB is updated to be greater than or equal to 2.7 a regular reboot would be enough.

For enabling this, the DPUSet provides one annotations in dpuTemplate: `provisioning.dpu.nvidia.com/host-power-cycle-required` - trigger the host power-cycle (cold boot) instead of warm reboot after DPU provisioning, notice that after

the power cycle command is done the annotation would be removed from the DPU and DPUSet objects.

Following is an example to enable host power-cycle:

```
---
apiVersion: provisioning.dpu.nvidia.com/v1alpha1
kind: DPUSet
metadata:
  name: dpuset
  namespace: dpf-operator-system
spec:
  dpuNodeSelector:
    matchLabels:
      feature.node.kubernetes.io/dpu-enabled: "true"
  strategy:
    rollingUpdate:
      maxUnavailable: "10%"
    type: RollingUpdate
  dpuTemplate:
    annotations:
      provisioning.dpu.nvidia.com/host-power-cycle-required: "true"
    spec:
      dpuFlavor: passthrough
      bfb:
        name: bf-bundle-new
      nodeEffect:
        taint:
          key: "dpu"
          value: "provisioning"
          effect: NoSchedule
```

IPMI Command Annotation for Kubernetes Worker Node

The provisioning controller will issue a `ipmi` command to the host to do host power-cycle(cold boot) or warm reboot after DPU provisioning. The default host power-cycle command is `ipmitool chassis power cycle` and warm reboot command is `ipmitool chassis power reset`

For some kinds of servers that uses `ipmitool chassis power reset` command for host cold power-cycle instead of `ipmitool chassis power cycle`. DPF supports changing the host power-cycle/warm reboot command by setting the following annotation on such kind of worker nodes:

```
provisioning.dpu.nvidia.com/powercycle-command: reset
provisioning.dpu.nvidia.com/reboot-command: cycle
```

Node effect

Node effect specifies how changes to the DPU should affect the Kubernetes Node the DPU belongs to. Only the following options can be specified:

- noEffect (bool) - no effect on the node at all
- customLabel (object) - adds the label provided to the Kubernetes Node and DPUNode, ONLY relevant to Kubernetes environment, when there is a match of DPUNode and Kubernetes Node
- taint (object) - marks the node as tainted, ONLY relevant to Kubernetes environment, when there is a match of DPUNode and Kubernetes Node
- drain (bool)(default) - drains the node and waits till the draining is finished, ONLY relevant to Kubernetes environment, when there is a match of DPUNode and Kubernetes Node - this is the default behaviour in a Kubernetes environment
- customAction (string) - name of a ConfigMap which contains a Pod definition - in YAML - to run which will apply the node effect. The Pod is expected to exit when node effect is done, if the Pod terminates with error then DPU would move to Error phase. First, create a ConfigMap with the pod definition:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: custom-node-effect
  namespace: dpf-operator-system
data:
  pod.yaml: |
    apiVersion: v1
    kind: Pod
    metadata:
      name: custom-node-effect-pod
      namespace: dpf-operator-system
    spec:
      containers:
      - name: node-effect
        image: ubuntu:20.04
        command: ["/bin/bash"]
        args:
        - -c
        - |
          # Example custom node effect script
          echo "Applying custom node effect..."
          # Add your custom logic here
          # For example: network configuration, system checks,
reboot.
          sleep 10  # Simulating some work
          exit 0    # Exit successfully when done
      restartPolicy: Never
```

Then, create the DPUSet that uses this custom action

```yaml
apiVersion: provisioning.dpu.nvidia.com/v1alpha1
kind: DPUSet
metadata:
  name: dpuset-custom-effect
  namespace: dpf-operator-system
spec:
  dpuNodeSelector:
    matchLabels:
      feature.node.kubernetes.io/dpu-enabled: "true"
  strategy:
    rollingUpdate:
      maxUnavailable: "10%"
    type: RollingUpdate
  dpuTemplate:
    spec:
      dpuFlavor: passthrough
      bfb:
        name: bf-bundle-new
      nodeEffect:
        customAction: custom-node-effect
```

- hold (bool) - places an annotation with key `wait-for-external-nodeeffect` on the DPU object and waits for it to be removed - this is the default behaviour in a non Kubernetes environment

Additional Node Effect Configuration

The following additional fields can be configured within the `nodeEffect` section:

- **force** (bool) - when set to `true`, the node effect is applied immediately, bypassing `dpfOperatorConfig.spec.provisioningController.multiDPUOperationsS` and `dpfOperatorConfig.spec.provisioningController.maxUnavailableDPUNo` settings. Default: `false`.

- **applyOnLabelChange** (bool) - when set to `true`, changes to `spec.cluster.nodeLabels` on DPUs in Ready state will trigger the node effect logic. This is useful when you want the configured node effect (drain, taint, etc.) to be applied when DPU cluster node labels are updated. Default: `false`.

Example configuration:

```
spec:
  dpuTemplate:
    spec:
      cluster:
        nodeLabels:
          environment: staging
          tier: standard
      nodeEffect:
        drain: true
        applyOnLabelChange: true
```

With this configuration, updating the `spec.cluster.nodeLabels` on a Ready DPU (e.g., adding a new label `tier: production` or changing `environment: staging` to `environment: production`) will trigger the drain node effect.

- **nodeMaintenanceAdditionalRequestors** (array of strings) - additional requestors to be added to the NVIDIA NodeMaintenance CR when Drain is selected. This field is automatically managed by the DPUDeployment controller for coordinating upgrades of DPUServices and DPUServiceChains. Advanced users can also set this manually in standalone DPUSets to coordinate maintenance with external systems.

Example for advanced manual coordination:

```yaml
spec:
  dpuTemplate:
    spec:
      nodeEffect:
        drain: true
        nodeMaintenanceAdditionalRequestors:
          - "custom-coordination-service"
          - "external-maintenance-system"
```

Architecture

This section contains architectural documentation for the DOCA Platform Framework (DPF), including:

- System Design - High-level overview of DPF components and their roles
- Component interactions - Detailed descriptions of each DPF component and how they interact

Component Description

DOCA Platform Framework (DPF) streamlines the provisioning and orchestration of NVIDIA BlueField DPUs in Kubernetes environments.

It implements a dual-cluster architecture:

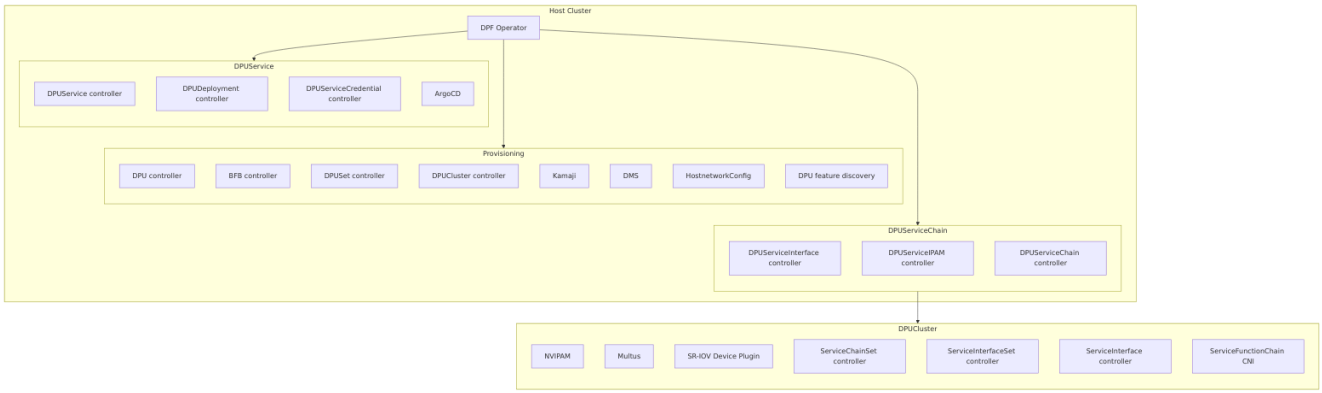
- **Host Cluster**: Provisions and managed DPUs and hosting DPU Cluster control plane components.
- **DPU Cluster**: Manages the lifecycle of services deployed on DPUs.

Refer to the installation guide for setup details, dependencies, and prerequisites.

DPF system components

DPF is made up of the following sets of components:

- DPF Operator to install and configure the DPF system.
- Provisioning components to manage the lifecycle of the DPUs including OS installation, configuration and Kubernetes Node creation.
- DPUService components to manage the full lifecycle of services running on DPUs.
- DPUServiceChain components to manage the lifecycle of ServiceFunctionChains on DPUs.



Provisioning components

This component set uses the Kamaji Cluster Manager - but other Cluster managers may be used.

In the host cluster control plane

- Kamaji system components - including the DPF Kamaji Cluster manager
    - Manage the lifecycle of a Kamaji pod-based control plane
    - Manage the Kamaji cluster load balancer
    - Communicate with the host control plane and DPU control plane
- BFB controller
    - Download the BFB from a remote server
    - Communicate with the host control plane and remote BFB server

- DPUSet controller
  - Create DPU objects and manage their lifecycle
  - Select a Kubernetes control plane for DPU Cluster nodes to join
  - Communicate with the host control plane
- DPU controller
  - Flash the BFB to the DPU
  - Communicate with the DOCA Management Service

On each node in the host cluster

- Node feature discovery - including DPU Detector
  - Add information about DPUs to the Kubernetes Node representing the host node
  - Communicate with the host control plane and host node filesystem
- DOCA Management Service (DMS)
  - Flash a BFB to DPU hardware
  - Communicate with the BlueField DPU and DPU Controller
- Hostnetwork configuration
  - Configure up Virtual Functions, bridges and routes for Host to DPU communication
  - Communicate with the host node through CLI calls

DPUService components

In the host cluster control plane

- DPUService controller
  - Manage the lifecycle of DPUServices created by users
  - Manage the lifecycle of ArgoCD Applications linked to DPUServices
  - Communicate with the host control plane
- DPUDeployment controller
  - Manage the lifecycle of a group of DPUServices and DPUSets
  - Communicate with the host control plane
- DPUServiceCredential controller
  - Manage authorization and authentication for communication between the host control plane and DPU control plane
  - Communicate with the host control plane and DPU control plane
- ArgoCD system components
  - Manage lifecycle of helm charts on the DPU Cluster
  - Communicate with the host control plane and DPU control plane

DPUServiceChain components

In the host cluster control plane

- DPUServiceInterface controller
  - Manage the lifecycle of DPUServiceInterfaces created by users
  - Communicate with the host control plane and DPU control plane
- DPUServiceIPAM controller
  - Manage the lifecycle of the DPUServiceIPAM created by users
  - Communicate with the host control plane and DPU control plane

- DPUServiceChain controller
    - Manage the lifecycle of the DPUServiceChain created by users
    - Communicate with the host control plane and DPU control plane

In the DPU cluster control plane

- NVIDIA IPAM Plugin (NVIPAM)
    - Manage allocation of IPs in the DPU cluster
    - Communicate with the DPU control plane
- ServiceChainSet controller
    - Manage the lifecycle of ServiceChainSets on the DPUCluster
    - Create ServiceChain objects for relevant DPU nodes
    - Communicate with the DPU control plane
- ServiceInterfaceSet controller
    - Manage the lifecycle of ServiceInterfaceSets on the DPUCluster
    - Create ServiceInterface objects for relevant DPU nodes
    - Communicate with the DPU control plane

On each node in the DPU cluster

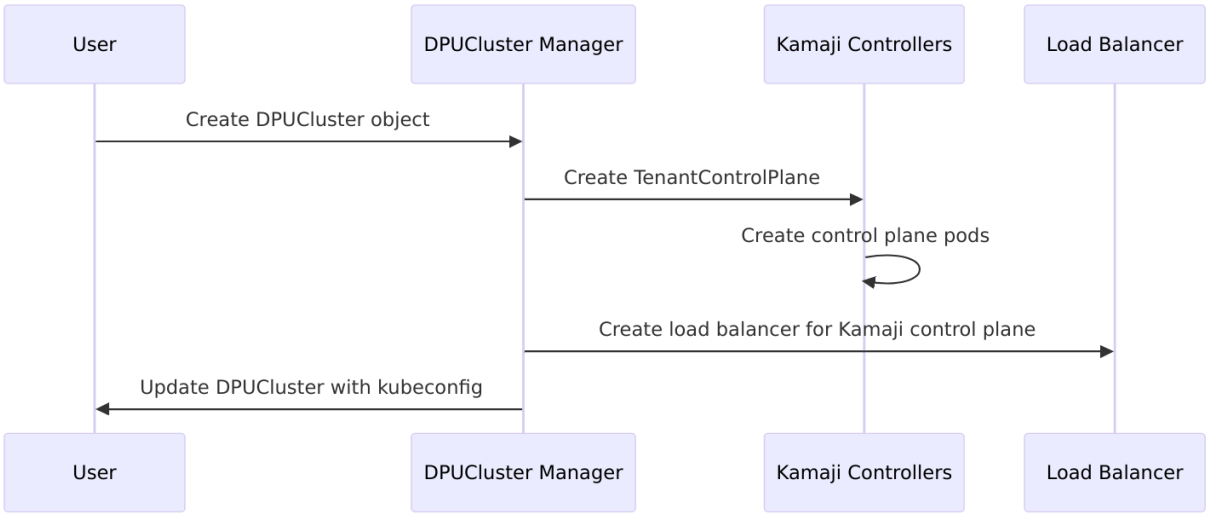
- ServiceInterface controller
    - Creates ovs ports on DPU based on ServiceInterface objects
    - Communicate withe the DPU control plane and host OVS
- ServiceChain controller

    - Create ovs flows on DPU based on ServiceChain objects
    - Communicate with the DPU control plane and DPU host OVS
- ServiceFunctionChain CNI
    - Adds ovs network interfaces to pods
    - Communicate with Container Runtime Interface and host OVS
- NVIDIA IPAM Plugin (NVIPAM)
    - Allocate IPs for pods on the DPU node
    - Communicate with the DPU control plane and host OS
- Multus
    - Allocate network devices for pods on DPU nodes
    - Communicate with the DPU control plane, Container Runtime Ibnterface and host OS
- SR-IOV Device Plugin
    - Manage the lifecycle of Virtual Functions on the DPU node
    - Communicate with the host OS and Kubelet

Provisioning user flows

DPF provisioning has four principle user flows.

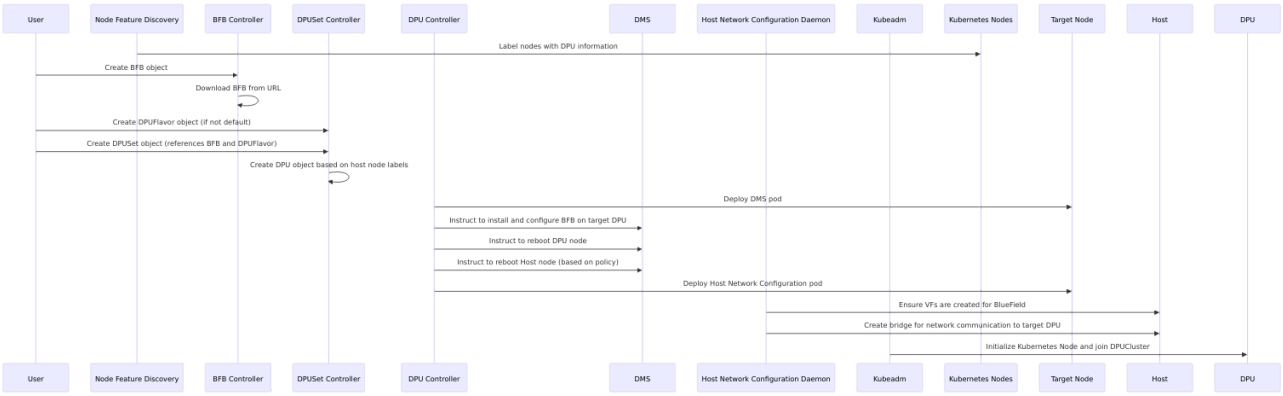
Create a DPU Cluster



This is a prerequisite to provision a DPU with DPF. This flow is based on the Kamaji Cluster Manager - but other Cluster managers may be used.

1. The user creates a DPUCluster object.
2. The DPUCluster manager creates an underlying Kamaji TenantControlPlane.
3. The Kamaji controllers create the cluster control plane pods.
4. The DPUCluster manager creates a load balancer for the Kamaji control plane.
5. The DPUCluster manager updates the DPUCluster with a kubeconfig for the Kamaji control plane.

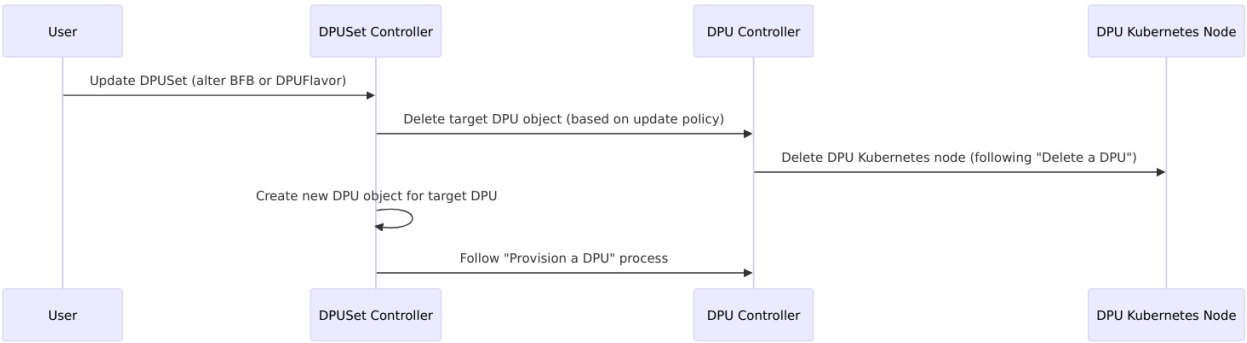
Provision a DPU



1. Node Feature Discovery labels Kubernetes nodes with DPU information.

2. The user creates a BFB object.
3. The BFB controller downloads the BFB from a URL.
4. The user creates a DPUFlavor object if not using a default.
5. The user creates a DPUSet object which references both the BFB and the DPUFlavor.
6. The DPUSet controller creates a DPU object based on host node labels.
7. The DPU controller deploys DMS pod to a target node.
8. The DPU controller instructs DMS to install and configure the BFB on a target DPU.
9. The DPU controller instructs DMS to reboot the DPU node.
10. The DPU controller instructs DMS to reboot the Host node based on reboot policy.
11. The DPU controller deploys the Host Network Configuration pod on a target node.
12. The Host Network Configuration daemon ensures VFs are created on the host for the BlueField.
13. The Host Network Configuration daemon creates a bridge for network communication to the target DPU.
14. Kubeadm initializes a Kubernetes Node on the DPU and joins the DPUCluster.

Update a DPU



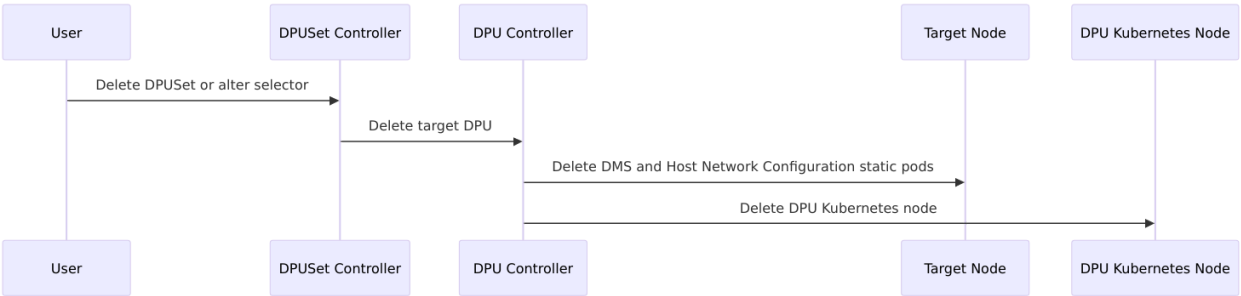
1. User updates the DPUSet altering its the BFB or DPUFlavor.
2. DPUSet controller deletes the target DPU object based on update policy rules.

3. DPU controller deletes the DPU Kubernetes node following `Delete a DPU`.

4. DPUSet controller creates a new DPU object for the target DPU.

5. The process in `Provision a DPU` is followed to create a new DPU node with the updated DPU system.

Delete a DPU



1. User deletes the DPUSet or alters the selector to reduce the number of DPU objects.

2. DPUSet controller deletes target DPU.

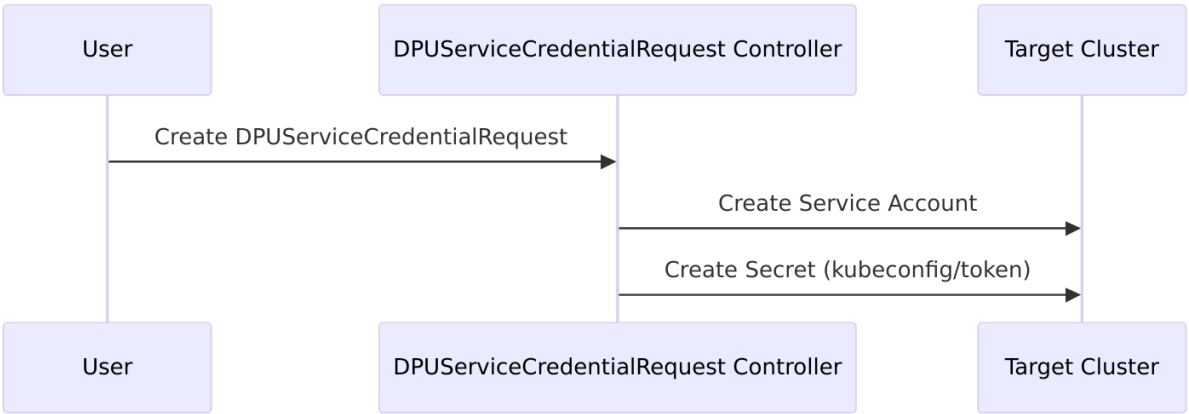
3. DPU controller deletes the DMS and Host Network Configuration static pods.

4. DPU controller deletes the DPU Kubernetes node.

DPUService flow

DPUService orchestration has four principle user flows.

Create a DPUServiceCredentialRequest



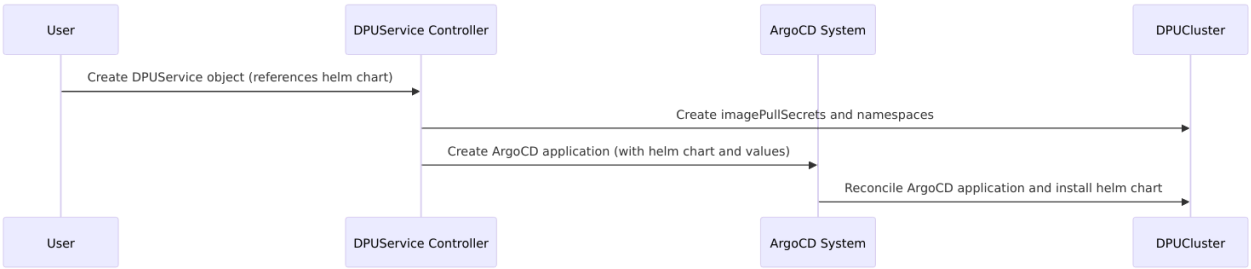
This is a prerequisite for DPUServices that are deployed in the host control plane but communicate with the DPU control plane and vice versa. Users must modify their DPUService to use the secret created by the DPUServiceCredentialRequest.

1. User creates a DPUServiceCredentialRequest

2. DPUServiceCredentialRequest controller creates a service account in the target cluster.

3. DPUServiceCredentialRequest controller creates a secret with either a kubeconfig or token for the service account.

Create DPUService



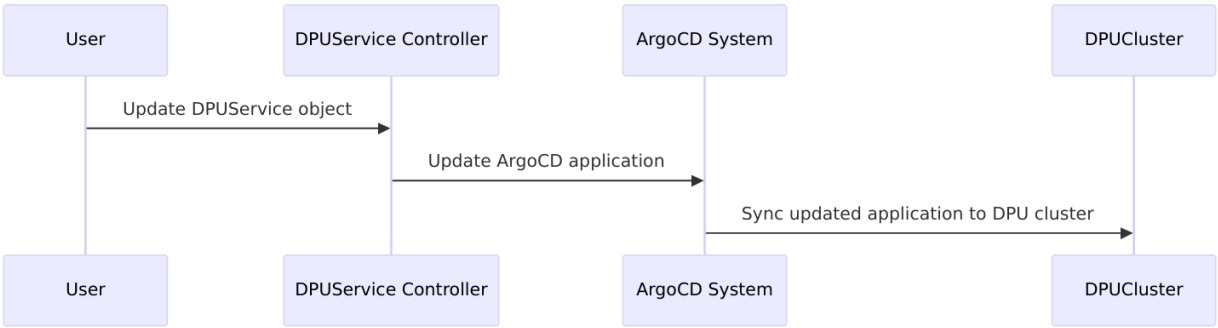
1. User creates a DPUService object referencing a helm chart in a repository.

2. DPUService controller creates associated imagePullSecrets and namespaces in the DPUCluster.

3. DPUService controller creates an ArgoCD application with the DPUService helm chart and values.

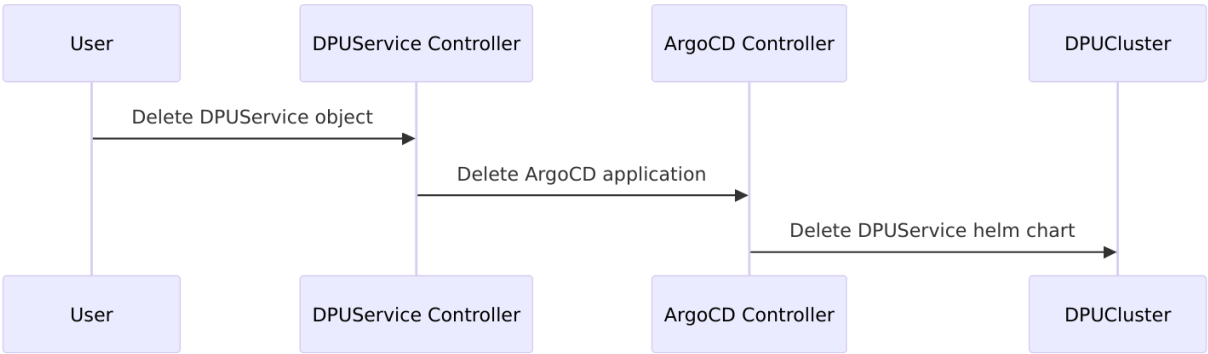
4. ArgoCD system reconciles the ArgoCD application and installs the helm chart to the DPU cluster.

Update DPUService



1. User updates the DPUService object.
2. DPUService controller updates the ArgoCD application.
3. ArgoCD system components sync the updated application to the DPU cluster.

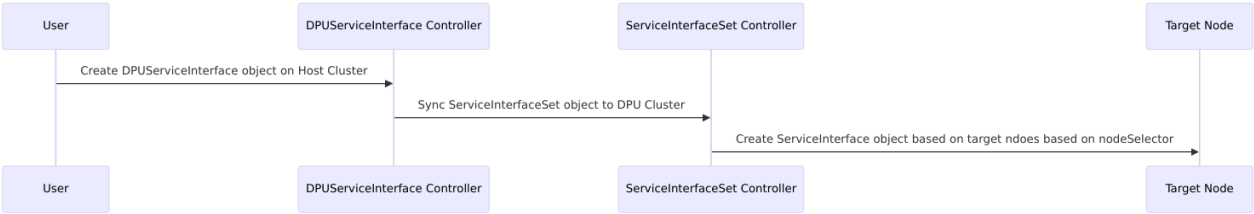
Delete DPUServices



1. User deletes the DPUService object.
2. DPUService controller deletes the ArgoCD application.
3. ArgoCD controller deletes the DPUService helm chart in the DPU cluster.

TODO: DPUDeployment flow

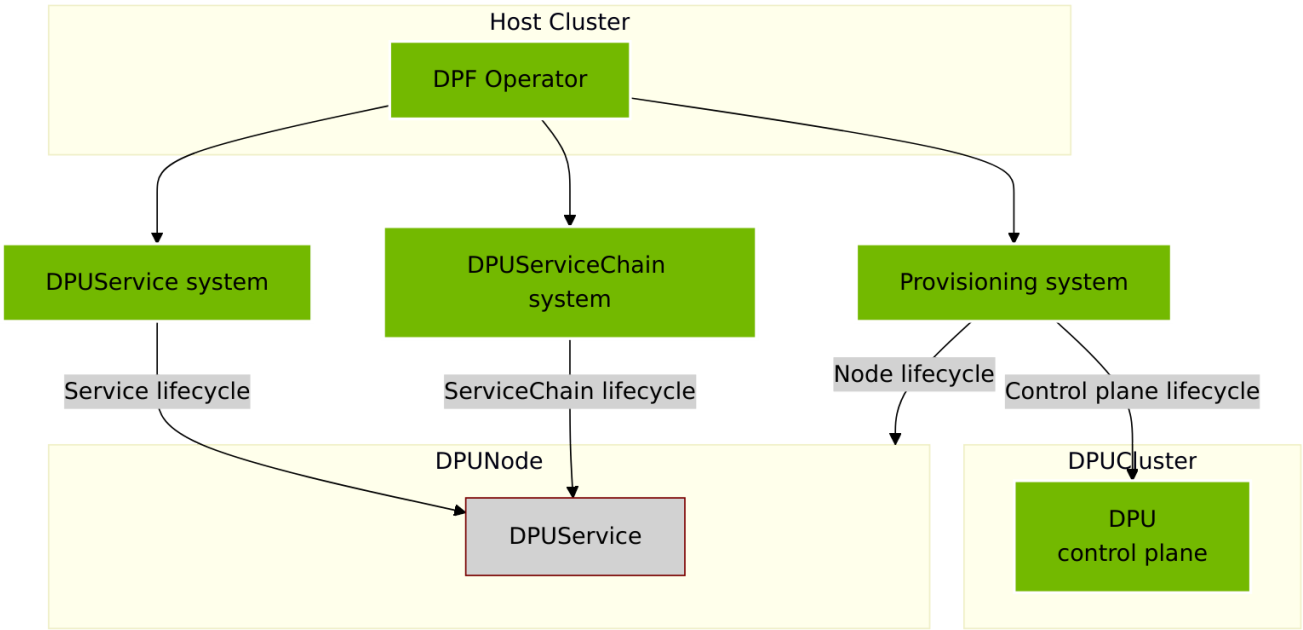
DPUServiceInterface flow



1. User creates the DPUServiceInterface object on the host cluster.
2. DPUServiceInterface controller creates the ServiceInterfaceSet object and syncs it to the DPU cluster.
3. ServiceInterfaceSet controller creates the ServiceInterface object for individual nodes based on the nodeSelector.
4. User refers to the DPUServiceInterface in DPUService to make actual use of it.

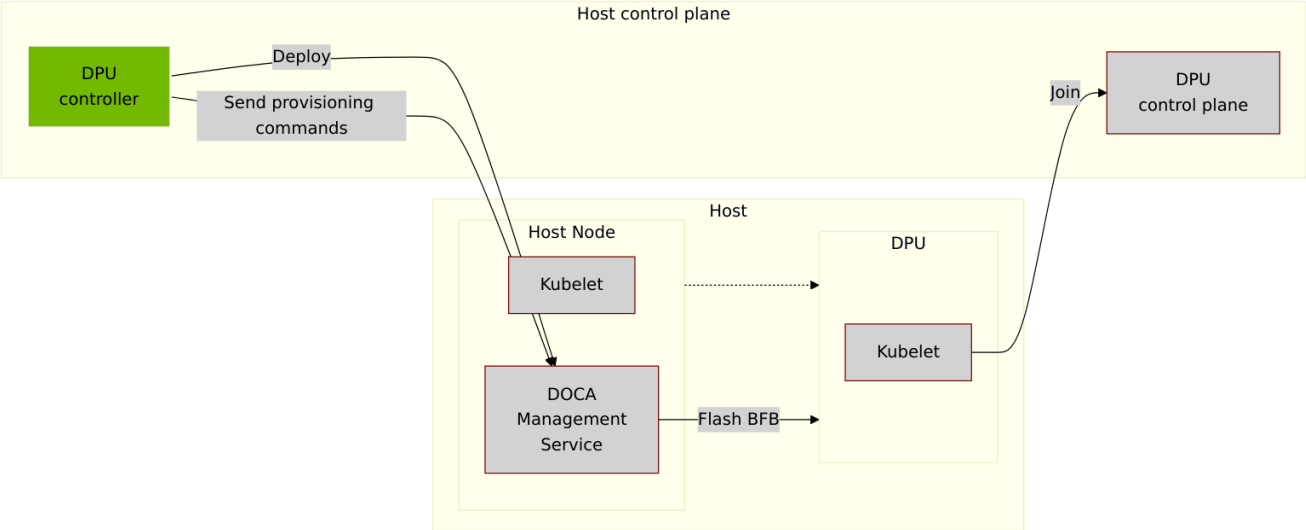
Overview

DOCA Platform Framework (DPF) streamlines the provisioning and orchestration of NVIDIA BlueField DPUs in Kubernetes environments. It supports scalable deployment and management of DPUs and the services running on them.



Provisioning system

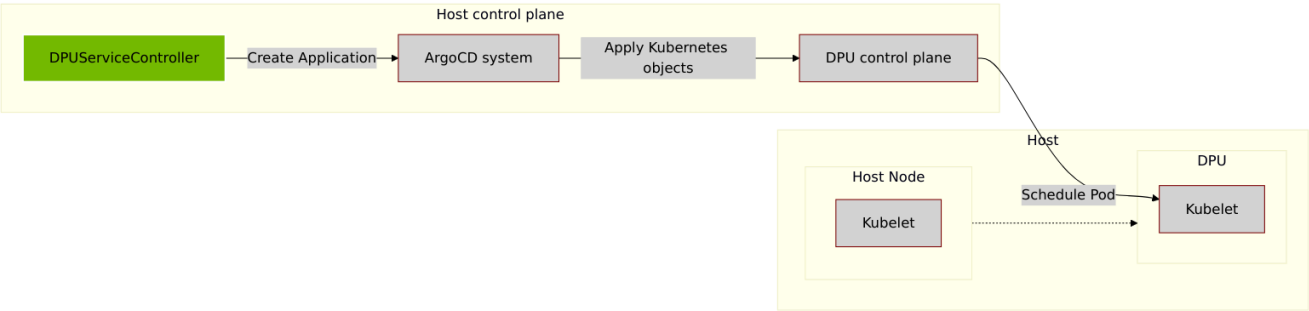
Manage the lifecycle of the DPU hardware and its Kubernetes orchestration system.



1. User creates a `DPUSet` and `BFB` object representing a set of DPUs they want to run services on.
2. DPUSet controller creates a `DPU` object based on the DPUSet NodeSelector.
3. DPU controller deploys DMS and communicates the BFB and other configuration to it.
4. DMS installs BFB to the DPU over rshim.
5. The DPU node joins the DPU control plane.

DPUService system

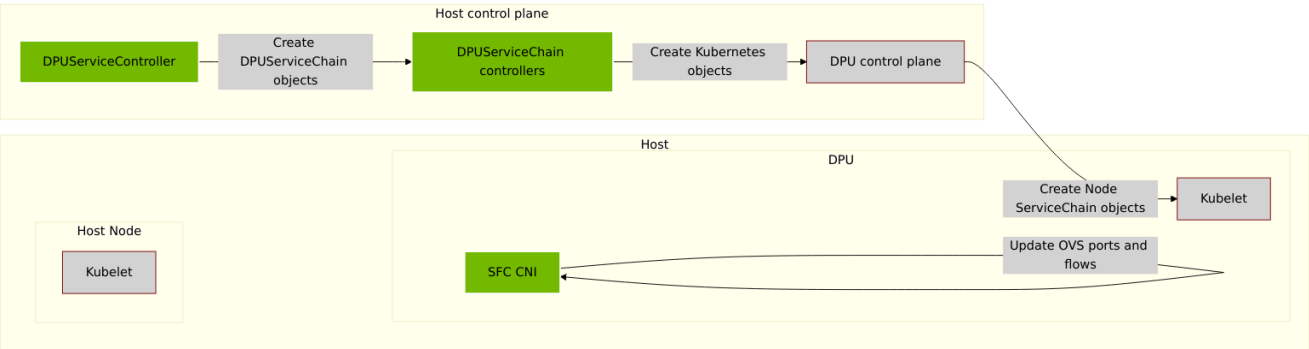
Orchestrate DOCA Services to run on DPUs.



1. User creates a `DPUService` referencing a helm chart.
2. DPUService controller creates an ArgoCD Application representing the DPUService.
3. ArgoCD system syncs the Application to the DPU Cluster.
4. A Pod for the `DPUService` runs on the DPU Node.

DPUServiceChain system

Orchestrate Service Chains for advanced network flows through DPUs.

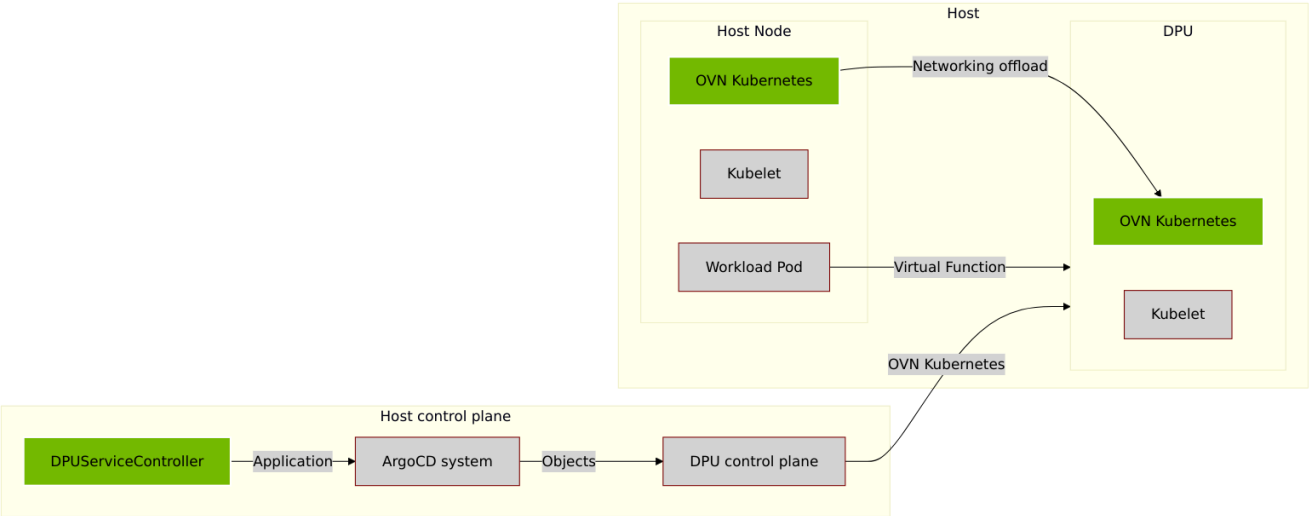


1. User creates `DPUServiceInterface`, `DPUServiceChain`, `DPUServiceIPAM` representing ovs ports, ovs flows and CNI IPAM.
2. Service Function Chain controllers sync these objects to the DPUCluster.
3. ServiceInterface controller creates OVS ports for the Service Function Chain
4. ServiceChain controller creates OVS flows for the Service Function Chain
5. NVIPAM allocates an IP Pool for the Service Function Chain

6. User creates a DPUService for the ServiceFunctionChain

OVN Kubernetes DPUService

Offload primary and secondary network to the DPU



1. User deploys OVN Kubernetes as the primary CNI
2. User creates OVN Kubernetes `DPUService`
3. DPUService controller and ArgoCD sync the OVNKubernetes DPUService to the DPUNode
4. OVN is the primary network which links workload pods with the ServiceFunctionChains operating on the DPU node
5. All pods get a VF from the Bluefield which is used for the primary network
6. OVN processing offloaded to DPU
7. Other network functions - like an BGP router and L3 firewall using HBN - can be added to the service chain for a pod.

Services

This section provides comprehensive guides for developing and extending DPU services within the DOCA Platform Framework (DPF).

Service Documentation Files

- DPUService Development Guide - Complete guide to creating DPU services, including architecture, implementation patterns, and best practices
- Storage Development Guide - Building storage services for DPU environments, covering storage protocols, performance optimization, and integration patterns

DPUService Development Guide

- Additional information about supported features
  - Version Constraints

The DOCA Platform Framework (DPF) manages application lifecycles through DPUService resources. A DPUService typically represents an application that runs on NVIDIA BlueField DPUs and utilizes the DOCA SDK.

DOCA provides libraries for networking and data processing programmability that leverage NVIDIA BlueField DPU hardware accelerators. For more information about developing DOCA applications see the DOCA SDK programming guide.

A DPUService consists of the following components:

- One or more applications designed to run on the DPU
- One or more container images that include the application and its runtime environment
- A Helm chart that defines the Kubernetes manifests required to run the application

Note: An example DPUService is available at github.com/nvidia/doca-platform under `dpuservices/dummydpuservice`. This example is a good base to start from when developing your own DPUServices.

Application

A DPUService application:

- can be written in any language and serve any function.
- must run in a containerized environment.

- must run on `arm64`.

The DPUService API is flexible, and some implementations may not fully conform to the constraints described in this document. For example, some DPUServices are deployed on clusters other than the DPUCluster using the `.spec.deployInCluster` parameter. Services deployed in this manner must match the architecture of their target cluster.

Container image

A DPUService container image must:

- be available on a container registry which is accessible from the DPU.
- run on `arm64` architecture.

DPUService Helm chart

A DPUService Helm chart must:

- be available on a Helm registry which is accessible from the DPF control plane.
- contain exactly one DaemonSet - the `serviceDaemonSet` - that is part of a service function chain.
- Each Kubernetes object must have a `{{ .Release.Name }}` in their `.metadata.name`. This enables deployment of multiple instances of the service.
- Include version constraints in the Helm chart annotations - e.g. `dpu.nvidia.com/doca-version` for DOCA version.
- Implement certain parameters

DPUService developers may implement additional parameters in the Helm chart to meet their needs.

Helm chart parameters

A DPUService Helm chart must expose the following parameters which can be set using Helm values:

Parameter	Type	Optional	Descr
`serviceDaemonSet.nodeSelector`	nodeSelector	No	Node `node`
`serviceDaemonSet.labels`	map[string]string	No	servic
`serviceDaemonSet.annotations`	map[string]string	No	servic
`serviceDaemonSet.upgradeStrategy`	daemonSetUpdateStrategy	No	servic
`serviceDaemonSet.resources`	map[string]string	No	servic
`exposedPorts.ports.<portName>`	boolean	Yes	Enabl
`exposedPorts.labels`	map[string]string	Yes*	Custo contr

Exposing ports

DPUServices can optionally expose ports to the outside world. Exposing these ports is enabled by setting the `exposedPorts` parameter in the Helm chart. This parameter is optional for chart implementers.

- `exposedPorts.ports` parameter is a map of port names to boolean values.
- By default, all ports should be disabled (`false`)
- Each port is configurable individually. Users can enable exposure by setting `exposedPorts.ports.<portName>=true`.
- Custom labels must be accepted under `exposedPorts.labels` and will be applied to the relevant resources. This is necessary for the `Service` to be discovered by the DPUService controller.
- The Helm chart contains a `Service` of type `NodePort`.
- Both `internalTrafficPolicy` and `externalTrafficPolicy` must be set to `Local` to ensure correct routing behavior.

Recommendations

The following list contains some recommended features for the DPUService Helm chart. If the chart developer doesn't implement these, users may face issues deploying the chart in e.g. air-gapped or large scale environments.

- ImagePullSecrets for pod controllers - e.g. DaemonSets, Deployments should be configurable.
- CRDs should be placed in the crd folder of the Helm chart.
- If the application needs to access resources in a different cluster than the one the application is deployed in, the chart should allow for configurable `volumes`, `volumeMounts` and `env` so that a DPUServiceCredentialRequest can be used.

Developing a DPUService

This guide requires working knowledge of Docker, Kubernetes and Helm.

Prerequisites

- a docker build environment
- `docker` CLI installed - see the guide
- `helm` CLI installed - see the guide
- `dpfctl` CLI installed - see the guide
- `kubectl` CLI installed - see the guide
- The repo `github.com/nvidia/doca-platform` cloned to `$PATH_TO_ROOT_OF_DPF_REPO`
- A docker registry - `$DPUSERVICE_DOCKER_REGISTRY` you can push images to.
- A Helm registry - `$DPUSERVICE_HELM_REGISTRY` you can push Helm charts to.

1. Choose an application to run as a DPUService

In this example we will use the `dummydpuservice` which is implemented as part of this repository. The `dummydpuservice` is an application - written in `Go` - which exposes an http server that returns information about its runtime environment. The implementation for this application is at `$PATH_TO_ROOT_OF_DPF_REPO/cmd/dummydpuservice/main.go`.

2. Containerize the application so it can be deployed and run on a BlueField DPU

Every DPUService has a container which runs the application on the DPU. Developers can define a container build process that meets the requirements of their DPUService as long as the basic requirements.

Writing the Dockerfile

For the `dummydpuservice` we need to use go to build a binary. This application has no runtime dependencies so we can use a minimal `distroless` base image.

This results in the following `Dockerfile`.

```
# Golang base image required to build the dummydpuservice.
FROM docker.io/library/golang:1.24 AS builder

WORKDIR /workspace

# Copy the go source from the DPF repo
COPY ./ ./

# Build the dummydpuservice binary.
RUN CGO_ENABLED=0 GOOS=linux GOARCH=arm64 go build -trimpath -o
dummydpuservice cmd/dummydpuservice/main.go

# Copy the binary to a clean base image.
FROM nvcr.io/nvidia/doca/dpf_containers:1.0.2-ubuntu22.04-
distroless
WORKDIR /
COPY --from=builder /workspace/manager .
USER 65532:65532

ENTRYPOINT ["/dummydpuservice"]
```

Building the image

Save this file to `Dockerfile.dummydpuservice` and run:

```
# Build the container image
docker build --platform linux/arm64 \
  -f Dockerfile.dummydpuservice \
  -t $DPUSERVICE_DOCKER_REGISTRY/dummydpuservice:v0.1.0 \
  $PATH_TO_ROOT_OF_DPF_REPO
```

Note: in order to build this image you need to have an environment capable of building `arm64` container images. For more information see the Docker guide.

Publishing the image

Publish the image to your docker registry:

```
docker tag dummydpuservice:latest
$DPUSERVICE_DOCKER_REGISTRY/dummydpuservice:v0.1.0
docker push $DPUSERVICE_DOCKER_REGISTRY/dummydpuservice:v0.1.0
```

3. Create a Helm chart for the application

A helm chart is a set of templates which are used to create Kubernetes objects. For information about building helm charts see the official documentation.

Each DPUService has its own Kubernetes objects which will differ from application to application.

The `dummydpuservice` chart requires:

- A DaemonSet which will run the application on each DPU
- A Service which will serve requests to the application

Writing the chart

The `dummydpuservice` chart is already developed and located at `$PATH_TO_ROOT_OF_DPF_REPO/dpuservices/dummydpuservice/chart`. This helm chart is an excellent base to start with when developing a new DPUService.

Validating the chart

The dpuservice values can be validated with:

```
helm lint
$PATH_TO_ROOT_OF_DPF_REPO/dpuservices/dummydpuservice/chart
```

This will validate the `dummydpuservice` chart against the `$PATH_TO_ROOT_OF_DPF_REPO/dpuservices/dummydpuservice/chart/values.sc` schema. This schema can be copied to any DPUService to ensure that the values are valid.

Publishing the chart

A DPUService Helm chart must be available on a Helm registry which is accessible from the DPF control plane.

To package and publish the chart:

```
helm package
$PATH_TO_ROOT_OF_DPF_REPO/dpuservices/dummydpuservice/chart --
version v0.1.0
helm push dummydpuservice-chart-0.1.0.tgz
$DPUSERVICE_HELM_REGISTRY
```

4. Deploy your DPUService

This step requires a working DPF installation. For information on how to set up DPF see the user guides.

A DPUService is a Kubernetes object which has a helm chart reference and a set of configuration values.

Writing the DPUService

For the `dummydpuservice` the DPUService will have the form:

```
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUService
metadata:
  name: dummydpuservice
  namespace: dummydpuservice
spec:
  helmChart:
    source:
      repoURL: https://$DPUSERVICE_HELM_REGISTRY/dpf
      version: v0.1.0
      chart: dummydpuservice-chart
    values:
      image:
        tag: v0.1.0
        repository: $DPUSERVICE_DOCKER_REGISTRY/dummydpuservice
  serviceID: dummydpuservice
  serviceDaemonSet:
    nodeSelector:
      nodeSelectorTerms:
      - matchExpressions:
          - key: dpu
            operator: In
            values:
              - dpu
    updateStrategy:
      type: RollingUpdate
      rollingUpdate:
        maxUnavailable: 2
    labels:
      dpuservice.dpu.nvidia.com/name: dummydpuservice
    annotations:
      dpuservice.dpu.nvidia.com/name: dummydpuservice
    resources:
      nvidia.com/bf_sf: 1
  configPorts:
```

```
    serviceType: NodePort
    ports:
      - name: exampletcp
        port: 8080
        protocol: TCP
      - name: exampleudp
        port: 8081
        protocol: UDP
```

You can deploy your DPUService by running the following command:

```
kubectl apply -f $PATH_TO_DPU_SERVICE_YAML
```

Checking DPUService health

You can monitor the health of your DPUService with:

```
dpfctl describe dpuservices --show-
conditions=dpuservice/dummydpuservice
```

Additional information about supported features

Version Constraints

The version constraints can be defined by setting specific annotations as part of the `Chart.yaml` file of the chart.

```
apiVersion: v2
name: dummydpuservice-chart
description: Dummydpuservice chart for Kubernetes
type: application
version: 0.1.0
appVersion: "0.1.0"
annotations:
  # Annotation to track the DOCA version compatibility
  # of a DPUService. This is a requirement for services
  # which implement synchronized upgrades using DPUDeployments.
  dpu.nvidia.com/doca-version: ">= 2.9"
```

The constraints must comply with https://github.com/Masterminds/semver/blob/v3.3.1/README.md?plain=1#L117-L212

There is a specific set of annotations that are handled today, the rest are stripped:

- `dpu.nvidia.com/doca-version`

Storage Development Guide

The DPF Storage Subsystem provides a framework for integrating 3rd-party storage plugins with the DPF system. This document outlines the architecture and guidelines for developing a plugin that integrates with the NVIDIA storage emulation service, called DOCA SNAP.

Architecture

The DPF storage subsystem consists of two types of components: core components provided by NVIDIA and vendor-specific components provided by storage vendors.

Core components are included in the DPF system release and typically require no modifications from end users.

The DPF storage subsystem supports two deployment scenarios:

- Kubernetes cluster on Host Trusted mode
- Zero Trust mode

The deployed components, available features, and APIs differ in each scenario.

Kubernetes cluster on Host Trusted mode

In this scenario, hosts function as worker nodes within the DPF management cluster. Users can utilize Kubernetes Storage APIs (StorageClass, PVC, PV, VolumeAttachment) to provision and attach storage to the host. The hosts run the SNAP CSI Plugin, which performs all necessary actions to make storage resources available to the host.

In this scenario, the following emulation methods are supported: * NVMe over VF on top of a static PF * NVMe over hot-plugged PF * Virtio-FS over hot-plugged PF

The list below contains the components that are deployed in this scenario.

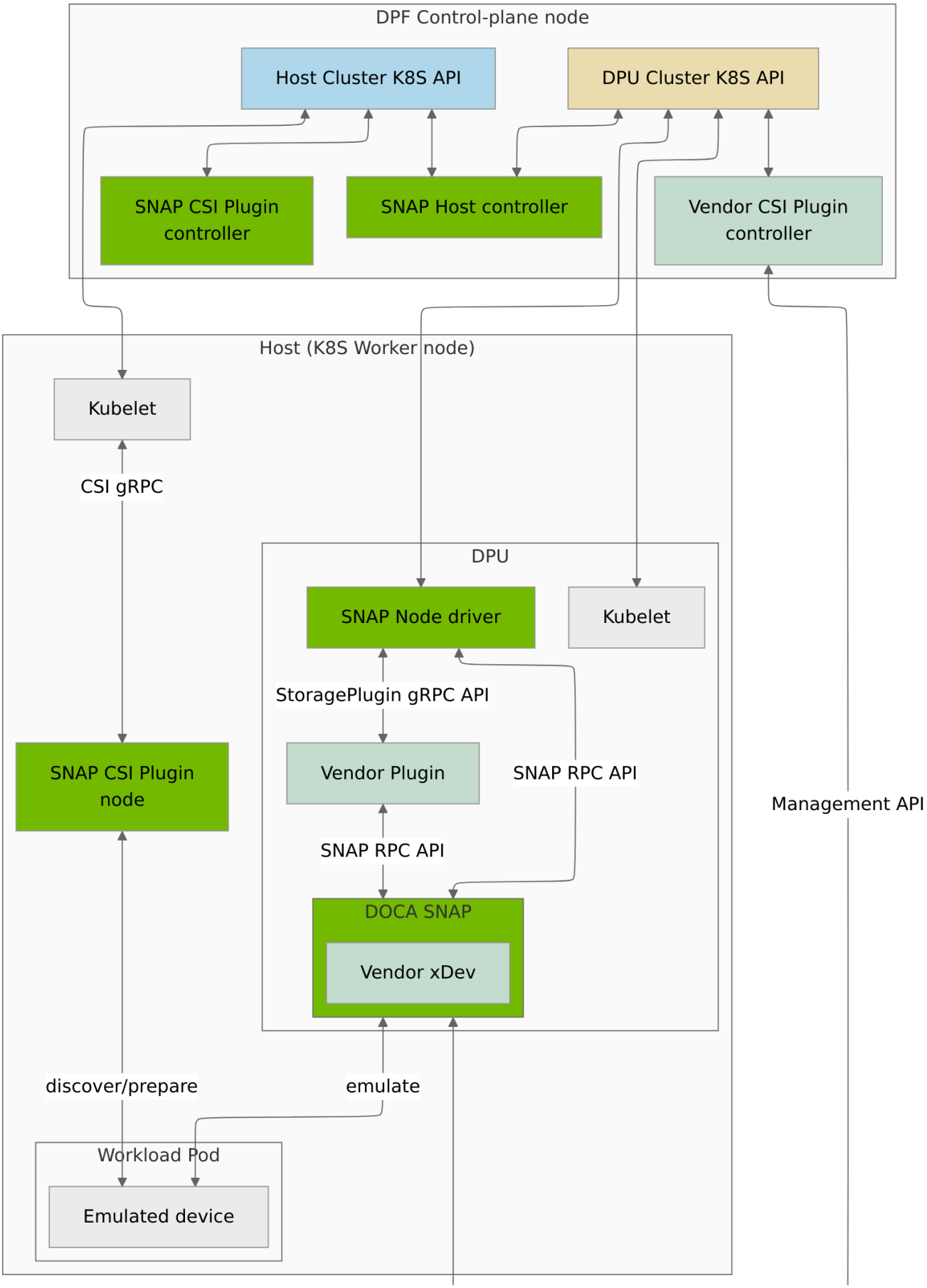
The core components are:

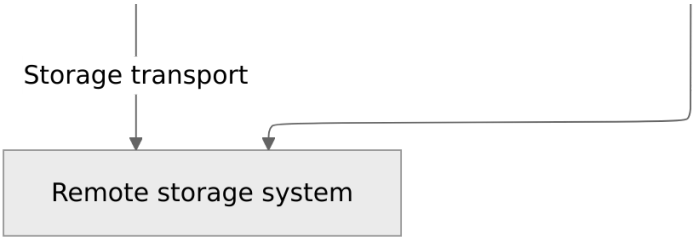
- SNAP CSI Plugin
- SNAP Host Controller
- SNAP Node Driver
- DOCA SNAP

Vendor-specific components are:

- Vendor CSI Plugin Controller
- Vendor Plugin (optional)
- Vendor xDev (optional)

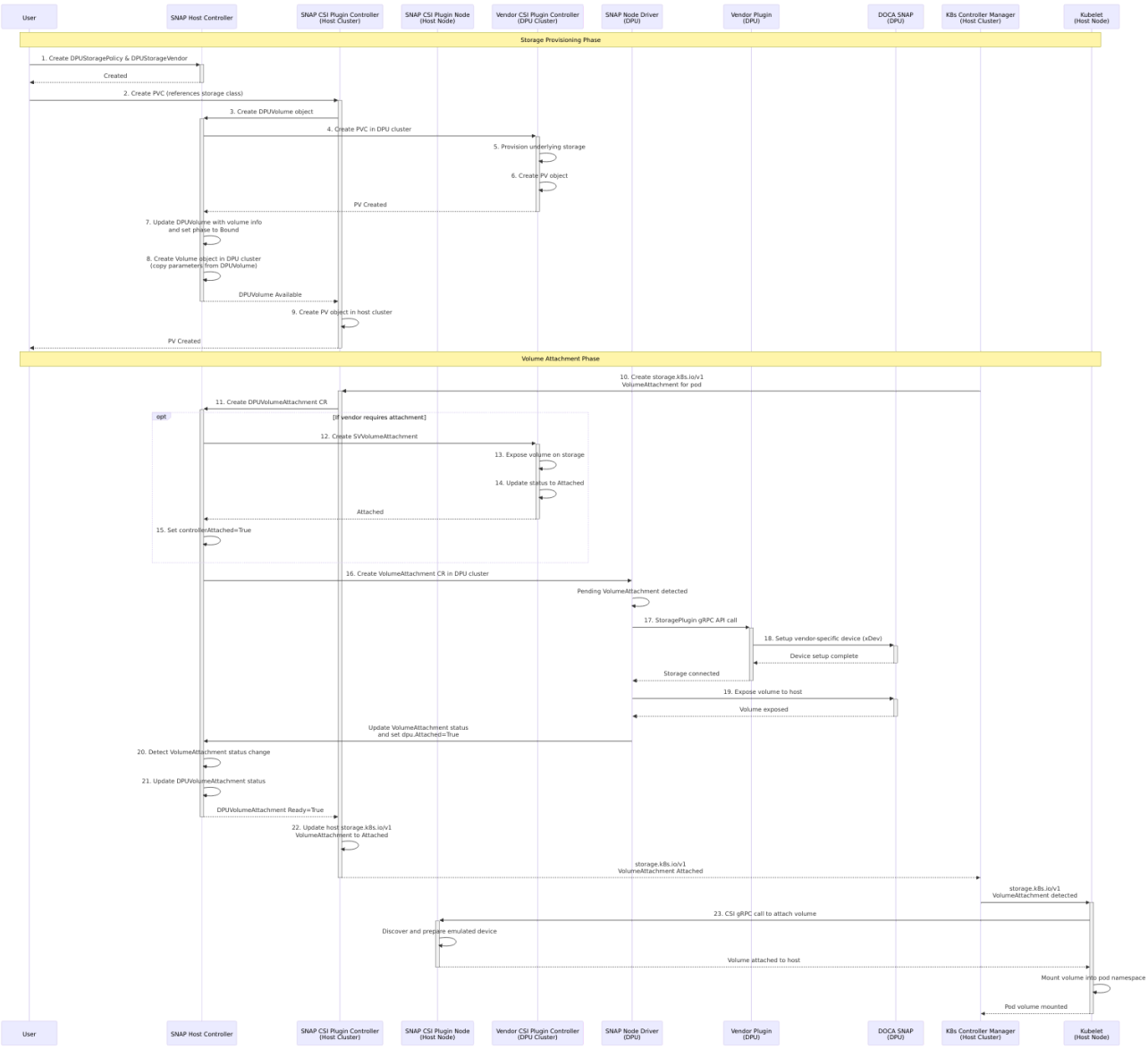
The high-level architecture of the storage subsystem for this scenario is presented in the following diagram.


DPF Control-plane node
Host Cluster K8S API
DPU Cluster K8S API
SNAP CSI Plugin controller
SNAP Host controller
Vendor CSI Plugin controller
Host (K8S Worker node)
Kubelet
CSI gRPC
DPU
SNAP Node driver
Kubelet
StoragePlugin gRPC API
SNAP CSI Plugin node
Vendor Plugin
SNAP RPC API
Management API
SNAP RPC API
DOCA SNAP
Vendor xDev
discover/prepare
emulate
Workload Pod
Emulated device



End-to-End Flow for Trusted Kubernetes cluster on host scenario

The following steps outline the end-to-end process for provisioning and attaching storage using the DPF storage subsystem in the Trusted Kubernetes cluster on host scenario:



1. **DPUStoragePolicy and DPUStorageVendor Creation**: The user creates a DPUStoragePolicy and DPUStorageVendor object in the host cluster.

2. **PVC Creation**: The user creates a **PersistentVolumeClaim (PVC)** object in the host cluster. The **PVC** references a storage class that specifies the SNAP CSI Plugin as its provisioner. The storage class contains parameters that specify the name of a specific DPUStoragePolicy.

3. **DPUVolume Object Creation**: The SNAP CSI Plugin Controller in the host cluster handles **PVC** creation and creates a DPUVolume object in the host cluster. This object includes references to the DPUStoragePolicy and the requested volume parameters from the storage class.

4. **Storage Vendor PVC Creation**: The SNAP Host Controller merges parameters from the DPUStoragePolicy and DPUVolume, selects the appropriate DPUStorageVendor, and creates a **PVC** in the DPU cluster that references the storage class of the Vendor CSI Plugin Controller.

5. **Vendor Storage Provisioning**: The Vendor CSI Plugin Controller detects the new **PVC** in the DPU cluster and provisions the underlying storage.

6. **Vendor PV Creation**: The Vendor CSI Plugin Controller creates the corresponding **PersistentVolume (PV)** object.

7. **DPUVolume Availability Update**: The SNAP Host Controller detects that the **PV** is created and updates DPUVolume object in the host cluster with information about the created volume and sets the phase of the object to **Bound**.

8. **Volume Object Creation**: The SNAP Host Controller creates a Volume object in the DPU cluster. The controller copies parameters from the DPUVolume object to the Volume object.

9. **PV Object Creation**: The SNAP CSI Plugin Controller in the host cluster detects the status change of the DPUVolume object and creates the **PV** object in the host cluster.

10. **Volume Attachment Initiation**: The Kubernetes Controller Manager in the host cluster detects the **PV** object and creates a native Kubernetes **VolumeAttachment [storage.k8s.io/v1]** object to attach the volume to the user's pod.

11. **DPUVolumeAttachment Object Creation**: The SNAP CSI Plugin Controller detects the new native Kubernetes **VolumeAttachment [storage.k8s.io/v1]** object and creates a corresponding DPUVolumeAttachment CR in the host cluster.

12. **SVVolumeAttachment Creation**: The SNAP Host Controller detects the new DPUVolumeAttachment object in the host cluster. If the Vendor CSI Plugin Controller requires attachment, it creates an SVVolumeAttachment object for the Vendor CSI Plugin Controller in the DPU cluster.

13. **Vendor Volume Exposure**: The Vendor CSI Plugin Controller controller detects the new SVVolumeAttachment object and exposes the volume on the underlying storage.

14. **Vendor Volume Attachment**: The Vendor CSI Plugin Controller updates the status of the SVVolumeAttachment object to **Attached**.

15. **DPUVolumeAttachment Status Update**: The SNAP Host Controller detects that SVVolumeAttachment is in **Attached** state and sets the `controllerAttached` of the DPUVolumeAttachment to **True**.

16. **VolumeAttachment Object Creation**: The SNAP Host Controller creates a VolumeAttachment object in the DPU cluster.

17. **Storage Device Connection**: The SNAP Node Driver on the DPU detects the pending VolumeAttachment object and calls the Vendor Plugin via the **StoragePlugin gRPC API** to connect the volume to the underlying storage.

18. **Vendor Plugin Device Setup**: The Vendor Plugin connects the volume to the underlying storage (if required) and sets up the vendor-specific device (xDev) inside the DOCA SNAP service.

19. **SNAP Process Volume Exposure**: The SNAP Node Driver calls the DOCA SNAP service to expose the volume to the host. Upon completion, the SNAP Node Driver updates the DPU parameters in the status of the VolumeAttachment and sets the `dpu.Attached` status to **True**.

20. **VolumeAttachment Status Detection**: The SNAP Host Controller detects the status change of the VolumeAttachment CR in the DPU cluster.

21. **DPUVolumeAttachment Status Update**: The SNAP Host Controller updates the status of the DPUVolumeAttachment object in the host cluster.

22. **Host VolumeAttachment Update**: The SNAP CSI Plugin Controller detects that DPUVolumeAttachment `Ready` Condition is `True` and updates the status of the native Kubernetes **VolumeAttachment [storage.k8s.io/v1]** object on the host cluster to **Attached**.

23. **Pod Volume Mounting**: The kubelet on the host node detects the native Kubernetes **VolumeAttachment [storage.k8s.io/v1]** object with status **Attached**. It calls the SNAP CSI Plugin Node to attach the volume to the host and mounts the volume into the pod's namespace.

Zero Trust mode

In this scenario, hosts are not trusted and the host OS configuration is not managed by DPF components. Standard Kubernetes Storage APIs (StorageClass, PVC, PV, VolumeAttachment) cannot be used in this scenario. Instead, DPF-specific storage APIs must be used to manage storage operations.

There are no DPF storage components deployed on the host. User-provided applications or scripts are responsible for performing the required actions on the host OS to make storage available to the host.

Refer to the DOCA SNAP documentation for detailed information about required host OS configuration.

In this scenario, the following emulation methods are supported: * NVMe over VF on top of a static PF * NVMe over hot-plugged PF * NVMe over static PF * Virtio-FS over hot-plugged PF

The list below contains the components that are deployed in this scenario.

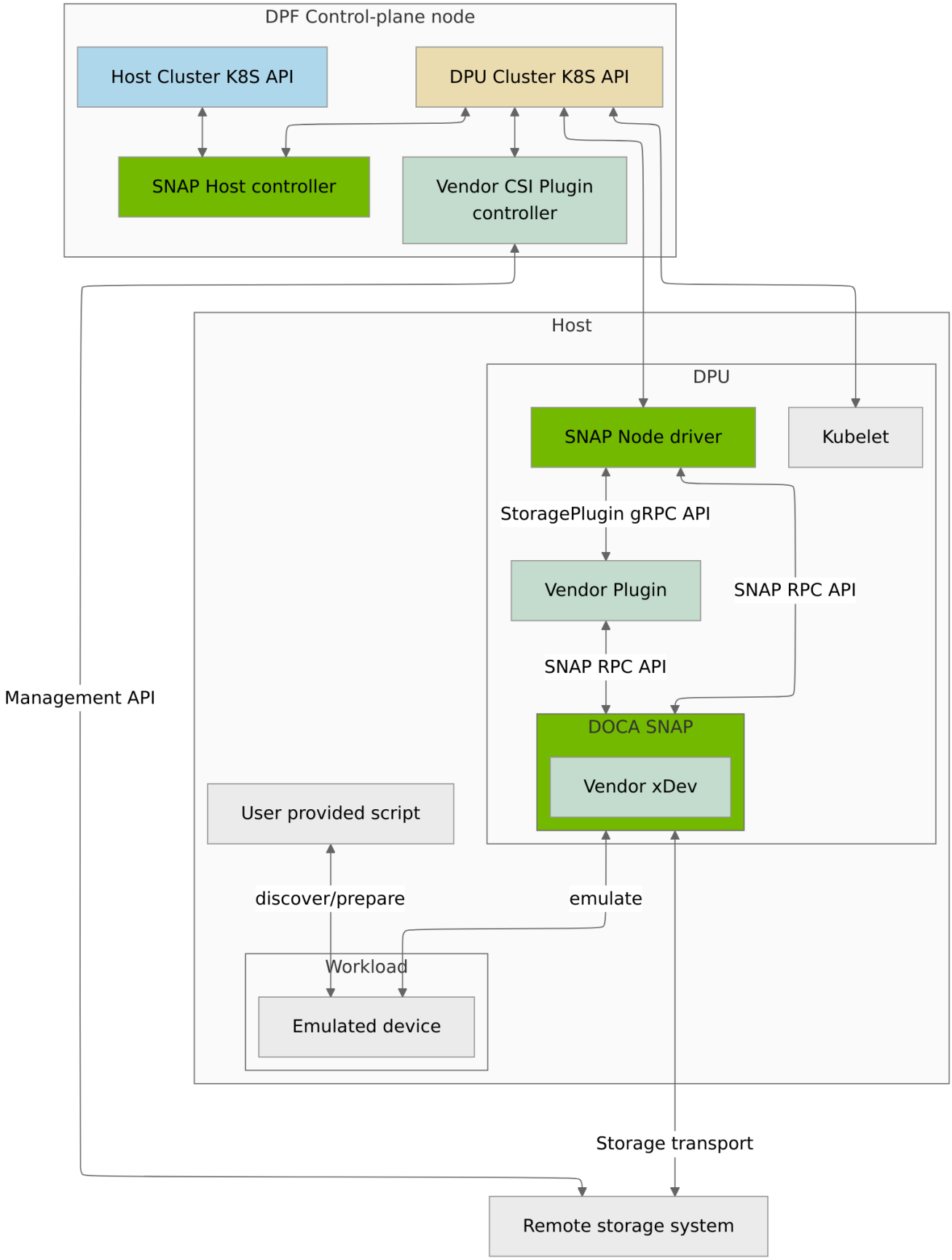
The core components are:

- SNAP Host Controller
- SNAP Node Driver
- DOCA SNAP

Vendor-specific components are:

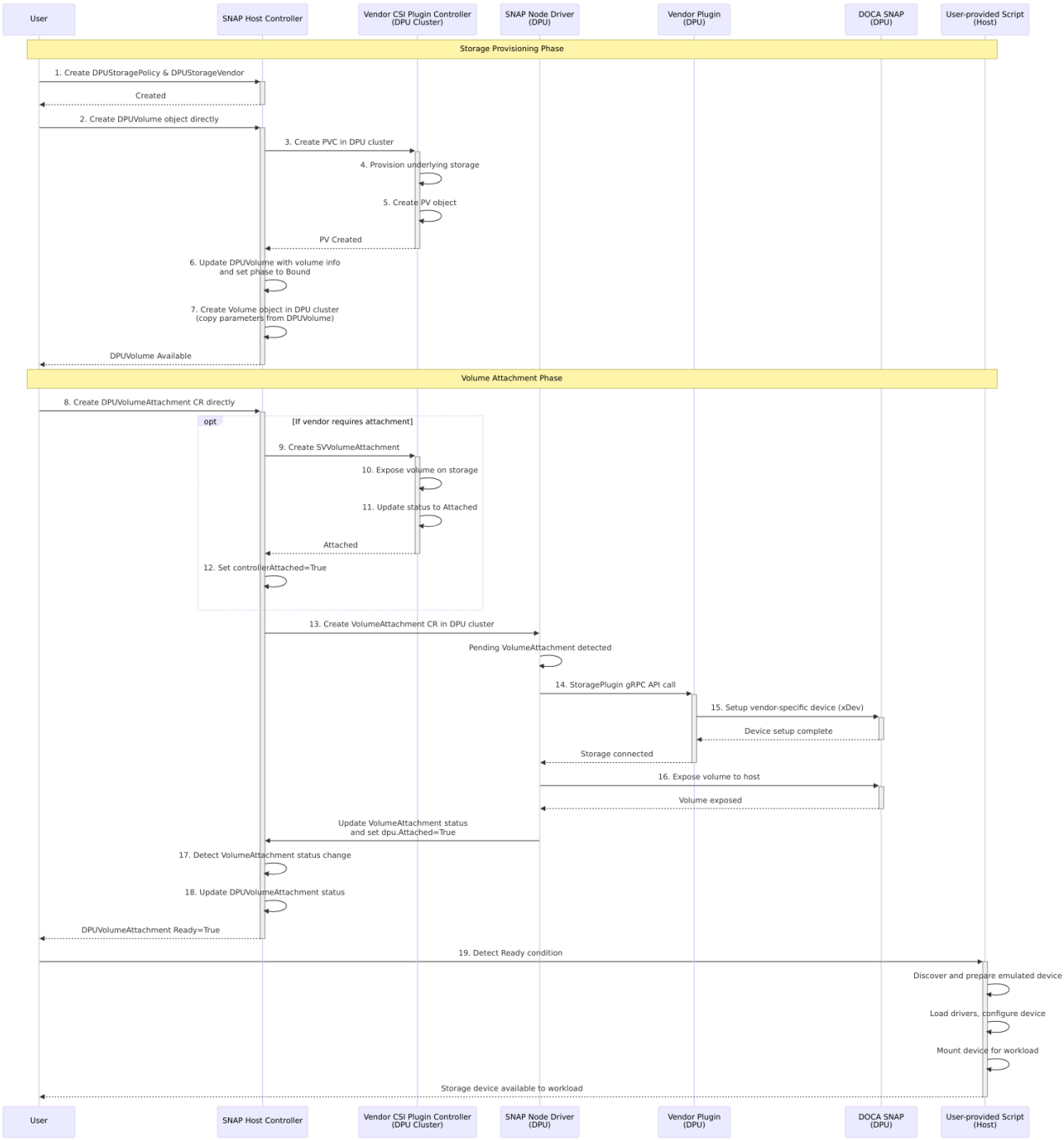
- Vendor CSI Plugin Controller
- Vendor Plugin (optional)
- Vendor xDev (optional)

The high-level architecture of the storage subsystem is presented in the following diagram.


DPF Control-plane node
Host Cluster K8S API
DPU Cluster K8S API
SNAP Host controller
Vendor CSI Plugin controller
Host
DPU
SNAP Node driver
Kubelet
StoragePlugin gRPC API
Vendor Plugin
SNAP RPC API
SNAP RPC API
Management API
DOCA SNAP
Vendor xDev
User provided script
discover/prepare
emulate
Workload
Emulated device
Storage transport
Remote storage system

End-to-End Flow Description for Zero Trust scenario

The following steps outline the end-to-end process for provisioning and attaching storage using the DPF storage subsystem in the Zero Trust scenario:



1. **DPUStoragePolicy and DPUStorageVendor Creation**: The user creates a DPUStoragePolicy and DPUStorageVendor object in the host cluster.

2. **DPUVolume Object Creation**: The user directly creates a DPUVolume object in the host cluster. This object includes references to the DPUStoragePolicy and the requested volume parameters.

3. **Storage Vendor PVC Creation**: The SNAP Host Controller merges parameters from the DPUStoragePolicy and DPUVolume, selects the appropriate DPUStorageVendor, and creates a **PVC** in the DPU cluster that references the storage class of the Vendor CSI Plugin Controller.

4. **Vendor PV Provisioning**: The Vendor CSI Plugin Controller detects the new **PVC** in the DPU cluster, provisions the underlying storage, and creates the corresponding **PersistentVolume (PV)** object.

5. **DPUVolume Availability Update**: The SNAP Host Controller detects that the **PV** is created and updates DPUVolume object in the host cluster with information about the created volume and sets the phase of the object to **Bound**.

6. **Volume Object Creation**: The SNAP Host Controller creates a Volume object in the DPU cluster. The controller copies parameters from the DPUVolume object to the Volume object.

7. **DPUVolumeAttachment Object Creation**: The user directly creates a DPUVolumeAttachment CR in the host cluster to attach the volume to a specific host node.

8. **SVVolumeAttachment Creation**: The SNAP Host Controller detects the new DPUVolumeAttachment object in the host cluster. If the Vendor CSI Plugin Controller requires attachment, it creates an SVVolumeAttachment object for the Vendor CSI Plugin Controller in the DPU cluster.

9. **Vendor Volume Exposure**: The Vendor CSI Plugin Controller detects the new SVVolumeAttachment object and exposes the volume on the underlying storage.

10. **Vendor Volume Attachment**: The Vendor CSI Plugin Controller updates the status of the SVVolumeAttachment object to **Attached**.

11. **DPUVolumeAttachment Status Update**: The SNAP Host Controller detects that SVVolumeAttachment is in **Attached** state and sets the `controllerAttached` of the DPUVolumeAttachment to **True**.

12. **VolumeAttachment Object Creation**: The SNAP Host Controller creates a VolumeAttachment object in the DPU cluster.

13. **Storage Device Connection**: The SNAP Node Driver on the DPU detects the pending VolumeAttachment object and calls the Vendor Plugin via the **StoragePlugin gRPC**

API to connect the volume to the underlying storage.

14. **Vendor Plugin Device Setup**: The Vendor Plugin connects the volume to the underlying storage (if required) and sets up the vendor-specific device (xDev) inside the DOCA SNAP service.

15. **SNAP Process Volume Exposure**: The SNAP Node Driver calls the DOCA SNAP service to expose the volume to the host. Upon completion, the SNAP Node Driver updates the DPU parameters in the status of the VolumeAttachment and sets the `dpu.Attached` status to **True**.

16. **VolumeAttachment Status Detection**: The SNAP Host Controller detects the status change of the VolumeAttachment CR in the DPU cluster.

17. **DPUVolumeAttachment Status Update**: The SNAP Host Controller updates the status of the DPUVolumeAttachment object in the host cluster to indicate the volume is ready.

18. **Ready Condition Detection**: The user-provided script on the host detects that the DPUVolumeAttachment `Ready` condition is `True`.

19. **Host Volume Preparation**: The user-provided script performs the necessary operations to discover, prepare, and make the emulated storage device available to the host workload. This includes any required driver loading, device configuration, and mounting operations.

Core components

SNAP CSI Plugin

> **Note:** This component is deployed only for the trusted Kubernetes cluster on host scenario.

The **SNAP CSI Plugin** is a Kubernetes CSI plugin responsible for managing the lifecycle of storage resources within the host cluster. It enables the nodes within the host cluster to consume storage resources provisioned by the DPF storage subsystem.

The SNAP CSI plugin handles the creation and management of Kubernetes storage resources such as **Storage Classes**, **Persistent Volumes (PV)**, and other Kubernetes Storage objects.

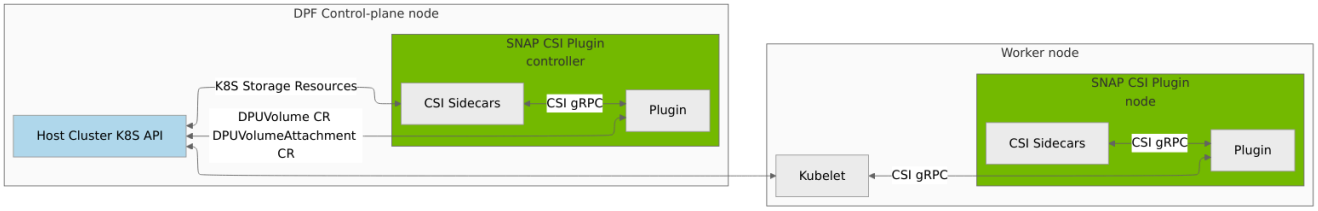
The plugin uses DPF storage APIs by creating DPUVolume and DPUVolumeAttachment custom resources in the host cluster to trigger the creation and attachment of storage resources in the DPU cluster.

The SNAP CSI Plugin consists of a controller and a node component. The controller component is deployed on control-plane nodes, while the node component is deployed on worker nodes. The node component is responsible for discovering and preparing emulated storage devices on the host, and mounting them into Pod namespaces when requested by the kubelet.

The plugin supports two emulation modes:

Emulation Mode	**Supported `volumeMode` for PVCs in host cluster**	**`volumeMode` in created DPUVolume CR**
`nvme`	Block (volume as raw block device in Pod)	Block (volume exposed as emulated NVMe device to the host)
`virtiofs`	Filesystem (volume as filesystem in Pod)	Filesystem (volume exposed as VirtioFS to the host)

Note: The emulation mode is set using the `--emulation-mode` flag.



Example of the StorageClass object:

```yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: snap-nvme
provisioner: csi.snap.nvidia.com
parameters:
  policy: "example-policy"
  functionType: "vf"
  hotplugFunction: "false"
```

The following combinations of `functionType` and `hotplugFunction` StorageClass parameters are supported:

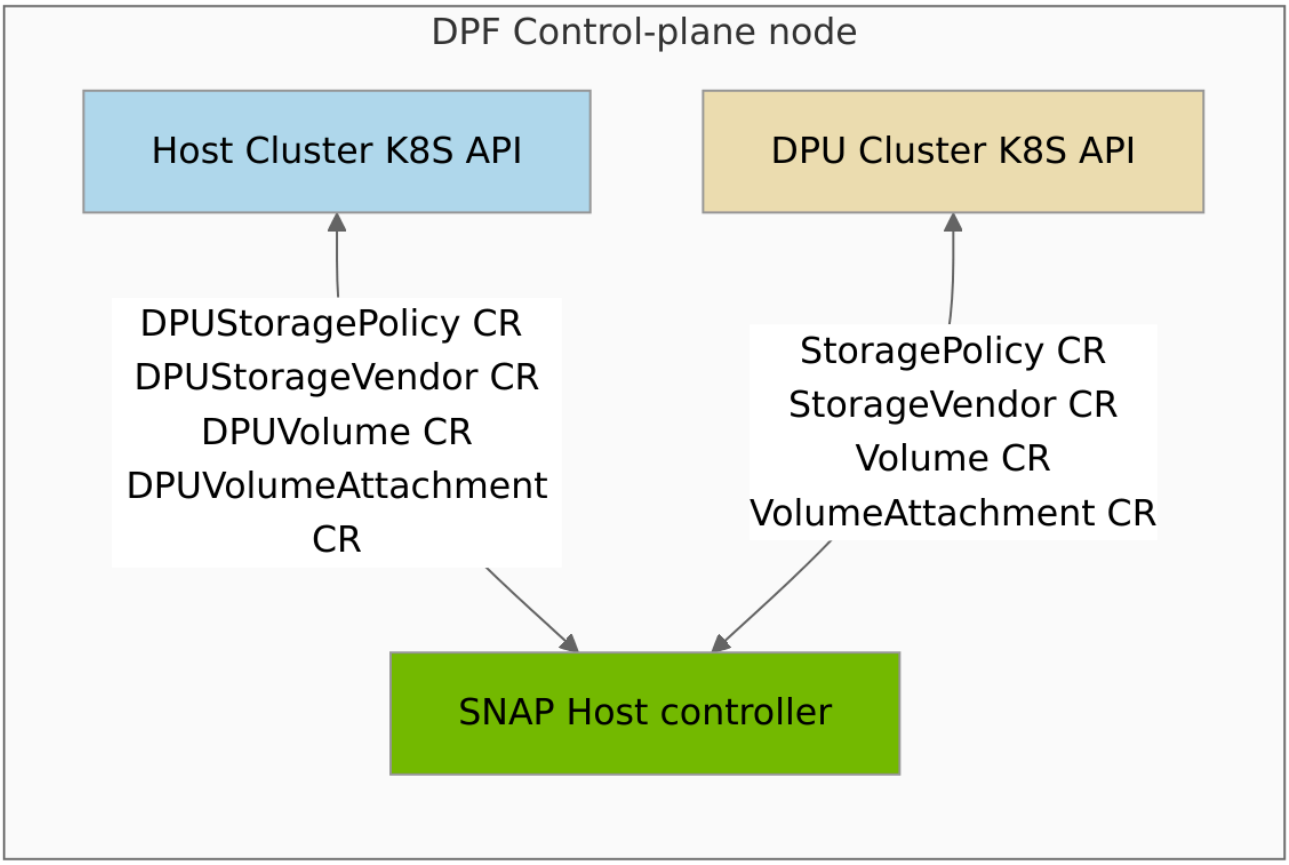
functionType	hotplugFunction	Description	emulationMode=nvme	emulationMode=v
`vf`	`false`	VF on top of static PF	Supported	Not supported
`pf`	`true`	Hot-plugged PF	Supported	Supported
`vf`	`true`	Hot-plugged VF	Not supported	Not supported
`pf`	`false`	Static PF	Not supported	Not supported

SNAP Host Controller

The **SNAP Host Controller** implements the user-facing DPF storage APIs, manages the synchronization of storage resources between the host cluster and the DPU cluster, and implements the core business logic of the DPF storage subsystem. Operating within the DPF control-plane, it:

- Reconciles DPUStoragePolicy and DPUStorageVendor objects in the host cluster.
- Reconciles DPUVolume objects in the host cluster, selects appropriate storage vendors, and creates PVCs directly in the DPU cluster to trigger the Vendor CSI Plugin Controller.

- Creates Volume objects in the DPU cluster with volume information and parameters copied from DPUVolume objects.
- Reconciles DPUVolumeAttachment objects in the host cluster and creates corresponding VolumeAttachment objects in the DPU cluster to trigger volume attachment operations.
- Creates SVVolumeAttachment custom resources when the Vendor CSI Plugin Controller requires ControllerPublishVolume/ControllerUnpublishVolume operations.
- Monitors the status of storage resources in the DPU cluster and propagates status updates back to the corresponding resources in the host cluster

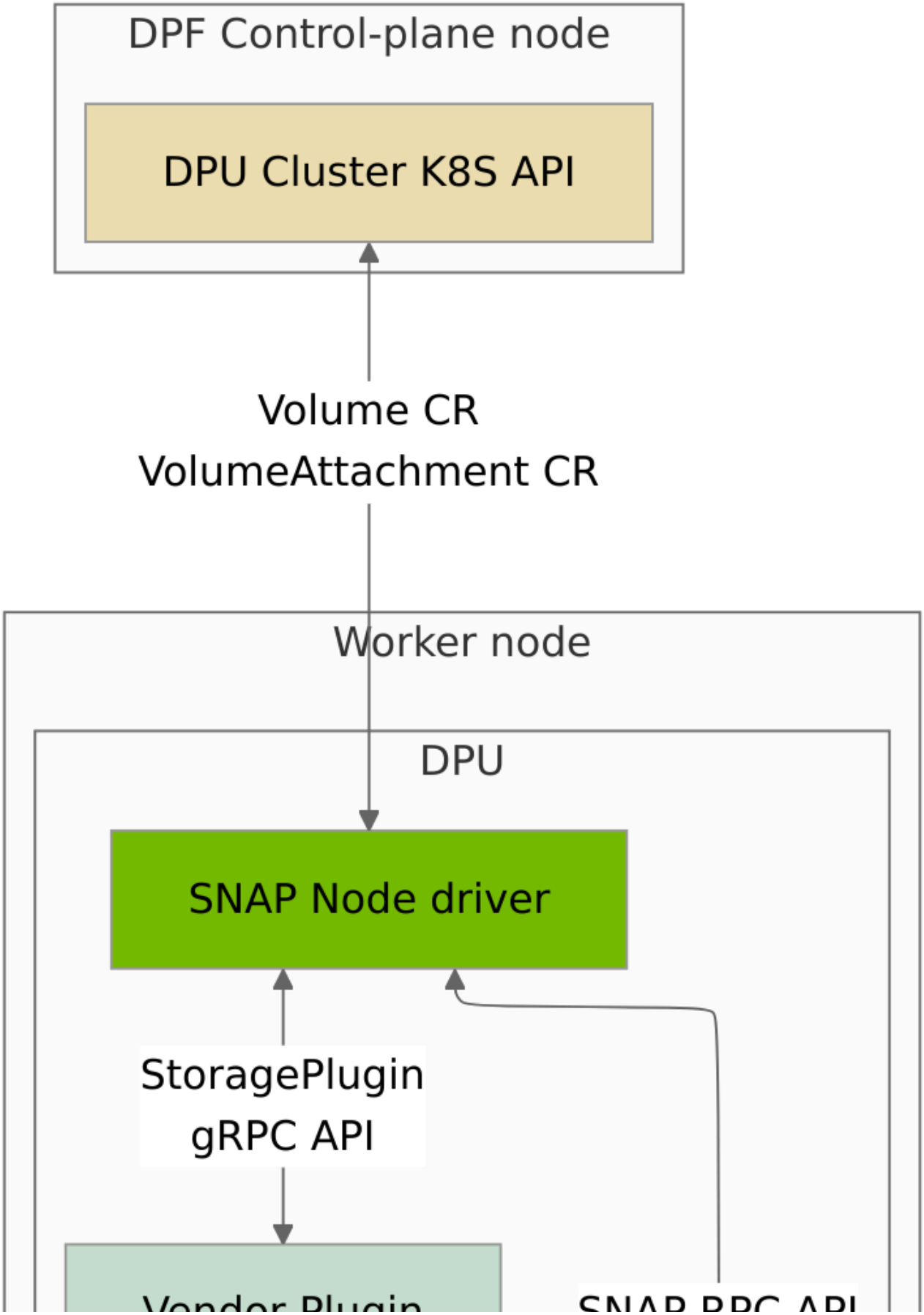


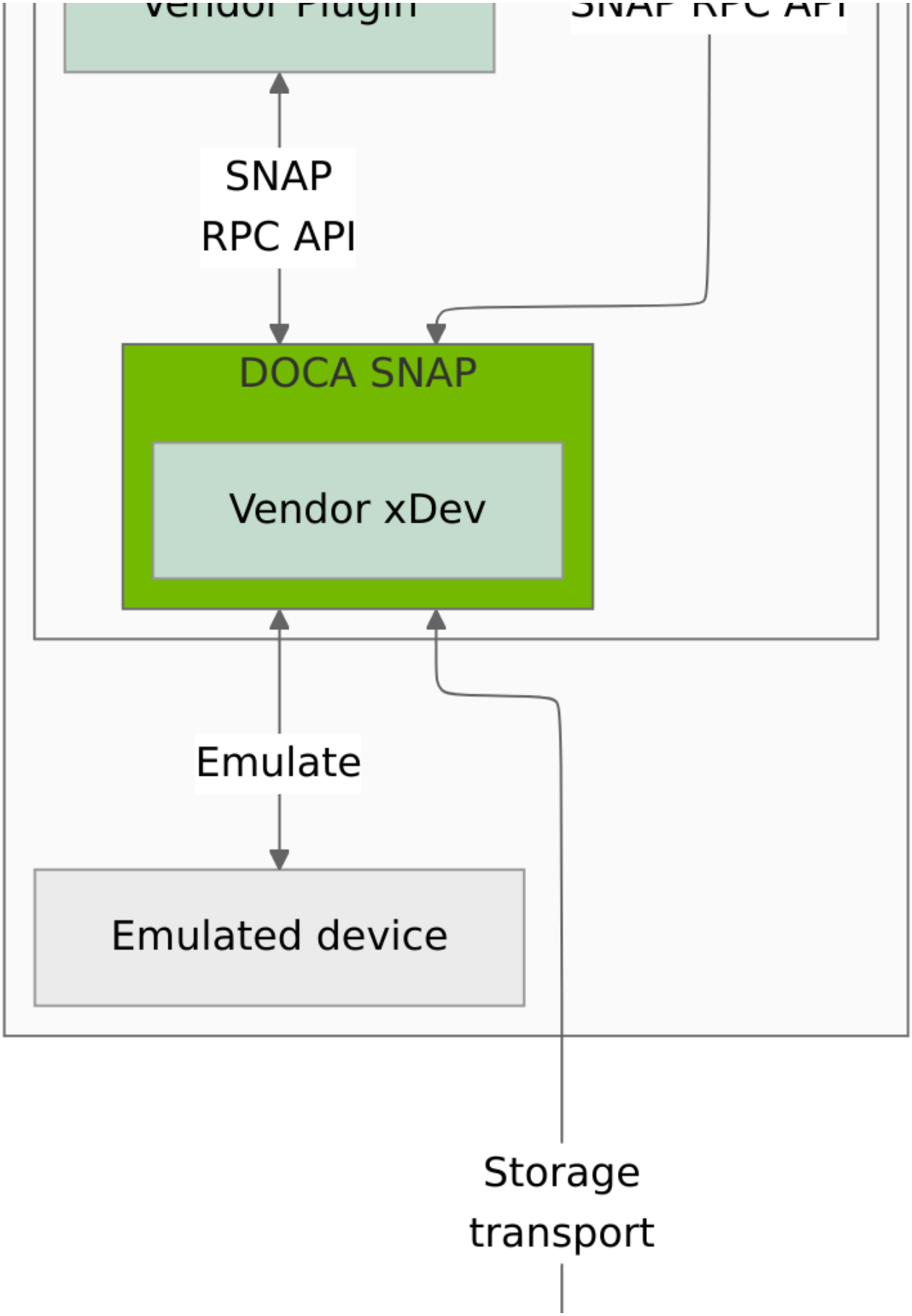
SNAP Node Driver

The SNAP Node Driver runs on each DPU that is used to attach storage. Responsibilities include:

- Invoking the Vendor Plugin via the StoragePlugin gRPC API to trigger the attachment of remote storage resources to the DOCA SNAP service.

- Interacting directly with the DOCA SNAP service via the **SNAP RPC API** to make storage resources available to the host.


DPF Control-plane node
DPU Cluster K8S API
Volume CR
VolumeAttachment CR
Worker node
DPU
SNAP Node driver
StoragePlugin
gRPC API
Vendor Plugin
SNAP RPC API


Vendor Plugin
SNAP RPC API
SNAP
RPC API
DOCA SNAP
Vendor xDev
Emulate
Emulated device
Storage
transport

DOCA SNAP

NVIDIA DOCA SNAP technology encompasses a family of services that enable hardware-accelerated virtualization of local storage running on NVIDIA BlueField products. The SNAP services present networked storage as local block or file system devices to the host, emulating local drives on the PCIe bus. Additional details about the SNAP services can be found in the DOCA SNAP documentation.

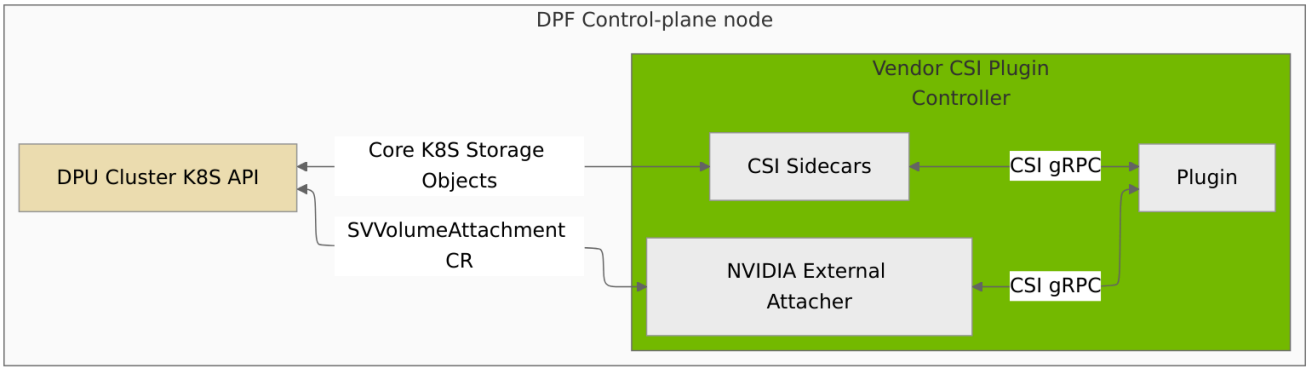
Vendor-specific components

Vendor CSI Plugin Controller

This is the controller part of the standard Kubernetes CSI driver of the storage vendor which is used for any Kubernetes deployment. It should be used to create/delete/manage the volumes on the underlying storage system.

If the storage vendor requires to support the ControllerPublishVolume/ ControllerUnpublishVolume operations, then a small adjustment in the controller deployment mechanism is required, due to the replacement of the Kubernetes native `VolumeAttachment [storage.k8s.io/v1]` object by the SVVolumeAttachment CRD object.

The original Kubernetes `external-attacher` sidecar should be replaced by a new external-attacher sidecar, provided by NVIDIA, which monitors the SVVolumeAttachment objects instead of the Kubernetes native `VolumeAttachment [storage.k8s.io/v1]` objects (see diagram below). The new external-attacher sidecar calls the `ControllerPublish` and `ControllerUnpublish` functions of the CSI controller.



An appropriate storage class that represents the CSI controller should be created on the DPU Kubernetes cluster (see example below). The reclaimPolicy field in the storage class MUST be set to Delete. The reason is that the reclaim policy is actually managed by the NVIDIA storage class.

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: example-vendor.example
  annotations:
    storageclass.kubernetes.io/is-default-class: "false"
provisioner: csi-driver.example-vendor.example
reclaimPolicy: Delete
```

In addition, a DPUStorageVendor CR object should be created in the host cluster, and the name of the StorageClass of the storage vendor (that exists in the DPU cluster) should be set in the storageClassName parameter (see example below).

```
apiVersion: storage.nvidia.com/v1alpha1
kind: DPUStorageVendor
metadata:
  name: example-vendor
  namespace: nvidia-storage
spec:
  storageClassName: "example-vendor.example"
  pluginName: "example-vendor-plugin"
```

If the storage vendor requires to support the controller attach/detach API then an appropriate `CSIDriver` object that represents the storage vendor CSI driver should be created on the DPU Kubernetes cluster (see example below), with the property attacheRequired set to True.

This object is used by the SNAP Host Controller to determine if ControllerPublishVolume/ ControllerUnpublishVolume operations are supported by the storage vendor.

```
apiVersion: storage.k8s.io/v1
kind: CSIDriver
metadata:
  name: csi-driver.example-vendor.example
spec:
  attachRequired: true // Indicates this CSI volume driver requires an attach operation
```

NVIDIA provides several sample Vendor CSI plugin controllers that can be used as references. Helm charts for these controllers are available in the DPF Storage Vendors Charts repository.

> **Note:** These plugins are not intended to be used in production environments.

Vendor Plugin

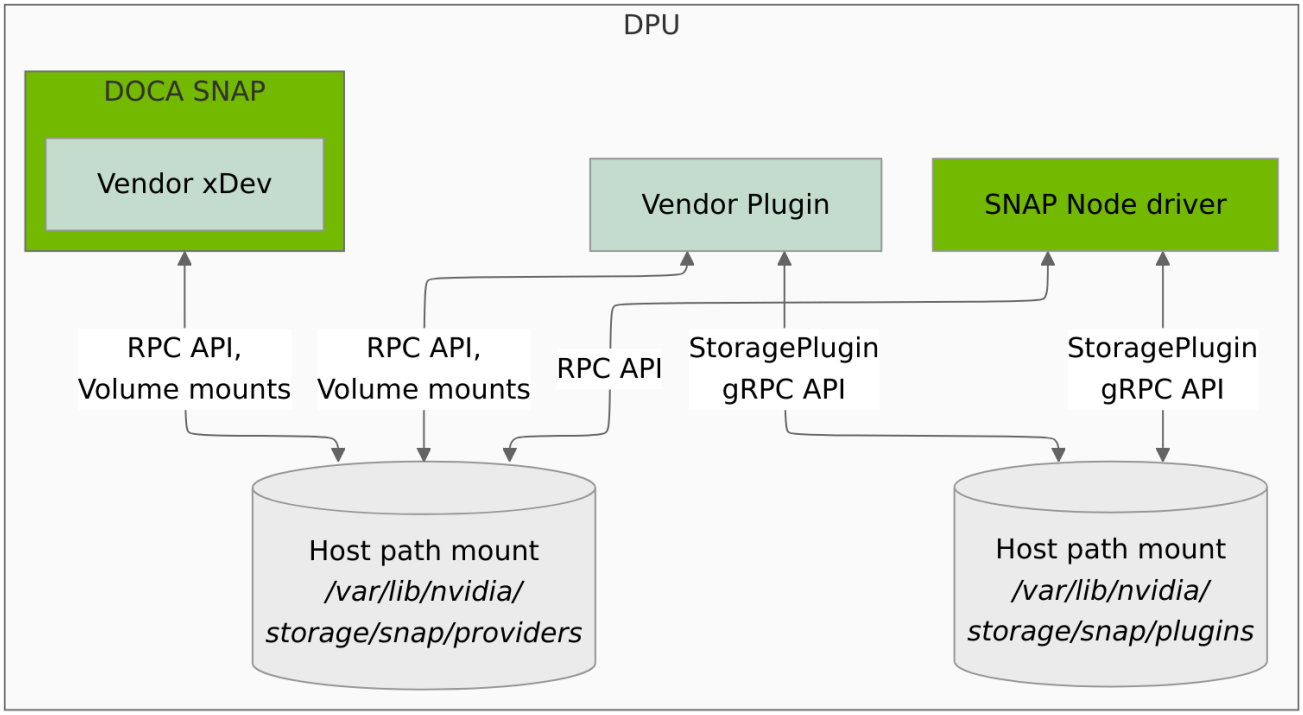
> **Note:** Vendors can use the plugins provided by NVIDIA or implement their own if needed. NVIDIA provides the following plugins: `nvidia-block` (uses NVMe-oF) and `nvidia-fs` (uses NFS-kernel client).

The concept of this component is very similar to the Kubernetes CSI node-driver. It is responsible to translate the StoragePlugin API into the storage vendor specific RPC calls.

The plugin requires direct access to the DPU. It should expose a gRPC interface through a UNIX domain socket which should be shared through a HostPath volume on the DPU worker node. This socket is used by the SNAP Node Driver to make calls to this component. It is responsible for creating the UNIX domain socket in the following shared folder: `/var/lib/nvidia/storage/snap/plugins/{plugin-name}/dpu.sock`, where `plugin-name` represent the name of the ***Vendor plugin***.

If a plugin needs to mount filesystems or block devices to the DOCA SNAP Pod before creating a `bdev` or `fsdev`, it should use the following path:
`/var/lib/nvidia/storage/snap/providers/{provider-name}/volumes/{plugin-name}/{volume-id}`
, where `provider-name` is the name of the DOCA SNAP provider that the plugin uses, `plugin-name` is the name of the ***Vendor plugin***, and `volume-id` is the ID of the volume.

It interacts with the Storage vendor xDev module, linked with the DOCA SNAP, through the SNAP UNIX domain socket which should be also shared through a HostPath volume on the DPU worker node. The connections between the node components are described in the diagram below.



Storage vendor xDev module

This is an optional SPDK device module which provides storage-specific logic inside the SNAP service. If exists - it should be managed by the storage vendor DPU plugin component. The module is deployed as a static or shared library along with the SNAP service.

Custom Resource Definitions (CRDs)

Public API

These are the APIs that users directly interact with to manage storage resources.

DPUStoragePolicy CRD

Defines a storage policy that maps between a policy and a list of DPU storage vendors.

```yaml
apiVersion: storage.dpu.nvidia.com/v1alpha1
kind: DPUStoragePolicy
metadata:
  name: example-dpu-storage-policy
spec:
  dpuStorageVendors:
    - example-vendor-1
    - example-vendor-2
  selectionAlgorithm: NumberVolumes  # Random or NumberVolumes
  parameters:
    parameter1: value1
    parameter2: value2
```

DPUStorageVendor CRD

Represents a DPU storage vendor. Each storage vendor must have exactly one `DPUStorageVendor` custom resource.

```yaml
apiVersion: storage.dpu.nvidia.com/v1alpha1
kind: DPUStorageVendor
metadata:
  name: example-vendor
spec:
  storageClassName: example-vendor-storage-class
  pluginName: example-vendor-plugin
```

DPUVolume CRD

Represents a persistent volume that will be provisioned by the DPF storage subsystem.

```yaml
apiVersion: storage.dpu.nvidia.com/v1alpha1
kind: DPUVolume
metadata:
  name: example-volume
spec:
  dpuStoragePolicyName: example-dpu-storage-policy
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
  volumeMode: Filesystem  # Filesystem or Block
  parameters:
    parameter1: value1
```

Note: The `volumeMode` field controls which emulation method will be used for the volume. For `Filesystem` mode, the volume will be exposed to the host as a Virtio-FS device. For `Block` mode, the volume will be exposed as an emulated NVMe device.

DPUVolumeAttachment CRD

Captures the intent to attach/detach a DPU volume to/from a specific DPU node.

```yaml
apiVersion: storage.dpu.nvidia.com/v1alpha1
kind: DPUVolumeAttachment
metadata:
  name: example-volume-attachment
spec:
  dpuVolumeName: example-volume
  dpuNodeName: dpu-node-01
  functionType: vf
  hotplugFunction: false
```

> **Note:** The `functionType` and `hotplugFunction` fields control the function type used for volume attachment. Currently supported combinations are `vf` with `false` (VF on top of a static PF) and `pf` with `true` (hot-plugged PF).

Internal API

These APIs are not intended to be directly used by end users. They are included here for reference only.

SVVolumeAttachment CRD

Captures the intent to attach/detach the specified volume to/from the specified node for the storage vendor.

```yaml
apiVersion: storage.dpu.nvidia.com/v1alpha1
kind: SVVolumeAttachment
metadata:
  name: sv-volume-attachment-example
spec:
  attacher: vendor-plugin
  nodeName: node01
  source:
    persistentVolumeName: pv-example
```

VolumeAttachment CRD

Captures the intent to attach/detach the specified NVIDIA volume to/from the specified node.

```yaml
apiVersion: storage.dpu.nvidia.com/v1alpha1
kind: VolumeAttachment
metadata:
  name: nv-volume-attachment-example
spec:
  nodeName: node01
  source:
    volumeRef:
      name: volume-example
      namespace: default
  parameters:
    parameter1: value1
  functionType: vf  # pf or vf
  hotplugFunction: false
status:
  storageAttached: true
  dpu:
    attached: true
    pciDeviceAddress: "0000:03:00.0"
    deviceName: nvme0n1
    bdevAttrs:
      nvmeNsID: 1
      nvmeUUID: "12345678-1234-1234-1234-123456789012"
```

Volume CRD

Represents a persistent volume on the DPU cluster, mapping between the tenant K8s PV object and the actual volume on the DPU cluster.

```yaml
apiVersion: storage.dpu.nvidia.com/v1alpha1
kind: Volume
metadata:
  name: volume-example
spec:
  request:
    capacityRange:
      request: "10Gi"
    accessModes:
      - ReadWriteOnce
    volumeMode: Filesystem
  storagePolicyRef:
    name: example-storage-policy
    namespace: default
  storageParameters:
    policy: example-storage-policy
    parameter1: value1
  volume:
    id: volume-12345
    capacity: "10Gi"
    accessModes:
      - ReadWriteOnce
    reclaimPolicy: Delete
    storageVendorName: example-vendor
    storageVendorPluginName: example-vendor-plugin
status:
  state: Available
```

StoragePlugin API

Each vendor plugin must implement the following gRPC API to interact with the DPF storage subsystem:

Services

```
syntax = "proto3";

package nvidia.storage.plugins.v1;

import "google/protobuf/wrappers.proto";

// The Identity service provides APIs to identify the plugin and verify its health.
service IdentityService {
  // GetPluginInfo returns the name and version of the plugin.
  rpc GetPluginInfo(GetPluginInfoRequest) returns
(GetPluginInfoResponse);

  // Probe checks the health and readiness of the plugin.
  rpc Probe(ProbeRequest) returns (ProbeResponse);
}

// The StoragePlugin service provides APIs to manage storage devices.
service StoragePluginService {
  // StoragePluginGetCapabilities returns the capabilities supported by the plugin.
  rpc
StoragePluginGetCapabilities(StoragePluginGetCapabilitiesRequest)
      returns (StoragePluginGetCapabilitiesResponse);

  // GetSNAPProvider retrieves the name of the SNAP provider used by the plugin.
  rpc GetSNAPProvider(GetSNAPProviderRequest) returns
(GetSNAPProviderResponse);

  // CreateDevice creates a new storage device and exposes it.
  rpc CreateDevice(CreateDeviceRequest) returns
(CreateDeviceResponse);

  // DeleteDevice removes a storage device.
  rpc DeleteDevice(DeleteDeviceRequest) returns
(DeleteDeviceResponse);
```

```
  // GetDevice gets a storage device.
  rpc GetDevice(GetDeviceRequest) returns (GetDeviceResponse);

  // ListDevices list all devices information
  rpc ListDevices(ListDevicesRequest) returns
(ListDevicesResponse);
}
```

All the RPC methods that are not implemented by the plugin MUST reply with the non-ok gRPC status code, `12 UNIMPLEMENTED`.

Any of the RPCs defined above may timeout and may be retried. The SNAP Node Driver may choose the maximum time it is willing to wait for a call, how long it waits between retries, and how many times it retries (these values are not negotiated between plugin and SNAP Node Driver).

Idempotency requirements ensure that a retried call with the same fields continues where it left off when retried. The only way to cancel a call is to issue a "negation" call if one exists. For example, issue a DeleteDevice call to cancel a pending CreateDevice operation.

GetPluginInfo

This method should return the name of the plugin (e.g., example-vendor-plugin.example) and the version of the plugin (e.g., 1.0). On success, the method should reply 0 OK in the gRPC status code.

If the plugin is unable to complete the `GetPluginInfo` call successfully, it MUST reply a non-ok gRPC code in the gRPC status.

Return codes:

Number	Code	Description
0	OK	
9	FAILED_PRECONDITION	Plugin is unable to complete the call successfully

```
// GetPluginInfoRequest is used to request the plugin's information.
message GetPluginInfoRequest {}

// GetPluginInfoResponse provides the plugin's name, version, and manifest details.
message GetPluginInfoResponse {
  // The name of the plugin. It must follow domain name notation format
  // (https://tools.ietf.org/html/rfc1035#section-2.3.1) and must be 63 characters
  // or less. This field is required.
  string name = 1;

  // The version of the plugin. This field is required.
  string vendor_version = 2;

  // The manifest provides additional opaque information about the plugin.
  // This field is required.
  map<string, string> manifest = 3;
}
```

Probe

This API is called to verify that the plugin is in a healthy and ready state. If an unhealthy state is reported, via a non-success response, the CO (in our case Kubernetes) may take action with the intent to bring the plugin to a healthy state (e.g., restarting the plugin container).

The plugin should verify its internal state and returns 0 OK if the validation succeeds. If the plugin is still initializing, but is otherwise perfectly healthy, it shall return 0 OK with the ready field set to False.

Return codes:

Number	Code	Description
0	OK	
9	FAILED_PRECONDITION	Plugin is unable to complete the call successfully

```
// ProbeRequest is used to check the plugin's health and readiness.
message ProbeRequest {}

// ProbeResponse indicates the plugin's health and readiness status.
message ProbeResponse {
  // Ready indicates whether the plugin is healthy and ready.
  google.protobuf.BoolValue ready = 1;
}
```

StoragePluginGetCapabilities

This API allows the SNAP Node-Driver to check the supported capabilities of the storage service provided by the plugin.

If the plugin is unable to complete the `StoragePluginGetCapabilities` call successfully, it MUST return a non-ok gRPC code in the gRPC status.

Return codes:

Number	Code	Description
0	OK	
9	FAILED_PRECONDITION	Plugin is unable to complete the call successfully

```
// StoragePluginGetCapabilitiesRequest is used to request the plugin's capabilities.
message StoragePluginGetCapabilitiesRequest {}

// StoragePluginGetCapabilitiesResponse provides the list of supported capabilities.
message StoragePluginGetCapabilitiesResponse {
  // Capabilities supported by the plugin.
  repeated StoragePluginServiceCapability capabilities = 1;
}

// StoragePluginServiceCapability describes a capability of the storage plugin service.
message StoragePluginServiceCapability {
  // RPC specifies an RPC capability type.
  message RPC {
    // Type defines the specific capability.
    enum Type {
      // UNSPECIFIED indicates an undefined capability.
      TYPE_UNSPECIFIED = 0;

      // CREATE_DELETE_BLOCK_DEVICE indicates support for block device creation and deletion.
      TYPE_CREATE_DELETE_BLOCK_DEVICE = 1;

      // CREATE_DELETE_FS_DEVICE indicates support for filesystem device creation and deletion.
      TYPE_CREATE_DELETE_FS_DEVICE = 2;

      // GET_DEVICE_STATS indicates support for retrieving device statistics.
      TYPE_GET_DEVICE_STATS = 3;

      // LIST_DEVICES indicates support for listing devices.
      TYPE_LIST_DEVICES = 4;
    }

    // The type of the capability.
    Type type = 1;
  }
```

```
  // The specific type of capability.
  oneof type {
    // Specifies the RPC capabilities of the service.
    RPC rpc = 1;
  }
}
```

GetSNAPProvider

The API is called by the SNAP Node Driver to retrieve the name of the SNAP provider used by the plugin. If the plugin uses the default SNAP process (provided by NVIDIA) then an empty string should be returned. Otherwise, a unique name represents the SNAP process used by this plugin should be returned. The name is used to identify the UNIX domain socket used by both the plugin and the SNAP Node Driver to communicate with the SNAP process.

If the plugin is unable to complete the `GetSNAPProvider` call successfully, it MUST return a non-ok gRPC code in the gRPC status.

Return codes:

Number	Code	Description
0	OK	
9	FAILED_PRECONDITION	Plugin is unable to complete the call successfully

```
// GetSNAPProviderRequest is used to retrieve the SNAP provider's name.
message GetSNAPProviderRequest {}

// GetSNAPProviderResponse provides the SNAP provider's name.
message GetSNAPProviderResponse {
  // The name of the SNAP provider. If this field is empty, the default provider
  // (e.g., NVIDIA) is used. This field is optional.
  string provider_name = 1;
}
```

CreateDevice

The API is called by the SNAP Node Driver prior to the volume being exposed by SNAP.

The operation must be idempotent. If the device corresponding to the volume_id is already created and is identical to the specified access_modes and volume_mode the plugin must reply 0 OK with the corresponding CreateDeviceResponse message.

Business logic required by the plugin: 1. Allocate a unique device name to be used by SNAP 2. Connect the device to the underlying storage system by using the storage vendor specific APIs (if needed). 3. Provide the device to the SNAP process by using the SNAP JSON-RPC APIs. For example: fsdev_aio_create aio0 /host-folder 4. Return the device name

If the plugin is unable to complete the `CreateDevice` call successfully, it MUST return a non-ok gRPC code in the gRPC status.

Return codes:

Number	Code	Description
0	OK	
9	FAILED_PRECONDITION	Plugin is unable to complete the call successfully

```
// CreateDeviceRequest is used to create a new storage device.
message CreateDeviceRequest {
  // The unique identifier for the volume.
  string volume_id = 1;

  // The access modes for the volume.
  repeated AccessMode access_modes = 2;

  // The volume mode, either Filesystem or Block. Default is Filesystem.
  string volume_mode = 3;

  // Static properties of the volume. This field is optional.
  map<string, string> publish_context = 4;

  // Static properties of the volume. This field is optional.
  map<string, string> volume_context = 5;

  // Static properties of the storage class. This field is optional.
  map<string, string> storage_parameters = 6;
}

// CreateDeviceResponse provides the details of the created device.
message CreateDeviceResponse {
  // The name of the created device (e.g., SPDK FSdev/Bdev name).
  string device_name = 1;
}

// AccessMode specifies how a volume can be accessed.
enum AccessMode {
  // ACCESS_MODE_UNSPECIFIED indicates an unspecified access mode.
  ACCESS_MODE_UNSPECIFIED = 0;

  // ACCESS_MODE_RWO indicates read/write on a single node.
  ACCESS_MODE_RWO = 1;
```

```
  // ACCESS_MODE_ROX indicates read-only on multiple nodes.
  ACCESS_MODE_ROX = 2;

  // ACCESS_MODE_RWX indicates read/write on multiple nodes.
  ACCESS_MODE_RWX = 3;

  // ACCESS_MODE_RWOP indicates read/write on a single pod.
  ACCESS_MODE_RWOP = 4;
}
```

DeleteDevice

The API is called by the SNAP Node Driver after the device has been deleted from SNAP.

The operation must be idempotent. If the device corresponding to the volume_id and the device_name does not exist, the plugin must reply 0 OK.

Business logic required by the plugin: 1. Remove the device from the SNAP process. 2. Disconnect the device from the underlying storage system by using the storage vendor specific APIs (if needed).

If the plugin is unable to complete the `DeleteDevice` call successfully, it MUST return a non-ok gRPC code in the gRPC status.

Return codes:

Number	Code	Description
0	OK	
9	FAILED_PRECONDITION	Plugin is unable to complete the call successfully

```
// DeleteDeviceRequest is used to delete a storage device.
message DeleteDeviceRequest {
  // The unique identifier for the volume.
  string volume_id = 1;

  // The name of the device to be deleted.
  string device_name = 2;
}

// DeleteDeviceResponse is the response for deleting a device.
message DeleteDeviceResponse {}
```

ListDevices

A Storage vendor plugin must implement this API if it has LIST_DEVICES capability. The plugin shall return the information about all the devices that it knows about. If devices are created and/or deleted while the SNAP Node Driver is concurrently paging through ListDevices results then it is possible that the SNAP Node Driver may either witness duplicate devices in the list, not witness existing devices, or both. The SNAP Node Driver shall not expect a consistent "view" of all devices when paging through the device list via multiple calls to ListDevices.

If the plugin is unable to complete the `ListDevices` call successfully, it MUST return a non-ok gRPC code in the gRPC status.

Return codes:

Number	Code	Description
0	OK	
10	ABORTED	Indicates that starting_token is not valid

```
message ListDevicesRequest {
  // number of entries that can be returned.
  int32 max_entries = 1;

  // A token to specify where to start paginating. Set this field to
  // `next_token` returned by a previous `ListDevices` call to get the
  // next page of entries.
  string starting_token = 2;
}

message ListDevicesResponse {

  message Entry {
    string volume_id = 1;
    string device_name = 2;
  }

  repeated Entry entries = 1;

  // This token allows the caller to get the next page of entries for
  // `ListDevices` request.
  string next_token = 2;
}
```

GetDevice

A Storage vendor plugin must implement this API if it has GET_DEVICE_STATS capability. The plugin shall return the information about the device corresponding to the volume_id and the device_name.

If the device corresponding to the volume_id and the device_name does not exist, the plugin must reply 5 NOT_FOUND.

If the plugin is unable to complete the GetDevice call successfully, it MUST return a non-ok gRPC code in the gRPC status.

Return codes:

Number	Code	Description
0	OK	
5	NOT_FOUND	The device was not found
9	FAILED_PRECONDITION	Plugin is unable to complete the call successfully

```
// GetDeviceRequest is used to get information about the device
message GetDeviceRequest {
  // volume identifier
  string volume_id = 1;

  // The device name. For example, SPDK FSdev/Bdev name
  string device_name = 2;
}

message GetDeviceResponse {
  // list of access modes for the volume.
  repeated AccessMode access_modes = 1;

  // Indicates the volume mode. Either Filesystem or Block. Default value is Filesystem. This field is OPTIONAL
  string volume_mode = 2;

  // Opaque static publish properties of the volume. This field is OPTIONAL
  map<string, string> publish_context = 3;

  // Opaque static properties of the volume. This field is OPTIONAL
  map<string, string> volume_context = 4;

  // Opaque static properties of the storage class. This field is OPTIONAL
  map<string, string> storage_parameters = 5;
}
```

Deployment

Example manifests for DPF storage subsystem components can be found DPF repo under `dpuservices/storage/examples`.

System

This section provides comprehensive guides for developing and building conformant DPF systems, including development environments, testing strategies, and system configuration.

System Documentation Files

- Developing a Conformant DPF System - Building DPF-compliant systems, including architecture patterns, implementation guidelines, and compliance requirements
- Conditions - Understanding feature flags, system conditions, and how to implement feature-gating mechanisms in DPF systems
- Local Development - Setting up development environments, tooling, and workflows for DPF system development
- End-To-End Testing - Comprehensive testing strategies and frameworks for validating complete DPF system functionality
- Scale Testing - Performance and scalability testing methodologies for DPF systems under various load conditions

Conditions

The conditions specified in the DPF Kubernetes objects represent the various states and requirements of the resource during its lifecycle. These conditions are critical for the reconciliation loop of the DPF controllers, providing insights into the current status, potential issues, and the overall health of the system.

Each condition consists of a Reason and a Message that can describe the current state of the reconciliation.

- reason contains a programmatic identifier indicating the reason for the condition's last transition.

- message is a human-readable message indicating details about the transition.

Note: Not all DPF objects follow this condition standard. Future work should align condition design across all Kubernetes objects.

Condition Type Categories

DPF has three categories of condition:

XXXReconciled: if the condition has been reconciled successfully. Reconciliation means that changes to the object have been applied. Reasons can be: Success, Error, Pending or AwaitingDeletion.

XXXReady: Check the objects status and reflect the ready state Reasons can be: Success, Failure, Pending or AwaitingDeletion.

Ready (singleton): the overall status of the controller – only True if ALL conditions are True. Reasons can be: Success, Failure, Pending or AwaitingDeletion.

Condition Reasons

- **AwaitingDeletion**: A resource is marked for deletion
- **Pending**: If the controller has not completed its initialization
- **Error**: Something the system CAN recover from
- **Failure**: Terminal state the system CANNOT recover from
- **Success**: the overall success reason

DPUService Example

The DPUService has the following condition types:

- **Ready**: The overall state of the DPU Service Controller reconciliation
- **ApplicationPrereqsReconciled**: Represents the status of the prereqs for an Application, which consists of the ArgoSecrets, AppProjects and ImagePullSecrets
- **ApplicationsReconciled**: Represents the status of the application deployments for the host and DPU clusters. This means only that the resources are reconciled

- **ApplicationsReady**: Represents the status of the application ready state for the host and DPU clusters. If a failure occurs a message is added to the condition with the hint taking a look at the ArgoCD resources

Go Implementation

These types are implemented in go in the DPUService API package.

```
// Condition types
const (
	// ConditionDPUServiceInterfaceReconciled is the condition type that indicates that the
	// DPUServiceInterface is reconciled.
	ConditionDPUServiceInterfaceReconciled conditions.ConditionType = "DPUServiceInterfaceReconciled"
	// ConditionApplicationPrereqsReconciled is the condition type that indicates that the
	// application prerequisites are reconciled.
	ConditionApplicationPrereqsReconciled conditions.ConditionType = "ApplicationPrereqsReconciled"
	// ConditionApplicationsReconciled is the condition type that indicates that the
	// applications are reconciled.
	ConditionApplicationsReconciled conditions.ConditionType = "ApplicationsReconciled"
	// ConditionConfigPortsReconciled is the condition type that indicates that the
	// applications config ports are reconciled.
	ConditionConfigPortsReconciled conditions.ConditionType = "ConfigPortsReconciled"
	// ConditionApplicationsReady is the condition type that indicates that the
	// applications are ready.
	ConditionApplicationsReady conditions.ConditionType = "ApplicationsReady"
)

var (
	// Conditions is the list of conditions that the DPUService can have.
	Conditions = []conditions.ConditionType{
		conditions.TypeReady,
		ConditionApplicationPrereqsReconciled,
		ConditionApplicationsReconciled,
		ConditionApplicationsReady,
		ConditionDPUServiceInterfaceReconciled,
		ConditionConfigPortsReconciled,
	}
```

```
)
```

Kubernetes Object Conditions

A ApplicationsReady condition looks like:

```
status:
conditions:
- lastTransitionTime: "2024-07-31T10:23:36Z"
  message: "Applications couldn't be deployed: cluster1/multus.applications.argoproj.io, cluster2/flannel.applications.argoproj.io"
  reason: Pending
  status: "False"
  type: ApplicationsReady
```

Where a ready DPUApplicationsReady condition looks like:

```
status:
conditions:
- lastTransitionTime: "2024-07-31T10:23:36Z"
  message: "Reconciliation successful"
  reason: Succesful
  status: "True"
  type: ApplicationsReady
```

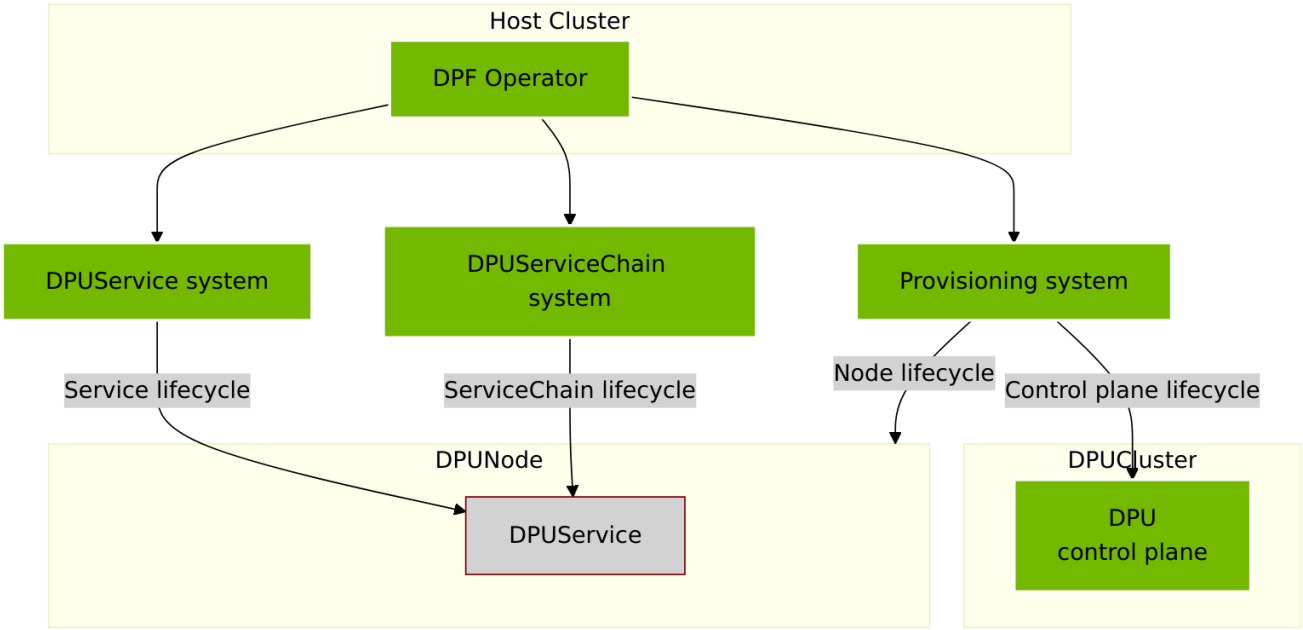
The full set of conditions on the DPUService will look like:

```
status:
conditions:
- lastTransitionTime: "2024-07-31T10:23:36Z"
  message: "Resources are pending in deletion"
  reason: Pending
  status: "False"
  type: Ready
- lastTransitionTime: "2024-07-31T10:23:36Z"
  message: "Applications %s are pending in deletion"
  reason: AwaitingDeletion
  status: "False"
  type: ApplicationsReady
- lastTransitionTime: "2024-07-31T10:23:36Z"
  message: "Application Prereqs are pending in deletion"
  reason: AwaitingDeletion
  status: "False"
  type: ApplicationPrereqsReady
- lastTransitionTime: "2024-07-31T10:23:36Z"
  message: "Application Prereqs are pending in deletion"
  reason: Error
  status: "False"
  type: ApplicationPrereqsReady
- lastTransitionTime: "2024-07-31T10:23:36Z"
  message: ""
  reason: Pending
  status: "Unknown"
  type: ApplicationReady
```

Developing a Conformant DPF System

Independent software vendors (ISVs) can integrate DOCA Platform Foundation (DPF). Integration may involve modifying or replacing dependencies and components used in the upstream version of DPF.

The below details the system highlighting which components can be changed in integration.



DPF Operator

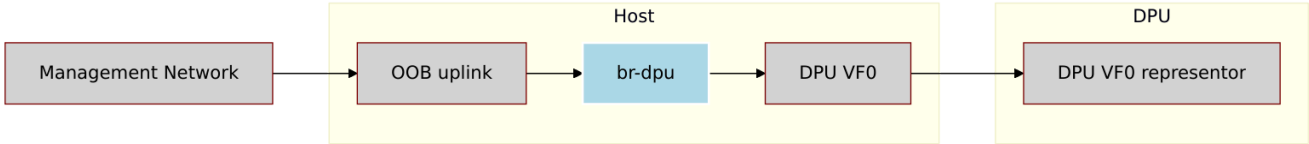
The DPF Operator has a number of dependencies which can be swapped out during integration in favor of solutions managed by the integrator. Replacing some of these components may require additional modification - i.e. certificate management with a tool other than cert manager may require changes to the deployments.

Today the following are either mentioned in the installation guide or installed as dependencies from the DPF Operator helm chart.

- certificate management using `cert-manager`
- storage provisioning using `NFS` and `local-path` provisioner
- NVIDIA Network Operator
- SR-IOV Network Operator
- NVIDIA Node maintenance operator

Host Networking Configuration

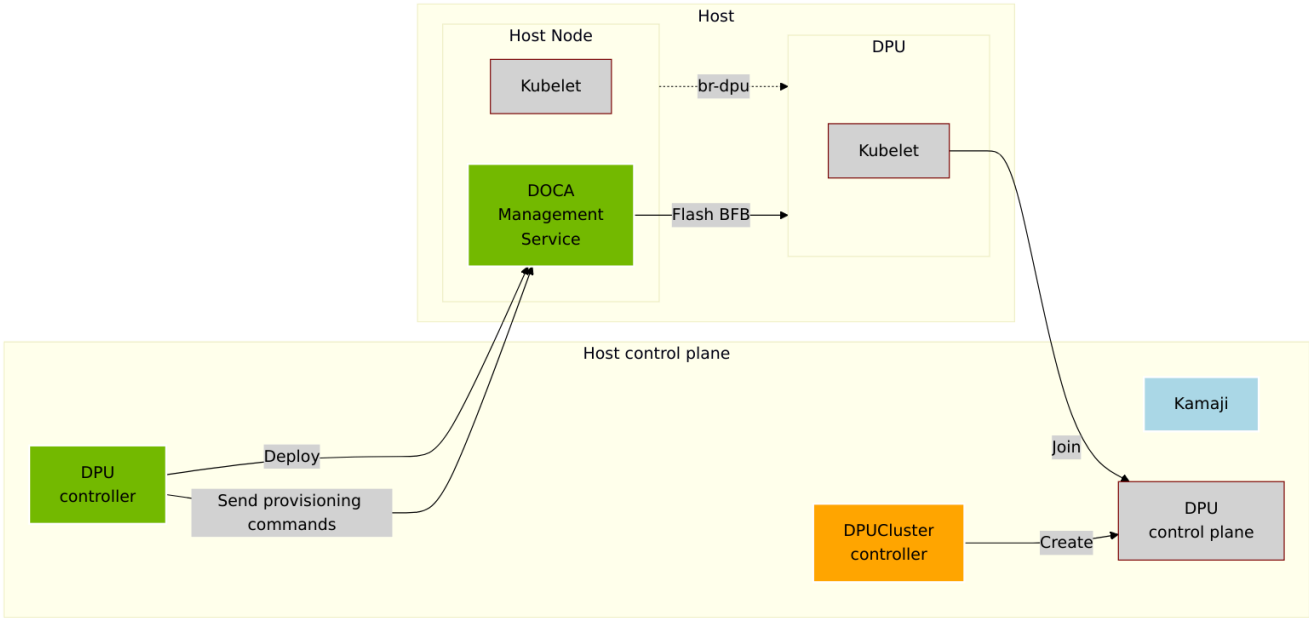
DPF relies on each worker node having a specific networking configuration. In this configuration the worker node management interface is plugged into a bridge called `br-dpu`. Virtual functions from the BlueField DPU are added to the same bridge.



This networking pattern must be implemented when integrating DPF. There is no component in DPF which configures this. For more information on the implementation details see the host networking prerequisites.

Provisioning System

Manage the lifecycle of the DPU hardware and its Kubernetes orchestration system.



The following components are used in upstream DPF and can be modified or replaced during integration:

- DPU Cluster manager- highlighted in orange - can be replaced in full in integration.
- Kamaji controller - a dependency of the DPUCluster manager - can be replaced.
- BF.cfg - a configuration file used when configuring the DPU.

For more information on implementing a DPUCluster manager see the DPUCluster documentation.

Custom bf.cfg

By default DPF will use a bf.cfg - the Bluefield configuration file - that is tied to the default BFB. If the BFB is replaced during integration it may also be required to replace the bf.cfg.

The DPU controller in the provisioning controller manager uses a template to render the bfb.cfg with values supplied by the user-supplied DPUFlavor. The default bf.cfg.template for v25.1 is here. The template takes the form of a go template. The values used to render it are supplied to it are described in this package.

To supply a custom bf.cfg template:

1. Create a `ConfigMap` with the following format in the target cluster:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: custom-bfb.cfg
  namespace: dpf-operator-system # Must be in the same namespace
as the DPFOperator.
data:
    BF_CFG_TEMPLATE: | # This key must be in the ConfigMap.
        {{.KubeadmJoinCMD}}
```

2. Configure the provisioning controller to use the custom bf.cfg.template through the DPFOperatorConfig.

```
apiVersion: operator.dpu.nvidia.com/v1alpha1
kind: DPFOperatorConfig
metadata:
  name: dpfoperatorconfig
  namespace: dpf-operator-system
spec:
  provisioningController:
    bfbPVCName: "bfb-pvc"
    bfCFGTemplateConfigMap: custom-bfb.cfg
...
```

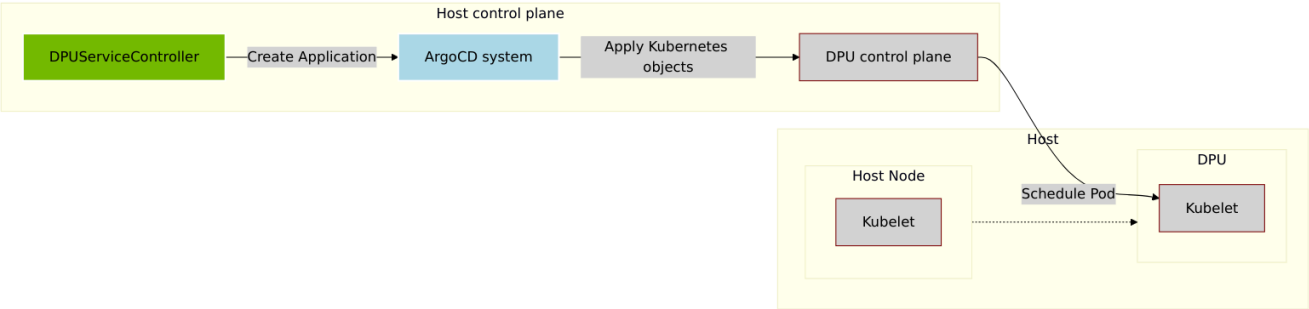
This example template contains only one variable and misses many important configuration steps. It is not sufficient to provision a DPU in DPF. To understand how to write a custom bf.cfg.template the default template serves as the best example.

DPUService System

Orchestrate DOCA Services to run on DPUs.

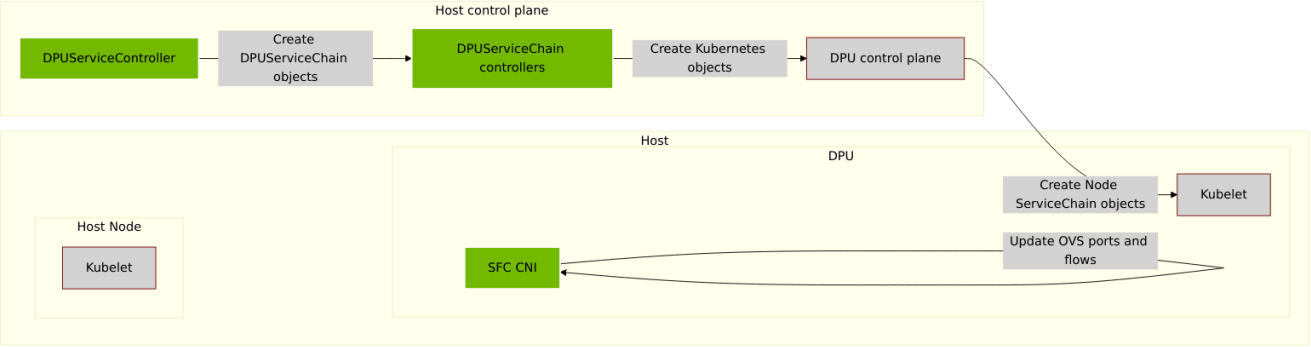
The following components are used in upstream DPF and can be modified or replaced during integration:

- ArgoCD - a dependency of the DPUService controller manager can be replaced with a managed version of ArgoCD



DPUServiceChain System

Orchestrate Service Chains for advanced network flows through DPUs.



None of the components used in Service chain orchestration should be replaced during integration.

End-To-End Testing

DPF end-to-end tests are written in go and use the `ginkgo` and `gomega` frameworks.

The tests use a configuration file which is unique per test run.

Prerequisites

All tests require:

- a working Kubernetes cluster
- DPF Operator controller manager deployed and running

Individual tests may also have their own prerequisites which should be defined in their CI definition. For example different tests may require different sets of container images and helm charts to run.

Structure

The tests have the following structure:

- `Suite`: Reads configuration and sets up the system for a test run. There is one test configuration per `Suite`

- `Describe`: The core of a test - contains the full set of tests to be run. There is one DPFOperatorConfig per `Describe`. `Describe` nodes never run in parallel with each other
- `Context`: A set of tests. These are categorised and configured as a group representing a subsystem or a feature
- `It`: An individual test. Each `It` block is independent and leaves a clean state once finished. `It` blocks are able to run in parallel and continue on failure
- `By` is used to describe a step either inside or outside a test

Local Development

This guide helps the developer in setting and running a local env that can be used to deploy the various DPF components and run tests. The local env doesn't require a DPU.

Prerequisites

1. GO >= 1.24
2. Docker Engine
3. kubectl

Dev Flow

1. Go to your local dev project git repo

```
cd <path_to_local_repo/doca-platform-foundation
```

2. export env vars (Linux):

```
export REGISTRY=<insert_your_registry>
export IMAGE_PULL_KEY=<IMAGE_PULL_TOKEN>
export TAG=v0.1.0-$(git rev-parse --short HEAD)-$USER-test
```

On Mac the development environment is designed to work with Docker Desktop running with Apple Virtualization.

3. Generate all

```
make generate
```

4. Build images required for the quick DPF e2e test.

```
make test-release-e2e-quick
```

5. Deploy the test environment

```
make clean-test-env test-env-e2e
```

6. Deploy DPF Operator

```
make test-deploy-operator-helm
```

7. Run e2e tests

```
make test-e2e
```

Note: If you want to keep the created resources (e.g., `DPFOperatorConfig`, `DPU` and such) you can use the environment variable `E2E_SKIP_CLEANUP=true`

```
E2E_SKIP_CLEANUP=true make test-e2e
```

8. Clean test env

```
make clean-test-env
```

Scale Testing

This document describes a test design for assessing the DPF core components at scale. It mocks a number of parts of the DPF system to enable performance testing of the core DPF components in response to growth in specific dimensions of scale.

Testing components

The major differences between a full DPF installation and the scale testing infrastructure are:

1. No DPU hardware: The scale test does not use Bluefield DPUs. Interactions with the DPU are implemented on the API level.

2. No DPU Kubernetes nodes: The scale test does not provision Kubernetes nodes

3. No DMS: DPF uses DMS to manage the lifecycle of the DPU. Scale testing relies on a `mock-dms` component which implements the API expected by the DPU controller.

4. No hostnetwork configuration: DPF uses a hostnetwork pod to configure networking on the host.

The scale test requires a new component - `mock-dms`. `mock-dms` is a Kubernetes controller that:

- Watches DPU objects

- Creates a mock DMS listener on a new port for each DPU
- Adds an annotation to DPUs overriding the DMS address, DMS pod, and hostnetwork pod
- Answers gRPC calls from the DPU controller
- Creates a Kubernetes node object representing the DPU node

Testing Dimensions

The initial scale targets for the test are shown in the table below. Testing will be an iterative process and these targets will be updated on in response to test results.

Object	Scale target
DPUs	1000
DPUServices	10
DPUServiceChains	30
DPUServiceIPAMs	30
DPUServiceInterfaces	30
DPUSets	10
DPUDeployments	10
BFBs	10
DPUServiceCredentialRequests	10
DPUClusters	1
DPFOperatorConfigs	1

Testing Targets

The scale tests rely on DPF metrics to assess the performance of the components.

The following categories of metrics are of interest. The testing process is iterative and these targets will be further specified and updated in response to test results.

1. time to provision target number of DPU nodes
2. time to provision target number of DPUServices

3. time to provision target number of DPUServiceInterfaces
4. time to provision target number of DPUServiceChains
5. time to provision target number of DPUServiceIPAMs
6. number of errors in DPF controllers
7. number of errors in DPU cluster control plane
8. number of errors in target cluster control plane
9. reconcile time for DPF controllers
10. CPU / memory usage DPU cluster control plane
11. CPU / memory usage target cluster control plane
12. CPU / memory usage DPF controllers

Gaps

This scale testing approach does not adequately test the following at scale:

- DPUCluster components and management network - i.e. `sfc-controller`, `ovs-cni` `nvipam` `flannel` etc.
- DPUCluster control plane scale including etcd performance
- DPF controllers at large target cluster scale
- Resources on individual DPUs at scale e.g. DPU file descriptors, memory
- Specific DPUServices - i.e. OVN-Kubernetes, HBN at scale
- DMS operations at scale

Running the Scale Tests

You can set up a scale testing environment locally with the DPF developer environment. This builds and pushes the required images, spins up a new cluster, deploys dpf and mock-dms.

```
export REGISTRY=$YOUR_REGISTRY
export TAG=$YOUR_TAG
export IMAGE_PULL_KEY=$YOUR_IMAGE_KEY
export NODE_MEMORY=16g #adjust as per your system limits
export E2E_TEST_ARGS="-v -ginkgo.v -e2e.config=config-scale.yaml -ginkgo.label-filter=SCALE"
export E2E_SKIP_CLEANUP=true


make clean-test-env generate test-release-e2e-quick test-env-e2e
test-deploy-operator-helm test-deploy-mock-dms test-e2e
```

Verify

```
k get pods -n dpf-operator-system | grep mock-dms
mock-dms-controller-manager-9b7db9b4d-rs4rb                    1/1
Running   0          39m

k get nodes -A | grep dpu-worker | wc -l
10
```

Future Work

Improving Test Signal

- choose specific metrics and target values for a given infrastructure
- iterate on scale dimensions

Extending Scale Test Coverage

- Adding compute to the DPUCluster to test DPUCluster components and DPUCluster control plane
- Adding compute to the target cluster to test scaling of DPF components in large target clusters

Getting Started

Getting Started with DOCA Platform Framework

Welcome to the DOCA Platform Framework (DPF)! This guide will help you get up and running with DPF quickly and efficiently.

Introduction

The DOCA Platform Framework (DPF) is a comprehensive platform for managing Data Processing Units (DPUs) and deploying distributed services across DPU-enabled infrastructure. DPF enables unified DPU management, service orchestration, network virtualization, and storage services with built-in security features.

Choose Your Deployment Path

DPF supports different deployment modes depending on your infrastructure and security requirements:

Zero Trust Mode

Recommended for environments requiring maximum security with BMC access and secure boot capabilities.

Host Trusted Mode

Suitable for controlled environments with host trusted infrastructure.

Next Steps

Once you've completed the initial setup using one of the deployment guides above, explore these areas:

Learn More

- **Architecture Overview** - Understand DPF's system design
- **API Reference** - Explore the complete API documentation

Configure Services

- **DPU Services** - Available services and configuration options

User Guides and Support

- **User Guides** - Step-by-step guides for using DPF for specific use cases
- **Troubleshooting** - Common issues and solutions

Getting Help

- Check the Troubleshooting section for common issues
- Review Release Notes for known issues and updates
- Visit the GitHub repository for community support

DPF Host Trusted

Get Started with DPF Host Trusted

This guide shows you how to deploy the NVIDIA DPF Operator prepared for the Host Trusted mode, which is designed for bare-metal infrastructure with NVIDIA BlueField-3

DPUs.

Before you Begin

To follow this guide, you need the following:

- **A Kubernetes cluster** with administrative access for DPF Operator deployment
- **Bare-metal infrastructure** with NVIDIA DPUs
- **Network access** to the NVIDIA NGC Catalog for downloading DPF Operator charts and container images

For detailed requirements, ensure your system meets these prerequisites:

- **System Prerequisites**: See the DPF System Prerequisites for complete hardware and system requirements
- **Helm Dependencies**: See the Helm Prerequisites Guide for required Helm charts that must be installed before the DPF Operator
- **Network Configuration**: See the Host Network Configuration Prerequisites for detailed network setup requirements

Objectives

- Deploy the DPF Operator to your Kubernetes cluster
- Understand Host Trusted mode architecture and security model
- Verify successful installation and readiness for DPU provisioning
- Set up foundation for DPU acceleration in vanilla Kubernetes environments

Understanding Host Trusted Mode

In Host Trusted mode:

- **The host is considered a trusted entity** within the data center network

- **DPUs are managed through in-band communication** over the host (out-of-band DPU interface is not used)
- **The host participates as a worker node** in the Kubernetes cluster
- **DPUs are provisioned and can run accelerated network services** (HBN, OVN-K, etc.)

Deploy the DPF Operator

1. Add the DPF Helm Repository

First, add the NVIDIA DPF Helm repository to access the DPF Operator charts:

```
helm repo add --force-update dpf-repository
https://helm.ngc.nvidia.com/nvidia/doca
helm repo update
```

2. Install the DPF Operator

> **Note**
>
> Ensure you have completed the Helm Prerequisites Guide before proceeding with the DPF Operator installation.

Deploy the DPF Operator to your Kubernetes cluster using Helm:

```
helm upgrade --install -n dpf-operator-system dpf-operator dpf-
repository/dpf-operator --version=v25.10.0
```

The command above does the following:

- Creates the `dpf-operator-system` namespace if it doesn't exist
- Installs the DPF Operator version v25.10.0 from the NVIDIA repository

3. Verify the Installation

Check that the DPF Operator has been successfully deployed and is running:

```
kubectl rollout status deployment --namespace dpf-operator-system
dpf-operator-controller-manager
```

The output should be similar to:

```
deployment "dpf-operator-controller-manager" successfully rolled out
```

You can also verify all components are running:

```
kubectl get pods -n dpf-operator-system
```

What's Next?

This minimal setup provides a foundation for DPF Host Trusted mode. To proceed with DPU provisioning and making the DPUs act as passthrough devices, explore the guide **DPF Minimal (passthrough)**. For further Host Trusted mode use cases, refer to the DPF Host Trusted Use Cases documentation.

Cleanup (Optional)

If you need to remove the DPF Operator from your cluster:

1. Uninstall the DPF Operator:

```
helm uninstall dpf-operator -n dpf-operator-system
```

2. Remove the namespace:

```
kubectl delete ns dpf-operator-system
```

DPF Zero Trust

Get Started with DPF Zero Trust

This guide shows you how to deploy the NVIDIA DPF Operator prepared for the Zero Trust mode, which is designed for bare-metal infrastructure with NVIDIA BlueField-3 DPUs.

Before you Begin

To follow this guide, you need the following:

- **A Kubernetes cluster** with administrative access for DPF Operator deployment
- **Bare-metal infrastructure** with NVIDIA DPUs and BMC access
- **Network access** to the NVIDIA NGC Catalog for downloading DPF Operator charts and container images

Detailed Requirements

For detailed requirements, ensure your system meets these prerequisites:

- **System Prerequisites**: See the DPF System Prerequisites for complete hardware and system requirements.

- **Helm Dependencies**: See the Helm Prerequisites Guide for required Helm charts that must be installed before the DPF Operator.

Objectives

- Deploy the DPF Operator to your Kubernetes cluster
- Understand Zero Trust mode architecture and security model
- Verify successful installation and readiness for DPU provisioning
- Set up foundation for secure bare-metal DPU management

Understanding Zero Trust Mode

In Zero Trust mode:

- **The host is considered an untrusted entity** towards the data center network
- **DPUs are managed through their Baseboard Management Controller (BMC)** via Redfish protocol
- **All management traffic occurs over the DPU's out-of-band (OOB) network** for secure isolation
- **The DPU acts as a security barrier** between the host and the network infrastructure
- **The host sees the DPU as a standard NIC** with no access to the internal DPU management plane
- **DPUs are provisioned and can run accelerated network services** with hardware-level isolation

Deploy the DPF Operator

1. Add the DPF Helm Repository

First, add the NVIDIA DPF Helm repository to access the DPF Operator charts:

```
helm repo add --force-update dpf-repository
https://helm.ngc.nvidia.com/nvidia/doca
helm repo update
```

2. Install the DPF Operator

> **Note**
>
> Ensure you have completed the Helm Prerequisites Guide before proceeding with the DPF Operator installation.

Deploy the DPF Operator to your Kubernetes cluster using Helm:

```
helm upgrade --install -n dpf-operator-system dpf-operator dpf-
repository/dpf-operator --version=v25.10.0
```

The command above does the following:

- Creates the `dpf-operator-system` namespace if it doesn't exist
- Installs the DPF Operator version v25.10.0 from the NVIDIA repository
- Configures the operator to manage DPU resources across your cluster

3. Verify the Installation

Check that the DPF Operator has been successfully deployed and is running:

```
kubectl rollout status deployment --namespace dpf-operator-system
dpf-operator-controller-manager
```

The output should be similar to:

```
deployment "dpf-operator-controller-manager" successfully rolled out
```

You can also verify all components are running:

```
kubectl get pods -n dpf-operator-system
```

What's Next?

This minimal setup provides a foundation for DPF Zero Trust mode. To proceed with DPU provisioning and making the DPUs act as passthrough devices, explore the **DPU Passthrough in DPF Zero Trust** guide. For further Zero Trust mode use cases, refer to the DPF Zero Trust Use Cases documentation.

Cleanup (Optional)

If you need to remove the DPF Operator from your cluster:

1. Uninstall the DPF Operator:

```
helm uninstall dpf-operator -n dpf-operator-system
```

2. Remove the namespace:

```
kubectl delete ns dpf-operator-system
```

Helm Prerequisites

Overview

The DPF Operator requires several prerequisite components to function properly in a Kubernetes environment. This document provides comprehensive guidance on the Helm chart dependencies and their configuration values needed for a successful DPF Operator deployment.

Important Note

Starting with DPF v25.7, all Helm dependencies have been removed from the DPF chart. This means that **all dependencies must be installed manually** before installing the DPF chart itself.

Prerequisites Overview

The following table lists all required and optional Helm chart dependencies with their specific versions and purposes:

Helm Chart	Version	Description	Required	Post/Pre-installation
cert-manager	v1.18.1	Certificate management for Kubernetes, provides automatic TLS certificate issuance and renewal	▯	Pre-installation
argo-cd	7.8.2	GitOps continuous delivery tool for Kubernetes, necessary for DPUService integration	▯	Pre-installation
node-feature-discovery	0.17.1	Discovers and advertises hardware features and capabilities of DPUs in the cluster	▯	Pre-installation

Helm Chart	Version	Description	Required	Post/Pre-installation
maintenance-operator	0.2.0	Manages node maintenance operations and ensures graceful handling of node updates	□	Pre-installation
kamaji	1.2.0	Kubernetes cluster management platform for creating and managing the DPU Kubernetes clusters	□	Pre-installation
local-path-provisioner	0.0.31	Provides a local storage provisioner for Kubernetes, used for Kamaji etcd storage	□	Pre-installation
kube-state-metrics	5.25.1	Exposes DPF Operator related objects as metrics	□	Post-installation
grafana	8.4.6	Open-source analytics and monitoring platform for visualizing metrics and logs	□	Post-installation
prometheus	25.26.0	Time-series database and monitoring system for collecting and querying metrics	□	Doesn't matter

Some of the components requires the DPF Operator to be installed before they can be installed.

This is necessary for `kube-state-metrics` and `grafana`, because we rely on ConfigMaps created by the DPF Operator to provide the necessary configuration for these components.

Installation Options

Option 1: Using Helmfile

We provide a working helmfile configuration that can be used to install all dependencies with the correct values.

The helmfiles are located at `deploy/helmfiles/` in the DPF repository.

This approach ensures consistent deployment across different environments and simplifies the installation process.

Option 2: Manual Installation

If you prefer to install dependencies manually, you can use the individual Helm chart values provided in the sections below.

Required Configuration Values

The following sections provide the specific Helm chart values that must be configured before installing each dependency. These configurations ensure proper integration with the DPF Operator and optimal performance in your environment.

cert-manager

```yaml
startupapicheck:
  enabled: false
crds:
  enabled: true
affinity:
  nodeAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
      nodeSelectorTerms:
        - matchExpressions:
            - key: node-role.kubernetes.io/master
              operator: Exists
        - matchExpressions:
            - key: node-role.kubernetes.io/control-plane
              operator: Exists
tolerations:
  - operator: Exists
    effect: NoSchedule
    key: node-role.kubernetes.io/control-plane
  - operator: Exists
    effect: NoSchedule
    key: node-role.kubernetes.io/master
cainjector:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
          - matchExpressions:
              - key: node-role.kubernetes.io/master
                operator: Exists
          - matchExpressions:
              - key: node-role.kubernetes.io/control-plane
                operator: Exists
  tolerations:
    - operator: Exists
      effect: NoSchedule
```

```
      key: node-role.kubernetes.io/control-plane
    - operator: Exists
      effect: NoSchedule
      key: node-role.kubernetes.io/master
webhook:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
          - matchExpressions:
              - key: node-role.kubernetes.io/master
                operator: Exists
          - matchExpressions:
              - key: node-role.kubernetes.io/control-plane
                operator: Exists
  tolerations:
    - operator: Exists
      effect: NoSchedule
      key: node-role.kubernetes.io/control-plane
    - operator: Exists
      effect: NoSchedule
      key: node-role.kubernetes.io/master
```

argo-cd

```
## Disable the ApplicationSet controller.
applicationSet:
  replicas: 0
dex:
  enabled: false
notifications:
  enabled: false
global:
  podLabels:
    ovn.dpu.nvidia.com/skip-injection: ""
  affinity:
    nodeAffinity:
      # -- Default node affinity rules. Either: `none`, `soft` or
`hard`
      type: hard
      # -- Default match expressions for node affinity
      matchExpressions:
        - key: "node-role.kubernetes.io/control-plane"
          operator: Exists
  tolerations:
    - key: node-role.kubernetes.io/master
      operator: Exists
      effect: NoSchedule
    - key: node-role.kubernetes.io/control-plane
      operator: Exists
      effect: NoSchedule
redis:
  image:
    repository: mirror.gcr.io/redis
```

node-feature-discovery

```yaml
# Node Feature Discovery configuration
master:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
          - matchExpressions:
              - key: "node-role.kubernetes.io/master"
                operator: Exists
          - matchExpressions:
              - key: "node-role.kubernetes.io/control-plane"
                operator: Exists
worker:
  enable: true
  hostNetwork: true
  tolerations:
    - key: node.kubernetes.io/not-ready
      operator: Exists
  config:
    sources:
      pci:
        deviceClassWhitelist:
          - "0200"
        deviceLabelFields:
          - "class"
          - "vendor"
          - "device"
gc:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
          - matchExpressions:
              - key: "node-role.kubernetes.io/master"
                operator: Exists
```

```
          - matchExpressions:
              - key: "node-role.kubernetes.io/control-plane"
                operator: Exists
  tolerations:
    - key: node-role.kubernetes.io/master
      operator: Exists
      effect: NoSchedule
    - key: node-role.kubernetes.io/control-plane
      operator: Exists
      effect: NoSchedule
```

maintenance-operator

```yaml
# Maintenance Operator Chart configuration
operatorConfig:
  maxParallelOperations: 60%
operator:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
          - matchExpressions:
              - key: "node-role.kubernetes.io/master"
                operator: Exists
          - matchExpressions:
              - key: "node-role.kubernetes.io/control-plane"
                operator: Exists
  tolerations:
    - key: node-role.kubernetes.io/master
      operator: Exists
      effect: NoSchedule
    - key: node-role.kubernetes.io/control-plane
      operator: Exists
      effect: NoSchedule
```

kamaji

```yaml
# Kamaji configuration
resources: null
affinity:
  nodeAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
      nodeSelectorTerms:
        - matchExpressions:
            - key: "node-role.kubernetes.io/master"
              operator: Exists
        - matchExpressions:
            - key: "node-role.kubernetes.io/control-plane"
              operator: Exists
tolerations:
  - key: node-role.kubernetes.io/master
    operator: Exists
    effect: NoSchedule
  - key: node-role.kubernetes.io/control-plane
    operator: Exists
    effect: NoSchedule
kamaji-etcd:
  persistentVolumeClaim:
    storageClassName: local-path
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
          - matchExpressions:
              - key: "node-role.kubernetes.io/master"
                operator: Exists
          - matchExpressions:
              - key: "node-role.kubernetes.io/control-plane"
                operator: Exists
  tolerations:
    - key: node-role.kubernetes.io/master
      operator: Exists
```

```
      effect: NoSchedule
    - key: node-role.kubernetes.io/control-plane
      operator: Exists
      effect: NoSchedule
  jobs:
    affinity:
      nodeAffinity:
        requiredDuringSchedulingIgnoredDuringExecution:
          nodeSelectorTerms:
            - matchExpressions:
                - key: "node-role.kubernetes.io/master"
                  operator: Exists
            - matchExpressions:
                - key: "node-role.kubernetes.io/control-plane"
                  operator: Exists
    tolerations:
      - key: node-role.kubernetes.io/master
        operator: Exists
        effect: NoSchedule
      - key: node-role.kubernetes.io/control-plane
        operator: Exists
        effect: NoSchedule
  datastore:
    enabled: true
    annotations:
      helm.sh/resource-policy: keep
    name: default
image:
  repository: ghcr.io/nvidia/kamaji
  tag: v1.34.0-25.9.3
  pullPolicy: Always
cfssl:
  image:
    tag: v1.6.5
```

local-path-provosioner

```
tolerations:
  - operator: Exists
    effect: NoSchedule
    key: node-role.kubernetes.io/control-plane
  - operator: Exists
    effect: NoSchedule
    key: node-role.kubernetes.io/master
```

kube-state-metrics

```
# Kube State Metrics configuration
affinity:
  nodeAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
      nodeSelectorTerms:
        - matchExpressions:
            - key: "node-role.kubernetes.io/master"
              operator: Exists
        - matchExpressions:
            - key: "node-role.kubernetes.io/control-plane"
              operator: Exists
tolerations:
  - key: node-role.kubernetes.io/master
    operator: Exists
    effect: NoSchedule
  - key: node-role.kubernetes.io/control-plane
    operator: Exists
    effect: NoSchedule
extraArgs:
  - --custom-resource-state-config-
file=/etc/customresourcestate/config.yaml
volumes:
  - configMap:
      defaultMode: 420
      name: dpf-operator-customresourcestate-config
    name: customresourcestate-config
volumeMounts:
  - mountPath: /etc/customresourcestate
    name: customresourcestate-config
    readOnly: true
rbac:
  extraRules:
    - apiGroups:
        - svc.dpu.nvidia.com
        - operator.dpu.nvidia.com
```

```
        - svc.dpu.nvidia.com
        - provisioning.dpu.nvidia.com
      resources:
        - dpuclusters
        - dpuclusters/status
        - dpudeployments
        - dpudeployments/status
        - dpuservicecredentialrequests
        - dpuservicecredentialrequests/status
        - dpuservices
        - dpuservices/status
        - dpuservicetemplates
        - dpuservicetemplates/status
        - dpfoperatorconfigs
        - dpfoperatorconfigs/status
        - dpuservicechains
        - dpuservicechains/status
        - dpuserviceinterfaces
        - dpuserviceinterfaces/status
        - dpuserviceipams
        - dpuserviceipams/status
        - dpus
        - dpus/status
        - bfbs
        - bfbs/status
      verbs: ["list", "watch"]
    - apiGroups: ["apiextensions.k8s.io"]
      resources: ["customresourcedefinitions"]
      verbs: ["list", "watch"]
```

grafana

```
# Grafana configuration
affinity:
  nodeAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
      nodeSelectorTerms:
        - matchExpressions:
            - key: "node-role.kubernetes.io/master"
              operator: Exists
        - matchExpressions:
            - key: "node-role.kubernetes.io/control-plane"
              operator: Exists
tolerations:
  - key: node-role.kubernetes.io/master
    operator: Exists
    effect: NoSchedule
  - key: node-role.kubernetes.io/control-plane
    operator: Exists
    effect: NoSchedule
persistence:
  enabled: true
  storageClassName: local-path
datasources:
  datasources.yaml:
    apiVersion: 1
    datasources:
      - name: prometheus
        type: prometheus
        url: http://prometheus-server
        access: proxy
        isDefault: true
dashboardProviders:
  dashboardproviders.yaml:
    apiVersion: 1
    providers:
      - name: 'default'
```

```
        orgId: 1
        folder: ''
        type: file
        disableDeletion: false
        editable: true
        options:
          path: /var/lib/grafana/dashboards/default
      - name: 'debug'
        orgId: 1
        folder: 'debug'
        type: file
        disableDeletion: false
        editable: true
        options:
          path: /var/lib/grafana/dashboards/debug
dashboardsConfigMaps:
  default: dpf-operator-grafana-dashboards
  debug: dpf-operator-grafana-debug-dashboards
```

prometheus

```
# Prometheus configuration
server:
  persistentVolume:
    storageClass: local-path
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
          - matchExpressions:
              - key: "node-role.kubernetes.io/master"
                operator: Exists
          - matchExpressions:
              - key: "node-role.kubernetes.io/control-plane"
                operator: Exists
  tolerations:
    - key: node-role.kubernetes.io/master
      operator: Exists
      effect: NoSchedule
    - key: node-role.kubernetes.io/control-plane
      operator: Exists
      effect: NoSchedule
alertmanager:
  enabled: false
prometheus-node-exporter:
  enabled: false
prometheus-pushgateway:
  enabled: false
kube-state-metrics:
  enabled: false
extraScrapeConfigs: |
  - job_name: 'doca-platform-framework'
    scrape_interval: 15s
    metrics_path: /metrics
    scheme: https
    authorization:
```

```
      type: Bearer
      credentials_file:
/var/run/secrets/kubernetes.io/serviceaccount/token
    tls_config:
      ca_file:
/var/run/secrets/kubernetes.io/serviceaccount/ca.crt
      insecure_skip_verify: true
    kubernetes_sd_configs:
      - role: pod
    relabel_configs:
      - source_labels:
[__meta_kubernetes_pod_label_dpu_nvidia_com_component]
        action: keep
        regex: ".*-controller-manager"
      - source_labels:
[__meta_kubernetes_pod_container_port_name]
        action: keep
        regex: metrics
```

Lifecycle Management

DPF Lifecycle Management

Guidance for managing the lifecycle of your DOCA Platform Framework (DPF) deployment.

DPF Operator Lifecycle

When a new major version is released:

- New major version is supported
- Previous major version enters EOL (end-of-life)

Upgrade Policy: DPF follows an **n-1 upgrade policy**. You can upgrade from the previous major version to the current major version. Patch upgrades are always supported. Skipping major versions is not supported.

Ensuring Deployment Readiness

Use `dpfctl` to verify that your DPFOperatorConfig and all DPU resources are in a `Ready` state before performing maintenance operations.

DPF Operator Upgrade

DPF Operator Upgrade Guide - Detailed instructions for upgrading DPF Operator with automated prevalidations and safe rollback capabilities.

DPF Operator Upgrade Guide

DPF Operator Version Migration Guides

This document describes the process of upgrading between different versions of the DPF Operator.

Prerequisites

Before starting the upgrade process, ensure you have the following Helm Prerequisites installed and configured.

Further prerequisites include:

- All DPF Operator components MUST be in a **ready state**
- All DPUs MUST be in a **ready and healthy state**
- Always **review the release notes** for the version you're upgrading to
- Release notes contain critical information about:
    - New features and capabilities
    - Bug fixes and improvements
    - Breaking changes and migration requirements
    - Known issues and workarounds
    - Configuration changes that may be required

Verifying Readiness

You can verify the readiness of your DPF Operator and DPUs using `dpfctl` or `kubectl`:

Using dpfctl (recommended):

```
# Check overall DPF Operator status
dpfctl describe all --show-resources=dpfoperatorconfig

# Check all conditions for troubleshooting
dpfctl describe all --show-resources=dpfoperatorconfig --show-conditions=all
```

The `dpfctl` tool provides a more user-friendly view of the DPFOperatorConfig status and all related conditions, making it easier to identify validation issues.

Using kubectl:

```
kubectl get dpfoperatorconfig -o yaml
```

All status conditions in the status section should be `True`, and the `version` field should match the current DPF Operator version.

Supported Upgrade Path (n-1 Policy)

Important: DPF Operator follows an **n-1 upgrade policy** with the version pattern `year.month.patchVersion`. You can upgrade from the previous release version to the current release version. Patch-level upgrades within the same minor version are always allowed.

 Supported:

- v25.7.0 → v25.10.0
- v25.7.0 → v25.7.1

 Not Supported:

- v25.4 → v25.7 (unsupported version upgrade)

- v25.7 → next GA release (skipping version v25.10)

Why this matters: Skipping versions can lead to:

- Failed upgrades due to missing intermediate API changes
- Data corruption or loss
- Incompatible resource states
- Unsupported configuration combinations

Prevalidation

Before the DPF Operator upgrades any components, the system performs prevalidation checks to ensure the cluster is in a safe state for the upgrade. These validations are **automatically executed** and **must pass** before the upgrade can continue.

- Version Compatibility Validation
  - **DPF Version Check**: Validates that the current DPF version is compatible with the target version
  - **Supported Upgrade Path**: Ensures you're upgrading from a supported version (n-1 policy)
- System Components Readiness
  - The operator validates that all critical system components are ready for upgrade
- DPU State Validation
  - **DPU Readiness**: All DPUs must be in a ready state
  - **DPF Version Compatibility**: Each DPU's DPF version must be compatible with the upgrade
  - **DPU Health**: All DPUs must be healthy and operational

Validation Failure Handling

If any validation fails:

- **Upgrade Blocked**: The upgrade process is automatically halted
- **Status Updates**: The `DPFOperatorConfig` status reflects the validation failure

Status Monitoring

You can monitor prevalidation status using either dpfctl or kubectl as described in the prerequisites above. The `DPFOperatorConfig` status will indicate whether prevalidations have passed or failed. All conditions must be `True` for the upgrade to proceed.

What to Do Next

After completing your DPF Operator upgrade, follow these steps to ensure everything is working correctly:

1. **Verify DPFOperatorConfig Status**: Verify that your DPFOperatorConfig custom resource is in a ready state and all conditions are met.
2. **Update to Supported BFB and DPUFlavor**: Create the new BFB object and DPUFlavor object according to the new guide. Review and update your DPUDeployment/DPUSet resources as needed to reference those objects to make the system compatible with the new operator version.
3. **Update DPUServices**: Review and update your DPUDeployment/DPUService resources to use services that are compatible with BFB version and operator version.
4. **Monitor the Upgrade**: Use `dpfctl describe all` to monitor the reconciliation process and ensure all resources reach ready state.

Platform Support

Prerequisites

Component	Version	Notes
Kubernetes	1.32-1.34	All currently supported upstream Kubernetes releases are supported
Helm	v3.5+	For information and methods of Helm installation, please refer to the official Helm Website.

DPF Component Matrix

DPF uses the following components:

Component	Origin	Repository	Image Name	Tag	Note
ArgoCD	Community (OSS)	quay.io/argoproj	argocd	v2.14.2	
Cert Manager Controller	Community (OSS)	quay.io/jetstack	cert-manager-controller	v1.18.1	
Flannel	Community (OSS)	docker.io/flannel	flannel	v0.26.5	
Multus	Community (OSS)	ghcr.io/k8snetworkplumbingwg	multus-cni	v3.9.3	
NVIDIA K8s IPAM	Community (OSS)	ghcr.io/mellanox	nvidia-k8s-ipam	v0.3.5	
NVIDIA Maintenance Operator	Community (OSS)	ghcr.io/mellanox	maintenance-operator	v0.2.2	
NVIDIA Network Operator	Community (OSS)	nvcr.io/nvidia/cloud-native	network-operator	v25.7.0	

Component	Origin	Repository	Image Name	Tag	Note
Node Feature Discovery	Community (OSS)	registry.k8s.io/nfd	node-feature-discovery	v0.17.1	
SR-IOV Device Plugin	Community (OSS)	ghcr.io/k8snetworkplumbingwg	sriov-network-device-plugin	v3.9.0	
Etcd Defrag	Community (OSS)	ghcr.io/ahrtr	etcd-defrag	v0.22.0	Optic
Kamaji	Community (OSS)	ghcr.io/clastix	kamaji	v1.34.0	Optic
Keepalived	Community (OSS)	docker.io/osixia	keepalived	2.0.20	Optic
Local Path Provisioner	Community (OSS)	docker.io/rancher	local-path-provisioner	v0.0.31	Optic
Kube State Metrics	Community (OSS)	registry.k8s.io/kube-state-metrics	kube-state-metrics	v2.13.0	Optic
Prometheus	Community (OSS)	quay.io/prometheus	prometheus	v2.54.1	Optic
Grafana	Community (OSS)	docker.io/grafana	grafana	11.1.0	Optic

Tested Network Adapters

The following NVIDIA BlueField 3 DPU models are recommended for DPF:

- B3240
- B3220
- B3210

Tested Operating Systems and Kubernetes Versions

NVIDIA DPF has been validated in the following scenarios:

Operating System	Kubernetes	Notes
Ubuntu 24.04 LTS	1.30	

Tested Container Runtimes

NVIDIA DPF has been validated in the following scenarios:

Operating System	Containerd	CRI-O	Notes
Ubuntu 24.04 LTS	Yes	No	

Release Notes

This section contains release notes for the DOCA Platform Framework (DPF), documenting new features, improvements, and bug fixes for each version.

- v25.10.0
- v25.7.1
- v25.7.0
- v25.4.0
- v25.1.1
- v25.1.0
- v24.10.0

DOCA Platform Framework v24.10.0

This is the first release of the DOCA Platform Framework (DPF), focused to streamline the provisioning and orchestration of NVIDIA BlueField DPUs in Kubernetes environments.

Features

- Kubernetes integration - extends K8s API to manage BlueField DPUs and DPU services in a declarative manner
- Automated cluster-wide orchestration - enables discovery, provisioning, and configuration of BlueField DPU devices in K8s clusters
- Software Consistency - synchronizes firmware, OS, DOCA, and DPU services
- Lifecycle Management - manages lifecycles of hosts and DPUs seamlessly
- Service Orchestration - handles installation, upgrades, and removal of DPU services = Service Function Chaining (SFC) - supports chaining of DPU services on DPF-

provisioned DPUs

Supported DPU Services:

- OVN-Kubernetes with OVS offload to the DPU
- DOCA Host-Based Networking (HBN)
- DOCA Telemetry Service (DTS)
- DOCA BlueMan

Dependencies

Hardware and Software Requirements

- DPU Hardware: NVIDIA BlueField-3 DPUs
- Minimal DOCA BFB Image: DOCA v2.5 or higher (must be pre-installed on DPUs to support DPF provisioning)
- Provisioned DOCA BFB Image: DOCA v2.9

Refer to Prerequisites for detailed requirements.

Known Issues and Limitations

1. **Title:** On initial deployment DPF CRs maybe stuck in initial state (Pending, Initializing, etc.) and not progressing

   Description:

   In case DPF CRs were created before DPF components are running they maybe be stuck in their initial state. DPF CRs need to be created after the DPF components have been deployed. In case CRs were created before they may remain in an initial state.

   Internal Ref #4241297

   Workaround:

Delete any CRs that were created before the System components have been deployed and recreate them.

2. **Title:** DPUService stuck in its Deleting phase

   Description:

   DPUService can be stuck in its Deleting phase when a pod on the DPU created as part of the DPUService can not be deleted.

   Internal Ref #4213229

   Workaround:

   Reprovision the DPU where the DPUService is deployed.

3. **Title:** DPU Cluster CNI not deployed when hitting docker rate limit

   Description:

   Flannel, the DPU Cluster default CNI can fail getting deployed if we have exceeded docker rate limit - this can happen since we pull flannel directly from Docker Hub.

   Internal Ref #4214500

   Workaround:

   Workaround is to add personal docker account to pull images, follow the bellow process:

   1. Create the docker pull image secret:

      1. Create the secret (add your personal user and access token):

         ```
         kubectl -n dpf-operator-system create secret docker-registry dockerio-secret --docker-server=docker.io --docker-username=<username> --docker-password=<personal access token>
         ```

      2. label the secret as a DPF image pull secret:

         ```
         kubectl -n dpf-operator-system label secret dockerio-secret dpu.nvidia.com/image-pull-secret=""
         ```

2. Disable flannel:
    1. Option 1: On a clean install (before deploying DPF controllers)
        1. Add secret to dpfoperatorconfig.spec.ImagePullSecrets list
        2. Disable flannel from the config spec.flannel.disable: true
    2. Option 2: Running cluster
        1. Patch the DPFOperatorConfig object to disable flannel DPUService:

            ```
            kubectl -n dpf-operator-system patch dpfoperatorconfig dpfoperatorconfig -p '{"spec":{"flannel":{"disable":true}}}' --type=merge
            ```

        2. Remove the existing flannel DPUService
3. Deploy Flannel as a DPUService with the correct image pull secret, create the following DPUService:

    ```
    yaml apiVersion: svc.dpu.nvidia.com/v1alpha1 kind: DPUService metadata: name: flannel namespace: dpf-operator-system spec: helmChart: source: repoURL: https://flannel-io.github.io/flannel chart: flannel version: v0.25.1 values: global: imagePullSecrets: - name: dockerio-secret
    ```

4. **Title:** Host DPU PF0 shouldn't be assigned to workload pods

    Description:

    Creating a NetworkAttachmentDefinition CR with host-device plugin to assign the DPU physical network device (PF) for a pod will break host and DPF won't be able to manage it.

    Internal Ref #4217296

    Workaround:

    Avoid creating NetworkAttachmentDefinition CR with host-device plugin to assign the DPU physical network device on the host to a pod, only assign SR-IOV Virtual Functions (VF) to workload pods as host-device interfaces

5. **Title:** Incompatible DPU flavor can cause DPU to get into an unstable state

Description:

Using an incompatible DPU flavor can cause the DPU Device to get into an error state which requires manual intervention.

For example allocating 14GB of hugepages in a DPU of 16GB memory.

Internal Ref #4200717

Workaround:

Manually provision DPU or follow DOCA troubleshooting documentation to return DPU to operational state https://docs.nvidia.com/networking/display/bfswtroubleshooting.

6. **Title:** DPUServiceInterface is not Immutable

   Description:

   DPUServiceInterface which are used by a DPUDeployment can be manually deleted which leads to unsupported behavior.

   Internal Ref #4217384

   Workaround:

   Redeploy DPUDeployment object.

7. **Title:** DPUSet which is managed by DPUDeployment can be modified

   Description:

   DPUSet which is managed by DPUDeployment can be modified which leads to mismatch in DPUDeployment, and recreation of old DPUSet when DPUDeployment is updated.

   Internal Ref #4217345

   Workaround:

   Avoid updating DPUSet manually in case it was created from a DPUDeployment.

8. **Title:** DpfOperatorConfig CR deletion hangs on uninstallation process

   Description:

When uninstalling DPF the DpfOperatorConfig CR might hang and not get deleted, due to leftover DPUServiceChains objects that need to be manually deleted.

Internal Ref #4214285

Workaround:

Removal all DPUServiceInterfaces remaining.

9. **Title:** Traffic loss after reconfiguration of DPUServices with chain between

   Description:

   Reconfiguration of DPUServices with chain between them may cause traffic loss due to outdated service chains.

   Internal Ref #4178445

   Workaround:

   Recreated SFC object between services.

10. **Title:** Stale ports after DPU reboot

    Description:

    When rebooting DPU, the old DPU service ports won't get deleted from DPU's OVS and would be stale

    Internal Ref #4174183

    Workaround:

    No workaround, known issue, shouldn't affect performance.

11. **Title:** BFB filename must be unique

    Description:

    If BFB CR#1 bfb.spec.filename is the same as a BFB CR#2 bfb.spec.filename but references a different URL (actual bfb file to download) then BFB CR#1 would reference the wrong bfb.

    Internal Ref #4143309

Workaround:

Use unique bfb.spec.filename when creating new bfb CRs.

12. **Title:** DPU Cluster control-plane connectivity is lost when physical port P0 is down on the worker node

    Description:

    Link down of p0 port on the DPU will result in DPU control plane connectivity loss of DPU components.

    Internal Ref #3751863

    Workaround:

    Make sure P0 link is up on the DPU, if down either restart DPU or refer to DOCA troubleshooting https://docs.nvidia.com/networking/display/bfswtroubleshooting.

13. **Title:** DPU Provisioning operations wouldn't be retried

    Description:

    DPU Provisioning operations wouldn't be retried, this can lead to DPU object in an error phase because of small environment glitch which would have worked if retried.

    Internal Ref #4202272

    Workaround:

    Delete the DPU object in an error phase which will cause it to get recreated and operation to begin from scratch.

14. **Title:** Changes to a DPU Object will trigger reprovisioning

    Description:

    Any edits or changes to a DPU Object will cause it to recreate even if the changes are small and doesn't require DPU reprovisioning such as changing the labels on the Node for the DPU in the DPU Cluster.

    Internal Ref #4206910

    Workaround:

No workaround, known limitation.

15. **Title:** Cluster MTU value cannot be dynamically changed

    Description:

    It is possible to deploy DPF cluster with a custom MTU value, however once deployed, it is not possible to modify the MTU value which is applied on multiple distributed components.

    Internal Ref #4241316

    Workaround:

    Uninstall DPF and re-install from scratch using the new MTU value.

16. **Title:** nvidia-k8s-ipam and servicechainset-controller DPF system DPUServices are in “Pending” phase

    Description:

    As long as there are no provisioned DPUs in the system, the nvidia-k8s-ipam and servicechainset-controller will appear as not ready / pending when querying dpuservices.

    This has no impact on performance or functionality since DPF system components are only relevant when there are DPUs to provision services on.

    Internal Ref #4241324

    Workaround: No workaround, known issue

17. **Title:** DPU object in pending phase without clear reason

    Description:

    DPU object wait in pending phase in case the BFB Object it references doesn't exist or none of the BFB objects are in Ready phase, this is the required behavior but doesn't have clear indication on DPU object.

    Internal Ref #4203694

    Workaround:

    Fix referenced BFB object.

18. **Title:** [OVN Kubernetes DPUService] Nodes marked as NotReady

    Description:

    When installing OVN Kubernetes as a CNI on a node running containerd version 1.7.0 and above the Node never becomes ready.

    Internal Ref #4241282

    Workaround:

    Option 1: Use containerd version below v1.7.0 when using OVN Kubernetes as a primary CNI.

    Option 2: Manually restart containerd on the host.

19. **Title:** [OVN Kubernetes DPUService] Fragmented packets are silently dropped when using custom MTU

    Description:

    When using custom MTU configuration, workload packets which require IP fragmentation (with size exceeding configured custom MTU) will be silently dropped.

    Internal Ref #4241299

    Workaround:

    None. For custom MTU configuration follow the DPF Reference Deployment Guide.

20. **Title:** [OVN Kubernetes DPUService] control plane node is not functional after reboot or network restart

    Description:

    During OVN Kubernetes CNI installation on the control plane nodes, the management interface is moved with its IP into a newly created OVS bridge.

    Since this network configuration is not persistent it will be lost during node or network restart.

    Internal Ref #4241306

    Workaround:

1. Pre-define the OVS bridge on each control plane node with the OOB port MAC and IP address and ensure it gets a persistent IP

   ```yaml

   # Ubuntu example for netplan persistent network configuration:

   network: ethernets: oob: match: # the mac address of the oob macaddress: xx:xx:xx:xx:xx:xx set-name: oob bridges: br0: addresses: x.x.x.x/x interfaces: [oob] # the mac address of the oob macaddress: xx:xx:xx:xx:xx:xx openvswitch: {} version: 2 ```

2. Set OVS bridge "bridge-uplink" in OVS metadata.

   ```
   bash ovs-vsctl br-set-external-id br0 bridge-id br0 -- br-set-external-id br0 bridge-uplink oob
   ```

21. **Title:** [OVN Kubernetes DPUService] Only a single OVN-Kubernetes DPU service version can be deployed across the cluster

    **Description:**

    OVN-Kubernetes service does not fully support customization using Helm parameters, meaning we only support a single OVN-Kubernetes DPU service across the entire cluster.

    Internal Ref #4209524

    **Workaround:**

    No workaround, known limitation.

22. **Title:** [OVN Kubernetes DPUService] Lost traffic from workloads to control plane components or K8S services after dpu reboot, port flapping, ovs restart or manual network configuration

    **Description:**

    Connectivity issues between workload pods to control plane components or K8S services may occur after the following events: DPU reboot without host reboot, high speed port flapping (link down/up), ovs restart, DPU network configuration change (for example using "netplan apply" command on DPU).
   ```

The issues are caused by network configuration that was applied by ovn CNI on DPUs and won't get reapplied automatically.

When rebooting DPU without the host, or high speed port link is going down/up, or manually changing dpu network (for example with netplan apply), network configuration which was applied by the dpu CNI components may be lost and won't reapply automatically.

Internal Ref #4191019

Workaround:

Need to recreate the ovnk8s node pod on the host to reapply the configuration.

23. **Title:** [OVN Kubernetes DPUService] host network configuration may result in lost traffic from host workloads (on overlay)

    Description:

    When changing host network (for example with netplan apply) custom network configuration which is done by the host CNI components may be lost and won't reapply automatically.

    Internal Ref #4188044

    Workaround:

    Need to recreate the ovnk8s node pod on the host to reapply the configuration.

24. **Title:** [OVN DPUService] traffic between workloads stops working after 5 minutes

    Description:

    Traffic between workloads (not hostnetwork) stops working after 5 minutes of workload creation.

    Internal Ref #4224295

    Workaround:

    Set the DHCP server for high speed network fabric (fabric connected to DPU ports) lease time to a short interval (less than 5 minutes).

25. **Title:** [OVN DPUService] No network connectivity for SR-IOV accelerated workload pods after DPU reboot

Description:

SR-IOV accelerated workload pod is losing its VF interface upon DPU reboot. VF is available on the host however not injected back into the pod.

Internal Ref #4234474

Workaround:

Recreate the SR-IOV accelerated workload pods.

26. **Title:** [OVN DPUService] DPU re-provisioning or VTEP IP change may result in lost traffic between cluster components

    Description:

    In a L2 setup with OVN, DPU VTEP IP address change caused by DPU re-provisioning or DHCP IP re-assignment, may lead to lost traffic between control plane, workers and K8s services components, due to stale DPU VTEP IP information on ovnk8s components across the cluster.

    Internal Ref #4238288, #4241388

    Workaround:

    Recreate the ovnkube-node pods on all control plane nodes and the ovnk8s DPUService to make sure all ovnk8s components are aligned with the new VTEP IPs.

27. **Title:** [HBN DPUService] HBN DPUService cannot dynamically reload configurations

    Description:

    When updating HBN configuration through a configmap, the running HBN container won't reload it, and need to get restarted with the new configuration.

    Internal Ref #4217788

    Workaround:

    Recreate HBN DPU service after changing configuration.

28. **Title:** [HBN DPUService]Invalid HBN configuration is not reflected to user in case it is syntactically valid

    Description:

If the HBN YAML configuration is valid but contains values that are illegal from an NVUE perspective then the HBN service will start with the last known valid configuration and it won't be reflected to the end user.

Internal Ref #

Workaround:

No workaround, known issue.

29. **Title:** [HBN + OVN DPUServices] HBN service restarts on DPU causes worker to lose traffic

    Description:

    If the HBN pod on the DPU will reset then the workloads on the host (any traffic on the OVN overlay) will not receive traffic.

    Internal Ref #4220185, #4223176

    Workaround:

    power cycle the host.

30. **Title:** [DTS DPUservice] DTS appears as OutOfSync

    Description:

    When creating a DPUDeployment for DTS DPU service, the DPUService object can be marked as OutOfSync although the pods are running on the DPUs.

    Internal Ref #4182929

    Workaround:

    No workaround, known issue.

DOCA Platform Framework v25.1.0

This patch release of the DOCA Platform Framework (DPF) includes bug fixes and improvements to enhance the provisioning and orchestration of NVIDIA BlueField DPUs in Kubernetes environments.

Features

- Enabled etcd compaction and added an etcd-defrag CronJob when using Kamaji clusters to optimize database performance
- Introduced a new CLI tool, `dpfctl`, for debugging and troubleshooting DPF components

Fixed Issues from Previous Release

> **Note**
>
> Detailed information about the fixed issues can be found in the release notes for v24.10.

- Avoid race condition when multiple DPUDeployments depend on same objects
- DPFOperatorConfig CR deletion hangs on uninstallation process
- Changes to a DPU Object will trigger reprovisioning
- DPU object in pending phase without clear reason
- DPU Cluster CNI not deployed when hitting docker rate limit
- DPUServiceInterface is not Immutable
- [OVN-Kubernetes DPUService] Fragmented packets are silently dropped when using custom MTU
- [OVN-Kubernetes DPUService] traffic between workloads stops working after 5 minutes
- [OVN-Kubernetes DPUService] DPU re-provisioning or VTEP IP change may result in lost traffic between cluster components

Supported DPU Services:

- OVN-Kubernetes with OVS offload to the DPU
- DOCA Host-Based Networking (HBN)
- DOCA Telemetry Service (DTS)
- DOCA BlueMan

Dependencies

Hardware and Software Requirements

> **Note**
>
> Refer to Prerequisites for detailed requirements.

- DPU Hardware: NVIDIA BlueField-3 DPUs
- Minimal DOCA BFB Image: DOCA v2.5 or higher (must be pre-installed on DPUs to support DPF provisioning)
- Provisioned DOCA BFB Image: `bf-bundle-2.9.1-40`

Known Issues and Limitations

- **On initial deployment DPF CRs maybe stuck in initial state (Pending, Initializing, etc.) and not progressing**
    - In case DPF CRs were created before DPF components are running they maybe be stuck in their initial state. DPF CRs need to be created after the DPF components have been deployed. In case CRs were created before they may remain in an initial state.
    - Internal Ref #4241297

    - **Workaround:** Delete any CRs that were created before the System components have been deployed and recreate them.

- **DPUService stuck in its Deleting phase**

    - DPUService can be stuck in its Deleting phase when a pod on the DPU created as part of the DPUService can not be deleted.

    - Internal Ref #4213229

    - **Workaround:** Reprovision the DPU where the DPUService is deployed.

- **Incompatible DPUFlavor can cause DPU to get into an unstable state**

    - Using an incompatible DPUFlavor can cause the DPU Device to get into an error state which requires manual intervention. For example allocating 14GB of hugepages in a DPU of 16GB memory.

    - Internal Ref #4200717

    - **Workaround:** Manually provision DPU or follow DOCA troubleshooting documentation to return DPU to operational state https://docs.nvidia.com/networking/display/bfswtroubleshooting.

- **Traffic loss after reconfiguration of DPUServices with chain between**

    - Reconfiguration of DPUServices with chain between them may cause traffic loss due to outdated service chains.

    - Internal Ref #4178445

    - **Workaround:** Recreated SFC object between services.

- **Stale ports after DPU reboot**

    - When rebooting DPU, the old DPU service ports won't get deleted from DPU's OVS and would be stale

    - Internal Ref #4174183

    - **Workaround:** No workaround, known issue, shouldn't affect performance.

- **BFB filename must be unique**

    - If BFB CR#1 bfb.spec.filename is the same as a BFB CR#2 bfb.spec.filename but references a different URL (actual bfb file to download) then BFB CR#1 would reference the wrong bfb.
    - Internal Ref #4143309
    - **Workaround:** Use unique bfb.spec.filename when creating new bfb CRs.
- **DPU Cluster control-plane connectivity is lost when physical port P0 is down on the worker node**
    - Link down of p0 port on the DPU will result in DPU control plane connectivity loss of DPU components.
    - Internal Ref #3751863
    - **Workaround:** Make sure P0 link is up on the DPU, if down either restart DPU or refer to DOCA troubleshooting https://docs.nvidia.com/networking/display/bfswtroubleshooting.
- **DPU Provisioning operations wouldn't be retried**
    - DPU Provisioning operations wouldn't be retried, this can lead to DPU object in an error phase because of small environment glitch which would have worked if retried.
    - Internal Ref #4202272
    - **Workaround:** Delete the DPU object in an error phase which will cause it to get recreated and operation to begin from scratch.
- **Cluster MTU value cannot be dynamically changed**
    - It is possible to deploy DPF cluster with a custom MTU value, however once deployed, it is not possible to modify the MTU value which is applied on multiple distributed components.
    - Internal Ref #3917006
    - **Workaround:** Uninstall DPF and re-install from scratch using the new MTU value.
- **nvidia-k8s-ipam and servicechainset-controller DPF system DPUServices are in "Pending" phase**

- As long as there are no provisioned DPUs in the system, the nvidia-k8s-ipam and servicechainset-controller will appear as not ready / pending when querying dpuservices.

  This has no impact on performance or functionality since DPF system components are only relevant when there are DPUs to provision services on.

- Internal Ref #4241324

- **Workaround:** No workaround, known issue

- **[OVN-Kubernetes DPUService] Nodes marked as NotReady**

  - When installing OVN-Kubernetes as a CNI on a node running containerd version 1.7.0 and above the Node never becomes ready.

  - Internal Ref #4178221

  - **Workaround:**

    Option 1: Use containerd version below v1.7.0 when using OVN-Kubernetes as a primary CNI.

    Option 2: Manually restart containerd on the host.

- **[OVN-Kubernetes DPUService] control plane node is not functional after reboot or network restart**

  - During OVN-Kubernetes CNI installation on the control plane nodes, the management interface is moved with its IP into a newly created OVS bridge. Since this network configuration is not persistent it will be lost during node or network restart.

  - Internal Ref #4241306

  - **Workaround:**

    1. Pre-define the OVS bridge on each control plane node with the OOB port MAC and IP address and ensure it gets a persistent IP

```
yaml #Ubuntu example for netplan persistent network
configuration: network: ethernets: oob: match: # the mac
address of the oob macaddress: xx:xx:xx:xx:xx:xx set-name:
oob bridges: br0: addresses: x.x.x.x/x interfaces: [oob] #
the mac address of the oob macaddress: xx:xx:xx:xx:xx:xx
openvswitch: {} version: 2
```

1. Set OVS bridge "bridge-uplink" in OVS metadata.

```
bash ovs-vsctl br-set-external-id br0 bridge-id br0 -- br-
set-external-id br0 bridge-uplink oob
```

- **[OVN-Kubernetes DPUService] Only a single OVN-Kubernetes DPU service version can be deployed across the cluster**
  - OVN-Kubernetes service does not fully support customization using Helm parameters, meaning we only support a single OVN-Kubernetes DPU service across the entire cluster.
  - Internal Ref #4209524
  - **Workaround:** No workaround, known limitation.
- **[OVN-Kubernetes DPUService] Lost traffic from workloads to control plane components or K8S services after dpu reboot, port flapping, ovs restart or manual network configuration**
  - Connectivity issues between workload pods to control plane components or K8S services may occur after the following events: DPU reboot without host reboot, high speed port flapping (link down/up), ovs restart, DPU network configuration change (for example using "netplan apply" command on DPU).

    The issues are caused by network configuration that was applied by ovn CNI on DPUs and won't get reapplied automatically.

    When rebooting DPU without the host, or high speed port link is going down/up, or manually changing dpu network (for example with netplan apply), network configuration which was applied by the dpu CNI components may be lost and won’t reapply automatically.
  - Internal Ref #4202272
  - **Workaround:** Recreate the OVN-Kubernetes node pod on the host to reapply the configuration.

- **[OVN-Kubernetes DPUService] host network configuration may result in lost traffic from host workloads (on overlay)**
    - When changing host network (for example with netplan apply) custom network configuration which is done by the host CNI components may be lost and won't reapply automatically.
    - Internal Ref #4188044
    - **Workaround:** Recreate the OVN-Kubernetes node pod on the host to reapply the configuration.
- **[OVN-Kubernetes DPUService] No network connectivity for SR-IOV accelerated workload pods after DPU reboot**
    - SR-IOV accelerated workload pod is losing its VF interface upon DPU reboot. VF is available on the host however not injected back into the pod.
    - Internal Ref #4236521
    - **Workaround:** Recreate the SR-IOV accelerated workload pods.
- **[HBN DPUService] HBN DPUService cannot dynamically reload configurations**
    - When updating HBN configuration through a configmap, the running HBN container won't reload it, and need to get restarted with the new configuration.
    - Internal Ref #4290426
    - **Workaround:** Recreate HBN DPU service after changing configuration.
- **[HBN DPUService] Invalid HBN configuration is not reflected to user in case it is syntactically valid**
    - If the HBN YAML configuration is valid but contains values that are illegal from an NVUE perspective then the HBN service will start with the last known valid configuration and it won't be reflected to the end user.
    - **Workaround:** No workaround, known issue.
- **[HBN + OVN-Kubernetes DPUServices] HBN service restarts on DPU causes worker to lose traffic**

    - If the HBN pod on the DPU will reset then the workloads on the host (any traffic on the OVN overlay) will not receive traffic.
    - Internal Ref #4220185, #4223176
    - **Workaround:** Wait 15 minutes for the system to recover or restart the OVN-Kubernetes Pod on that particular DPU.
- **[DTS DPUService] DTS appears as OutOfSync**
    - When creating a DPUDeployment for DTS DPU service, the DPUService object can be marked as OutOfSync although the pods are running on the DPUs.
    - Internal Ref #4182929
    - **Workaround:** No workaround, known issue.
- **[OVN-Kubernetes DPUService] SR-IOV test Pod cannot reach Kubernetes API Service**
    - When running a SR-IOV test Pod, the pod cannot reach a Kubernetes API Service. The issue is that the related conntrack entries miss the un-nat sometimes.
    - Internal Ref #4313629
    - **Workaround:** Unblock is to run the following on the DPUs:
      ```
      ovs-appctl revalidator/purge ovs-appctl dpctl/flush-conntrack
      ```

DOCA Platform Framework v25.1.1

This patch release of the DOCA Platform Framework (DPF) includes bug fixes and improvements to enhance the provisioning and orchestration of NVIDIA BlueField DPUs in Kubernetes environments.

Fixed Issues from Previous Release

> **Note**
>
> Detailed information about the fixed issues can be found in the release notes for v25.1.

Known Issues with New Workarounds

- **[HBN + OVN-Kubernetes DPUServices] HBN service restarts on DPU causes worker to lose traffic**
  - If the HBN pod on the DPU will reset then the workloads on the host (any traffic on the OVN overlay) will not receive traffic.
  - Internal Ref #4220185, #4223176
  - **Workaround:** Deploy the HBN DPUService with the below values to update the helm chart and image tag:
    ```
    yaml spec: helmChart: source: repoURL: $NGC_HELM_REGISTRY_REPO_URL version: 1.0.3 chart: doca-hbn values: image: repository: $HBN_NGC_IMAGE_URL tag: 2.4.2-doca2.9.2-32
    ```

Supported DPU Services:

- OVN-Kubernetes with OVS offload to the DPU
- DOCA Host-Based Networking (HBN)
- DOCA Telemetry Service (DTS)
- DOCA BlueMan

Dependencies

Hardware and Software Requirements

> **Note**
>
> Refer to Prerequisites for detailed requirements.

- DPU Hardware: NVIDIA BlueField-3 DPUs
- Minimal DOCA BFB Image: DOCA v2.5 or higher (must be pre-installed on DPUs to support DPF provisioning)
- Provisioned DOCA BFB Image: `bf-bundle-2.9.1-40`

Known Issues and Limitations

- **On initial deployment DPF CRs maybe stuck in initial state (Pending, Initializing, etc.) and not progressing**
  - In case DPF CRs were created before DPF components are running they maybe be stuck in their initial state. DPF CRs need to be created after the DPF components have been deployed. In case CRs were created before they may remain in an initial state.
  - Internal Ref #4241297
  - **Workaround:** Delete any CRs that were created before the System components have been deployed and recreate them.
- **DPUService stuck in its Deleting phase**
  - DPUService can be stuck in its Deleting phase when a pod on the DPU created as part of the DPUService can not be deleted.
  - Internal Ref #4213229
  - **Workaround:** Reprovision the DPU where the DPUService is deployed.
- **Incompatible DPUFlavor can cause DPU to get into an unstable state**

    - Using an incompatible DPUFlavor can cause the DPU Device to get into an error state which requires manual intervention. For example allocating 14GB of hugepages in a DPU of 16GB memory.
    - Internal Ref #4200717
    - **Workaround:** Manually provision DPU or follow DOCA troubleshooting documentation to return DPU to operational state https://docs.nvidia.com/networking/display/bfswtroubleshooting.
- **Traffic loss after reconfiguration of DPUServices with chain between**
    - Reconfiguration of DPUServices with chain between them may cause traffic loss due to outdated service chains.
    - Internal Ref #4178445
    - **Workaround:** Recreated SFC object between services.
- **Stale ports after DPU reboot**
    - When rebooting DPU, the old DPU service ports won't get deleted from DPU's OVS and would be stale
    - Internal Ref #4174183
    - **Workaround:** No workaround, known issue, shouldn't affect performance.
- **BFB filename must be unique**
    - If BFB CR#1 bfb.spec.filename is the same as a BFB CR#2 bfb.spec.filename but references a different URL (actual bfb file to download) then BFB CR#1 would reference the wrong bfb.
    - Internal Ref #4143309
    - **Workaround:** Use unique bfb.spec.filename when creating new bfb CRs.
- **DPU Cluster control-plane connectivity is lost when physical port P0 is down on the worker node**
    - Link down of p0 port on the DPU will result in DPU control plane connectivity loss of DPU components.

    - Internal Ref #3751863
    - **Workaround:** Make sure P0 link is up on the DPU, if down either restart DPU or refer to DOCA troubleshooting https://docs.nvidia.com/networking/display/bfswtroubleshooting.

- **DPU Provisioning operations wouldn't be retried**
    - DPU Provisioning operations wouldn't be retried, this can lead to DPU object in an error phase because of small environment glitch which would have worked if retried.
    - Internal Ref #4202272
    - **Workaround:** Delete the DPU object in an error phase which will cause it to get recreated and operation to begin from scratch.

- **Cluster MTU value cannot be dynamically changed**
    - It is possible to deploy DPF cluster with a custom MTU value, however once deployed, it is not possible to modify the MTU value which is applied on multiple distributed components.
    - Internal Ref #3917006
    - **Workaround:** Uninstall DPF and re-install from scratch using the new MTU value.

- **nvidia-k8s-ipam and servicechainset-controller DPF system DPUServices are in "Pending" phase**
    - As long as there are no provisioned DPUs in the system, the nvidia-k8s-ipam and servicechainset-controller will appear as not ready / pending when querying dpuservices.

      This has no impact on performance or functionality since DPF system components are only relevant when there are DPUs to provision services on.
    - Internal Ref #4241324
    - **Workaround:** No workaround, known issue

- **[OVN-Kubernetes DPUService] Nodes marked as NotReady**

- When installing OVN-Kubernetes as a CNI on a node running containerd version 1.7.0 and above the Node never becomes ready.
- Internal Ref #4178221
- **Workaround:**

  Option 1: Use containerd version below v1.7.0 when using OVN-Kubernetes as a primary CNI.

  Option 2: Manually restart containerd on the host.

- **[OVN-Kubernetes DPUService] control plane node is not functional after reboot or network restart**
  - During OVN-Kubernetes CNI installation on the control plane nodes, the management interface is moved with its IP into a newly created OVS bridge. Since this network configuration is not persistent it will be lost during node or network restart.
  - Internal Ref #4241306
  - **Workaround:**

    1. Pre-define the OVS bridge on each control plane node with the OOB port MAC and IP address and ensure it gets a persistent IP

```
yaml #Ubuntu example for netplan persistent network
configuration: network: ethernets: oob: match: # the mac
address of the oob macaddress: xx:xx:xx:xx:xx:xx set-name:
oob bridges: br0: addresses: x.x.x.x/x interfaces: [oob] #
the mac address of the oob macaddress: xx:xx:xx:xx:xx:xx
openvswitch: {} version: 2
```

    1. Set OVS bridge "bridge-uplink" in OVS metadata.

```
bash ovs-vsctl br-set-external-id br0 bridge-id br0 -- br-
set-external-id br0 bridge-uplink oob
```

- **[OVN-Kubernetes DPUService] Only a single OVN-Kubernetes DPU service version can be deployed across the cluster**
  - OVN-Kubernetes service does not fully support customization using Helm parameters, meaning we only support a single OVN-Kubernetes DPU service

across the entire cluster.

  - Internal Ref #4209524

  - **Workaround:** No workaround, known limitation.

- **[OVN-Kubernetes DPUService] Lost traffic from workloads to control plane components or K8S services after dpu reboot, port flapping, ovs restart or manual network configuration**

  - Connectivity issues between workload pods to control plane components or K8S services may occur after the following events: DPU reboot without host reboot, high speed port flapping (link down/up), ovs restart, DPU network configuration change (for example using "netplan apply" command on DPU).

    The issues are caused by network configuration that was applied by ovn CNI on DPUs and won't get reapplied automatically.

    When rebooting DPU without the host, or high speed port link is going down/up, or manually changing dpu network (for example with netplan apply), network configuration which was applied by the dpu CNI components may be lost and won’t reapply automatically.

  - Internal Ref #4202272

  - **Workaround:** Recreate the OVN-Kubernetes node pod on the host to reapply the configuration.

- **[OVN-Kubernetes DPUService] host network configuration may result in lost traffic from host workloads (on overlay)**

  - When changing host network (for example with netplan apply) custom network configuration which is done by the host CNI components may be lost and won’t reapply automatically.

  - Internal Ref #4188044

  - **Workaround:** Recreate the OVN-Kubernetes node pod on the host to reapply the configuration.

- **[OVN-Kubernetes DPUService] No network connectivity for SR-IOV accelerated workload pods after DPU reboot**

    - SR-IOV accelerated workload pod is losing its VF interface upon DPU reboot. VF is available on the host however not injected back into the pod.
    - Internal Ref #4236521
    - **Workaround:** Recreate the SR-IOV accelerated workload pods.
- **[HBN DPUService] HBN DPUService cannot dynamically reload configurations**
    - When updating HBN configuration through a configmap, the running HBN container won't reload it, and need to get restarted with the new configuration.
    - Internal Ref #4290426
    - **Workaround:** Recreate HBN DPU service after changing configuration.
- **[HBN DPUService] Invalid HBN configuration is not reflected to user in case it is syntactically valid**
    - If the HBN YAML configuration is valid but contains values that are illegal from an NVUE perspective then the HBN service will start with the last known valid configuration and it won't be reflected to the end user.
    - **Workaround:** No workaround, known issue.
- **[DTS DPUService] DTS appears as OutOfSync**
    - When creating a DPUDeployment for DTS DPU service, the DPUService object can be marked as OutOfSync although the pods are running on the DPUs.
    - Internal Ref #4182929
    - **Workaround:** No workaround, known issue.
- **[OVN-Kubernetes DPUService] SR-IOV test Pod cannot reach Kubernetes API Service**
    - When running a SR-IOV test Pod, the pod cannot reach a Kubernetes API Service. The issue is that the related conntrack entries miss the un-nat sometimes.
    - Internal Ref #4313629

- **Workaround:** Unblock is to run the following on the DPUs:
  ```
  ovs-appctl revalidator/purge ovs-appctl dpctl/flush-conntrack
  ```

DOCA Platform Framework v25.4.0

This is the GA release of the DOCA Platform Framework (DPF). It includes bug fixes and improvements to enhance the provisioning and orchestration of NVIDIA BlueField DPUs in Kubernetes environments.

Features

- (Beta) Using Redfish to provision DPUs connected to the management network with the Out-Of-Band interface
- (Beta) Provisioning of DPUs in servers that are not Kubernetes workers
- (Beta) Graceful upgrade of BFB and DPUServices via the DPUDeployment Custom Resource
- Ability to attach DPUServices to a user defined network via the DPUServiceNAD Custom Resource

Fixed Issues from Previous Release

> **Note**
>
> Detailed information about the fixed issues can be found in the release notes for v25.1.1.

- DPUService stuck in its Deleting phase
- Traffic loss after reconfiguration of DPUServices with chain between
- DPU Provisioning operations wouldn't be retried

- [OVN-Kubernetes DPUService] Only a single OVN-Kubernetes DPU service version can be deployed across the cluster
- [HBN DPUService] HBN DPUService cannot dynamically reload configurations
- [DTS DPUService] DTS appears as OutOfSync

Supported DPU Services:

- OVN-Kubernetes
- DOCA Host-Based Networking (HBN)
- DOCA Telemetry Service (DTS)
- DOCA BlueMan
- DOCA SNAP
    - Supports NVMe-oF protocol with TCP and RDMA transport
    - Supports Kubernetes PersistentVolume with volumeMode: Block
- DOCA Firefly

Dependencies

Hardware and Software Requirements

> **Note**
>
> Refer to Prerequisites for detailed requirements. In addition, notice the additional requirements for zero trust

- DPU Hardware: NVIDIA BlueField-3 DPUs
- Minimal DOCA BFB Image: DOCA v2.5 or higher (must be pre-installed on DPUs to support DPF provisioning)

- Minimal BMC DPU Firmware Version: >= 24.10
- Provisioned DOCA BFB Image: `bf-bundle-3.0.0-135`

Known Issues and Limitations

- **On initial deployment DPF CRs maybe stuck in initial state (Pending, Initializing, etc.) and not progressing**
    - In case DPF CRs were created before DPF components are running they maybe be stuck in their initial state. DPF CRs need to be created after the DPF components have been deployed. In case CRs were created before they may remain in an initial state.
    - Internal Ref #4241297
    - **Workaround:** Delete any CRs that were created before the System components have been deployed and recreate them.
- **Incompatible DPUFlavor can cause DPU to get into an unstable state**
    - Using an incompatible DPUFlavor can cause the DPU Device to get into an error state which requires manual intervention. For example allocating 14GB of hugepages in a DPU of 16GB memory.
    - Internal Ref #4200717
    - **Workaround:** Manually provision DPU or follow DOCA troubleshooting documentation to return DPU to operational state https://docs.nvidia.com/networking/display/bfswtroubleshooting.
- **Stale ports after DPU reboot**
    - When rebooting DPU, the old DPU service ports won't get deleted from DPU's OVS and would be stale
    - Internal Ref #4174183
    - **Workaround:** No workaround, known issue, shouldn't affect performance.
- **BFB filename must be unique**

- If BFB CR#1 bfb.spec.filename is the same as a BFB CR#2 bfb.spec.filename but references a different URL (actual bfb file to download) then BFB CR#1 would reference the wrong bfb.
    - Internal Ref #4143309
    - **Workaround:** Use unique bfb.spec.filename when creating new bfb CRs.

- **DPU Cluster control-plane connectivity is lost when physical port P0 is down on the worker node**
    - Link down of p0 port on the DPU will result in DPU control plane connectivity loss of DPU components.
    - Internal Ref #3751863
    - **Workaround:** Make sure P0 link is up on the DPU, if down either restart DPU or refer to DOCA troubleshooting https://docs.nvidia.com/networking/display/bfswtroubleshooting.

- **Cluster MTU value cannot be dynamically changed**
    - It is possible to deploy DPF cluster with a custom MTU value, however once deployed, it is not possible to modify the MTU value which is applied on multiple distributed components.
    - Internal Ref #3917006
    - **Workaround:** Uninstall DPF and re-install from scratch using the new MTU value.

- **nvidia-k8s-ipam and servicechainset-controller DPF system DPUServices are in "Pending" phase**
    - As long as there are no provisioned DPUs in the system, the nvidia-k8s-ipam and servicechainset-controller will appear as not ready / pending when querying dpuservices. This has no impact on performance or functionality since DPF system components are only relevant when there are DPUs to provision services on.
    - Internal Ref #4241324
    - **Workaround:** No workaround, known issue

- **System doesn't recover after DPU Reset**
  - When the user triggers a reset of a DPU in any way other than using DPF APIs (e.g. recreation of a DPU CR), the system may not recover.
  - Internal Ref #4424305, #4424235, #4188044
  - **Workaround:** Power cycle the host. Note that this operation is dangerous and there might be file system corruption which will require triggering the reprovisioning of the DPU.
- **DPUDeployment disruptive upgrade handling is not always graceful**
  - When a DPUDeployment triggers a disruptive upgrade (default behavior), there is a case that the upgrade will not trigger the related nodeEffect. This can happen when there were changes only to the DPUService related fields.
  - Internal Ref #4252072
  - **Workaround:** No workaround, known issue.
- **DPUDeployment spec.dpuSets.nodeEffect changes require recreation of the CR**
  - In case the user wants to change the spec.dpuSets.nodeEffect field on an already applied DPUDeployment, the reconciliation of the resource won't succeed.
  - Internal Ref #4410218
  - **Workaround:** Recreate the DPUDeployment which will trigger workload disruption and reprovisioning of DPUs.
- **Leftover CRs if worker is removed from the cluster permanently**
  - When a worker was added to a cluster, optionally had DPU provisioned and later was removed from the host cluster permanently, there may be leftover DPF related CRs in both the host cluster and the DPU cluster.
  - Internal Ref #4403130, #4426516
  - **Workaround:** No workaround, known issue.
- **DPUSet and BFB removal leaves DPU CRs in OS Installing Phase indefinitely**

- When a DPUSet that owns DPU CRs that are in OS Installing phase is deleted together with the BFB that the DPUSet is referencing, the DPUs are stuck in OS Installing Phase indefinitely and can't be removed. This can also be a race condition when deleting the DPFOperatorConfig CR and there are DPUs in OS Installing Phase.
- Internal Ref #4426349
- **Workaround:** No workaround, known issue.

- **DPUDeployment with duplicate nameSuffix in dpuSets causes DPF to ignore the first entry and triggers endless reprovisioning**
  - When defining multiple dpuSets in a single DPUDeployment, if two entries use the same nameSuffix but have different nodeSelectors, the system only respects the last defined dpuSet. The earlier one is ignored without validation or error. This results in DPF attempting to provision DPUs that don't match the second dpuSet's nodeSelector, causing them to get stuck in an infinite reprovisioning loop.
  - Internal Ref #4413138
  - **Workaround:** Ensure the nameSuffix is unique across all the entries in the dpuSets list.

- **Deleting ArgoCD pods on a cluster with workers causes pod to be created on worker nodes**
  - When deleting the ArgoCD pods, the new ones might end up on a worker and in case the OVN Kubernetes VF injector is installed, this may lead to a deadlock where DPUServices are not reconciled.
  - Internal Ref #4401382
  - **Workaround:**

    1. Cordon all the worker nodes

    ```
    bash kubectl cordon -l node-role.kubernetes.io/control-plane!=
    ```

    1. (Optional) Delete the MutatingWebhookConfiguration related to the OVN Kubernetes VF Injector

```
bash kubectl delete mutatingwebhookconfiguration ovn-kubernetes-resource-injector
```

1. Delete the ArgoCD pods

```
bash kubectl delete pod -n dpf-operator-system -l app.kubernetes.io/part-of=argocd
```

1. Uncordon all the worker nodes

```
bash kubectl uncordon -l node-role.kubernetes.io/control-plane!=
```

1. (Optional) Reinstall the OVN Kubernetes VF Injector as instructed in the OVN related guides

- **DPUDeployment or DPUSet created in namespace different from the dpf-operator-system namespace do not trigger DPU provisioning**
  - When creating a DPUDeployment or a DPUSet in a namespace other than dpf-operator-system, there are no DPU CRs created due to the DPUNode CRs residing in the dpf-operator-system namespace.
  - Internal Ref #4427091
  - **Workaround:** Create the DPUDeployment or DPUSet in the dpf-operator-system namespace.
- **Service Function Chain(SFC) is configured only when all chain components are installed and correct**
  - When creating a service chain with more than 2 endpoints, introducing bad configuration in one of the DPUServiceInterfaces involved or making one of the DPUServices unavailable, prevents the relevant controller from properly configuring the chain. This means that a bad configuration somewhere in this chain, breaks the entire chain, affecting all services using it for traffic.
  - Internal Ref #4367518
  - **Workaround:** No workaround, known issue.
- **DPUService stops reconciling when DPUCluster is unavailable for long time**

- When the DPUCluster is unavailable for long time (more than 5 mins), changes to DPUServices (also generated ones via DPUDeployment or DPFOperatorConfig) that have happened during that time might not be reflected to the DPUCluster.
- Internal Ref #4359857
- **Workaround:** Recreate DPUServices that are stuck.

- **dmsinit.sh fails to fetch newly created certificate secret on first run for non Kubernetes workers**
  - When provisioning DPUs on servers that are not Kubernetes workers, the dmsinit.sh script specified in the documentation may fail in the first run due to cert-manager updating the Certificate spec.secretName before the secret is created in the API server.
  - Internal Ref #4445638
  - **Workaround:** Rerun the dmsinit.sh script.
- **Instability with DPU provisioning when NFS service is not available**
  - When the NFS service is unavailable, the following behaviors may occur: 1. Deleting a Ready DPU CR does not succeed until the NFS service is restored. No error message is shown to the user. 2. DPU CR in the "OS Installing" phase hangs without any error, and DPU provisioning does not proceed.
  - Internal Ref #4123204
  - **Workaround:** Ensure the NFS service is available. For issue #2, delete the affected DPU CR and wait for a new CR to be created. Note that deleting the DPU CR in this phase may take up to 30 minutes.
- **DPUDeployment should specify supported spec.dpuSets.nodeEffect when deploying on DPUs that are part of servers that are not Kubernetes workers**
  - When provisioning DPUs on servers that are not Kubernetes workers, the DPUDeployment must have spec.dpuSets.nodeEffect specified and the value should be a value supported in this configuration. See the Node Effects guide for more information.
  - Internal Ref #4445649

- **Workaround:** Specify correct node effect by recreating the DPUDeployment.

- **[OVN-Kubernetes DPUService] Nodes marked as NotReady**
  - When installing OVN-Kubernetes as a CNI on a node running containerd version 1.7.0 and above the Node never becomes ready.
  - Internal Ref #4178221
  - **Workaround:** Option 1: Use containerd version below v1.7.0 when using OVN-Kubernetes as a primary CNI. Option 2: Manually restart containerd on the host.

- **[OVN-Kubernetes DPUService] control plane node is not functional after reboot or network restart**
  - During OVN-Kubernetes CNI installation on the control plane nodes, the management interface is moved with its IP into a newly created OVS bridge. Since this network configuration is not persistent it will be lost during node or network restart.
  - Internal Ref #4241306
  - **Workaround:**

    1. Pre-define the OVS bridge on each control plane node with the OOB port MAC and IP address and ensure it gets a persistent IP

```
yaml #Ubuntu example for netplan persistent network
configuration: network: ethernets: oob: match: # the mac
address of the oob macaddress: xx:xx:xx:xx:xx:xx set-name:
oob bridges: br0: addresses: x.x.x.x/x interfaces: [oob] #
the mac address of the oob macaddress: xx:xx:xx:xx:xx:xx
openvswitch: {} version: 2
```

    1. Set OVS bridge "bridge-uplink" in OVS metadata.

```
bash ovs-vsctl br-set-external-id br0 bridge-id br0 -- br-
set-external-id br0 bridge-uplink oob
```

- **[OVN-Kubernetes DPUService] Lost traffic from workloads to control plane components or K8S services after dpu reboot, port flapping, ovs restart or manual network configuration**

    - Connectivity issues between workload pods to control plane components or K8S services may occur after the following events: DPU reboot without host reboot, high speed port flapping (link down/up), ovs restart, DPU network configuration change (for example using "netplan apply" command on DPU). The issues are caused by network configuration that was applied by ovn CNI on DPUs and won't get reapplied automatically. When rebooting DPU without the host, or high speed port link is going down/up, or manually changing dpu network (for example with netplan apply), network configuration which was applied by the dpu CNI components may be lost and won’t reapply automatically.

    - Internal Ref #4424305, #4188044, #4424235

    - **Workaround:** Recreate the OVN-Kubernetes node pod on the host to reapply the configuration.

- **[OVN-Kubernetes DPUService] host network configuration may result in lost traffic from host workloads (on overlay)**

    - When changing host network (for example with netplan apply) custom network configuration which is done by the host CNI components may be lost and won’t reapply automatically.

    - Internal Ref #4188044

    - **Workaround:** Recreate the OVN-Kubernetes node pod on the host to reapply the configuration.

- **[OVN-Kubernetes DPUService] InternalTrafficPolicy and ExternalTrafficPolicy Service options are not handled correctly**

    - When external traffic reaches a NodePort Kubernetes Service on the host cluster via the OOB Network of the host, the user won't see the relevant policy option working as expected. The same applies for traffic originating from a Pod on the overlay network hitting the same type of service.

    - Internal Ref #4320953

    - **Workaround:** No workaround, known issue.

- **[OVN-Kubernetes DPUService] No network connectivity for SR-IOV accelerated workload pods after DPU reboot**

- SR-IOV accelerated workload pod is losing its VF interface upon DPU reboot. VF is available on the host however not injected back into the pod.
- Internal Ref #4236521
- **Workaround:** Recreate the SR-IOV accelerated workload pods.

- **[OVN-Kubernetes DPUService] SR-IOV test Pod cannot reach Kubernetes API Service**
  - When running a SR-IOV test Pod, the pod cannot reach a Kubernetes API Service. The issue is that the related conntrack entries miss the un-nat sometimes.
  - Internal Ref #4313629
  - **Workaround:** Unblock is to run the following on the DPUs:
    ```
    ovs-appctl revalidator/purge ovs-appctl dpctl/flush-conntrack
    ```

- **[HBN DPUService] Invalid HBN configuration is not reflected to user in case it is syntactically valid**
  - If the HBN YAML configuration is valid but contains values that are illegal from an NVUE perspective then the HBN service will start with the last known valid configuration and it won't be reflected to the end user.
  - Internal Ref #4172029
  - **Workaround:** No workaround, known issue.

- **[HBN + OVN-Kubernetes DPUServices] HBN service restarts on DPU causes worker to lose traffic**
  - If the HBN pod on the DPU will reset then the workloads on the host (any traffic on the OVN overlay) will not receive traffic.
  - Internal Ref #4220185, #4223176
  - **Workaround:** Wait 15 minutes for the system to recover or restart the OVN-Kubernetes Pod on that particular DPU.

- **[Firefly DPUService] Deletion of the Firefly DPUService leaves stale flows in OVS**

    - When the Firefly DPUService is deleted after successful deployment or the labels of the serviceDaemonSet are modified, flows are not cleaned up from OVS.
    - Internal Ref #4382535
    - **Workaround:** Although these flows are very unlikely to cause an issue, reprovisioning the DPU or power cycling the host will bring the OVS in good state. Note that power cycling the host is dangerous and there might be file system corruption which will require triggering the reprovisioning of the DPU.
- **[OVN-Kubernetes DPUService] Kubernetes Services sporadically can not be reached after system reboots**
    - After a node reboots, there is a small chance workload pods cannot reach the Kubernetes Service addresses (i.e. Kubernetes API server). This is because the OVS ARP cache of tunnel endpoints is not correctly populated on the DPU.
    - Internal Ref #4448292, #4448355
    - **Workaround:** Generate ICMP traffic from the affected node through the Kubernetes pod network. For example: If the cluster was created with the following pod network POD_CIDR=10.233.64.0/18, each node will get a /24 slice. From the affected node trigger ICMP traffic (i.e. using the ping utility) to the ovn-k8s-mp0 interface on one of the control plane nodes (i.e. ping 10.233.64.2)

DOCA Platform Framework v25.7.0

This is the GA release of the DOCA Platform Framework (DPF). It includes bug fixes and improvements to enhance the provisioning and orchestration of NVIDIA BlueField DPUs in Kubernetes environments.

Features

- **New DOCA services are now deployable as DPUServices**:
    - DOCA Argus service
- **DPU Zero Trust is now Beta**:
    - **VPC Service support**: Enables centralized VPC network creation and configuration on BlueField DPUs with a declarative API

    - **SNAP based storage volume management**: Ability to expose storage resources to the host as emulated NVMe devices or as a VirtioFS filesystem
    - **Auto-discovery of DPUs**: DPU devices can be automatic discovered in the fabric removing the need to manually create an object for each DPUDevice with its details
    - **Auto update of old BMC Versions**: DPF will automatically identify old BMC versions and update them to the minimal version, revmoving the need to manually update the BMC from an old version
- **Improved DPU Management and Kubernetes Integration**:
    - **DPUDeployment service dependency management**: DPF now supports service deployment dependencies, allowing dependent service Helm values to be templated with items from dependencies
    - **Workload prevention on Kubernetes host**: Improved host worker node and DPUServices life cycle synchronization. Kubernetes host worker node would be marked as NotReady until all the required services on DPU are running

Fixed Issues from Previous Release

> **Note**
>
> Detailed information about the fixed issues can be found in the release notes for v25.4.0.

- Incompatible DPUFlavor can cause DPU to get into an unstable state
- BFB filename must be unique
- DPUDeployment spec.dpuSets.nodeEffect changes require recreation of the CR
- DPUDeployment with duplicate nameSuffix in dpuSets causes DPF to ignore the first entry and triggers endless reprovisioning
- Deleting ArgoCD pods on a cluster with workers causes pod to be created on worker nodes

- Service Function Chain(SFC) is configured only when all chain components are installed and correct
- dmsinit.sh fails to fetch newly created certificate secret on first run for non Kubernetes workers
- Instability with DPU provisioning when NFS service is not available
- DPUDeployment should specify supported spec.dpuSets.nodeEffect when deploying on DPUs that are part of servers that are not Kubernetes workers
- [HBN + OVN-Kubernetes DPUServices] HBN service restarts on DPU causes worker to lose traffic

Supported DPU Services:

- OVN-Kubernetes
- DOCA Host-Based Networking (HBN)
- DOCA Telemetry Service (DTS)
- DOCA BlueMan
- DOCA SNAP
    - Supports NVMe and VirtioFS emulation for DPF deployments in non-trusted mode
    - Supports NVMe emulation for DPF deployments in trusted mode
    - Supports Kubernetes PersistentVolume with volumeMode: Block for DPF deployments in trusted mode
- DOCA Firefly
- DOCA Argus
- OVN VPC

Dependencies

Hardware and Software Requirements

> **Note**
>
> For Host Trusted mode: Refer to Host Trusted Prerequisites for detailed requirements.
>
> For Zero Trust mode: Refer to Zero Turst Prerequisites for detailed requirements. In addition, notice the additional requirements for deploying via Redfish.

- DPU Hardware: NVIDIA BlueField-3 DPUs
- Minimal DOCA BFB Image: DOCA v2.5 or higher (must be pre-installed on DPUs to support DPF provisioning)
- Supported DOCA BFB Image: `bf-bundle-3.1.0`

Installation & Upgrade Notes

Installation Changes

Starting with DPF v25.7, **all Helm dependencies have been removed** from the DPF chart. This means that all dependencies must be installed manually before installing the DPF chart itself.

For detailed installation instructions and configuration values, refer to the Helm Prerequisites documentation.

Upgrade Support

DPF v25.7 does not support upgrades from previous versions. Users must completely uninstall/purge all DPF components and install on a fresh system.

> **Note**
>
> **Future Upgrade Support**: Starting with DPF v25.10, we plan to introduce proper upgrade support that will allow seamless upgrades between compatible versions.

Known Issues and Limitations

- **On initial deployment DPF CRs maybe stuck in initial state (Pending, Initializing, etc.) and not progressing**
  - In case DPF CRs were created before DPF components are running they may be stuck in their initial state. DPF CRs need to be created after the DPF components have been deployed. In case CRs were created before they may remain in an initial state.
  - Internal Ref #4241297
  - **Workaround:** Delete any CRs that were created before the System components have been deployed and recreate them.
- **Stale ports after DPU reboot**
  - When rebooting DPU, the old DPU service ports won't get deleted from DPU's OVS and would be stale
  - Internal Ref #4174183
  - **Workaround:** No workaround, known issue, shouldn't affect performance.
- **DPU Cluster control-plane connectivity is lost when physical port P0 is down on the worker node**
  - Link down of p0 port on the DPU will result in DPU control plane connectivity loss of DPU components.
  - Internal Ref #3751863

    - **Workaround:** Make sure P0 link is up on the DPU, if down either restart DPU or refer to DOCA troubleshooting https://docs.nvidia.com/networking/display/bfswtroubleshooting. **Note:** This issue is relevant for Host Trusted deployments only.

- **Cluster MTU value cannot be dynamically changed**
    - It is possible to deploy DPF cluster with a custom MTU value, however once deployed, it is not possible to modify the MTU value which is applied on multiple distributed components.
    - Internal Ref #3917006
    - **Workaround:** Uninstall DPF and re-install from scratch using the new MTU value.

- **nvidia-k8s-ipam and servicechainset-controller DPF system DPUServices are in "Pending" phase**
    - As long as there are no provisioned DPUs in the system, the nvidia-k8s-ipam and servicechainset-controller will appear as not ready / pending when querying dpuservices. This has no impact on performance or functionality since DPF system components are only relevant when there are DPUs to provision services on.
    - Internal Ref #4241324
    - **Workaround:** No workaround, known issue

- **System doesn't recover after DPU Reset**
    - When the user triggers a reset of a DPU in any way other than using DPF APIs (e.g. recreation of a DPU CR), the system may not recover.
    - Internal Ref #4424305, #4424235, #4188044
    - **Workaround:** Power cycle the host. Note that this operation is dangerous and there might be file system corruption which will require triggering the reprovisioning of the DPU. **Note:** This issue is relevant for Host Trusted deployments where OVN-Kubernetes is used as a primary CNI.

- **DPUDeployment disruptive upgrade handling is not always graceful**

- When a DPUDeployment triggers a disruptive upgrade (default behavior), there is a case that the upgrade will not trigger the related nodeEffect. This can happen when there were changes only to the DPUService related fields.
- Internal Ref #4252072
- **Workaround:** No workaround, known issue.

- **Leftover CRs if worker is removed from the cluster permanently**
  - When a worker was added to a cluster, optionally had DPU provisioned and later was removed from the host cluster permanently, there may be leftover DPF related CRs in both the host cluster and the DPU cluster.
  - Internal Ref #4403130, #4426516
  - **Workaround:** No workaround, known issue.

- **DPUSet and BFB removal leaves DPU CRs in OS Installing Phase indefinitely**
  - When a DPUSet that owns DPU CRs that are in OS Installing phase is deleted together with the BFB that the DPUSet is referencing, the DPUs are stuck in OS Installing Phase indefinitely and can't be removed. This can also be a race condition when deleting the DPFOperatorConfig CR and there are DPUs in OS Installing Phase.
  - Internal Ref #4426349
  - **Workaround:** No workaround, known issue.

- **DPUDeployment or DPUSet created in namespace different from the dpf-operator-system namespace do not trigger DPU provisioning**
  - When creating a DPUDeployment or a DPUSet in a namespace other than dpf-operator-system, there are no DPU CRs created due to the DPUNode CRs residing in the dpf-operator-system namespace.
  - Internal Ref #4427091
  - **Workaround:** Create the DPUDeployment or DPUSet in the dpf-operator-system namespace.

- **DPUService stops reconciling when DPUCluster is unavailable for long time**

    - When the DPUCluster is unavailable for long time (more than 5 mins), changes to DPUServices (also generated ones via DPUDeployment or DPFOperatorConfig) that have happened during that time might not be reflected to the DPUCluster.
    - Internal Ref #4359857
    - **Workaround:** Recreate DPUServices that are stuck.
- **DPUFlavor nvconfig limitations**
    - DPUFlavor doesn't support assigning nvconfig parameters to only a specific device(s). Currently, it assigns the same parameters across all devices (i.e., '*')
    - Internal Ref #4583584
- **DPUFlavor nvconfig.hostPowerCycleRequired is not used**
    - The `nvconfig.hostPowerCycleRequired` parameter is defined in the DPUFlavor spec but is not used in the code.
    - Internal Ref #4583693
- **[OVN-Kubernetes DPUService] Nodes marked as NotReady**
    - When installing OVN-Kubernetes as a CNI on a node running containerd version 1.7.0 and above the Node never becomes ready.
    - Internal Ref #4178221
    - **Workaround:** Option 1: Use containerd version below v1.7.0 when using OVN-Kubernetes as a primary CNI. Option 2: Manually restart containerd on the host.
- **[OVN-Kubernetes DPUService] Lost traffic from workloads to control plane components or Kubernetes services after dpu reboot, port flapping, ovs restart or manual network configuration**
    - Connectivity issues between workload pods to control plane components or Kubernetes services may occur after the following events: DPU reboot without host reboot, high speed port flapping (link down/up), ovs restart, DPU network configuration change (for example using "netplan apply" command on DPU). The issues are caused by network configuration that was applied by ovn CNI on DPUs and won't get reapplied automatically.

    - Internal Ref #4424305, #4188044, #4424235
    - **Workaround:** Recreate the OVN-Kubernetes node pod on the host to reapply the configuration.
- **[OVN-Kubernetes DPUService] host network configuration may result in lost traffic from host workloads (on overlay)**
    - When changing host network (for example with netplan apply) custom network configuration which is done by the host CNI components may be lost and won't reapply automatically.
    - Internal Ref #4188044
    - **Workaround:** Recreate the OVN-Kubernetes node pod on the host to reapply the configuration.
- **[OVN-Kubernetes DPUService] InternalTrafficPolicy and ExternalTrafficPolicy Service options are not handled correctly**
    - When external traffic reaches a NodePort Kubernetes Service on the host cluster via the OOB Network of the host, the user won't see the relevant policy option working as expected. The same applies for traffic originating from a Pod on the overlay network hitting the same type of service.
    - Internal Ref #4320953
    - **Workaround:** No workaround, known issue.
- **[OVN-Kubernetes DPUService] No network connectivity for SR-IOV accelerated workload pods after DPU reboot**
    - SR-IOV accelerated workload pod is losing its VF interface upon DPU reboot. VF is available on the host however not injected back into the pod.
    - Internal Ref #4236521
    - **Workaround:** Recreate the SR-IOV accelerated workload pods.
- **[HBN DPUService] Invalid HBN configuration is not reflected to user in case it is syntactically valid**
    - If the HBN YAML configuration is valid but contains values that are illegal from an NVUE perspective then the HBN service will start with the last known valid

configuration and it won't be reflected to the end user.

  - Internal Ref #4172029
  - **Workaround:** No workaround, known issue.

- **[Firefly DPUService] Deletion of the Firefly DPUService leaves stale flows in OVS**
  - When the Firefly DPUService is deleted after successful deployment or the labels of the serviceDaemonSet are modified, flows are not cleaned up from OVS.
  - Internal Ref #4382535
  - **Workaround:** Although these flows are very unlikely to cause an issue, reprovisioning the DPU or power cycling the host will bring the OVS in good state. Note that power cycling the host is dangerous and there might be file system corruption which will require triggering the reprovisioning of the DPU.

- **[VPC OVN DPUService] E/W Traffic Not Passing within Same VPC after DPU Reprovisioning:**
  - After DPU reprovisioning, there is a chance to experience issues where east-west (E/W) traffic is not passing between nodes under the same VPC. This is because OVN-controller failed to register chassis in ovn-sb DB. This may also happen when VPC is completely removed and re-applied
  - Internal Ref: #4553212
  - **Workaround**:
    - Verify the ServiceInterfaces are ready on all nodes.
    - Confirm that these ServiceInterfaces are associated with the same DPUVirtualNetwork and are part of the expected DPUVPC.
    - Locate the ovn-central pod running in the management cluster, example:
      ```
      sh kubectl get pods -n dpf-operator-system | grep ovn-central
      ```
    - run the following command in ovn-central pod to remove leftover chassis records:

```
sh kubectl exec -it -n dpf-operator-system <in-cluster-
ovn-central> -- \ sh -c 'for chassis in $(ovn-sbctl
list Chassis | awk "/_uuid/ {print \$3}"); do ovn-sbctl
destroy Chassis $chassis; done'
```

- **DPUServiceInterface doesn't wait for ServiceInterface to be deleted**
    - When a DPUServiceInterface, DPUVPC and DPUVirtualNetwork are deleted, the DPUServiceInterface reconciler doesn't wait for the underlying ServiceInterfaces to be deleted before proceeding. This causes DPUVirtualnetwork and DPUVPC to be removed and consequently the vpc serviceinterface controller cannot properly unplug a serviceinterface from vpc since the objects are already deleted (it waits for them to be present).
    - Internal Ref #4578503

DOCA Platform Framework v25.7.1

This is a patch release on top of the 25.7.0 release of the DOCA Platform Framework (DPF). It includes bug fixes and improvements to enhance the provisioning and orchestration of NVIDIA BlueField DPUs in Kubernetes environments.

The release notes for 25.7.0 still apply for this release and information presented in this documentation is an amendment to the previous release notes.

This release contains artifacts only for the core DPF components and doesn't include VPC, Storage and OVN Kubernetes related artifacts.

Features

- **Support provisioning a single DPU in nodes that have more than one DPU**: This feature enables customers to select which DPU DPF should provision and run services on in systems that have more than 1 DPU. It's still not supported to provision more than 1 DPU per node.

Fixed Issues from Previous Release

- **[HBN DPUService] Pod stuck in CrashLoopBackOff due to hbn-sidecar failing**
    - Occasionally, on creation of the HBN DPUService Pod, it could happen that the hbn-sidecar container would fail to run `ovs-ofctl add-flow` with

arguments that contain `in_port=-1`. In this release, we introduced a patch that helps reduce the probability of this happening.

- Internal Ref #4618866

DOCA Platform Framework v25.10.0

This is the GA release of the DOCA Platform Framework (DPF). It includes bug fixes and improvements to enhance the provisioning and orchestration of NVIDIA BlueField DPUs in Kubernetes environments.

Features

- **Multi-DPU Support**:
  - Enhancements to support multiple DPUs throughout DPF components
  - Support provisioning multiple DPUs on the same host with node effect and reboot synchronization
- **Networking**:
  - RDMA CNI support for DPU workloads
- **DPF Operator**:
  - Support Resource configuration for DPF components
- **Support for DPUDeployment disruptive upgrades**:
  - Support for triggering node effect on update of disruptive standard or in-cluster DPUServices and DPUServiceChains
  - Support for removing the node effect only after all the DPUServices have Pods that are Ready
  - Support for removing the node effect only after all the DPUServiceChains have ServiceChains that are Ready
- **Provisioning Enhancements**:
  - New Host Agent component to handle host operations

- **Upgrade Support**:
  - DPF now supports upgrades from previous GA releases
- **DPF Storage Enhancements**:
  - Support VirtioFS for Trusted and Zero Trust cases
  - Support for static PF PCI emulated functions
  - Support for multiple DPUs on same host
- **OVN VPC DPUService**:
  - (tech preview) OVN HA support
  - Zero downtime upgrade guide
- **dpfctl**:
  - Enhancements to reflect the status of additional DPF objects such as storage, dpunode, dpudevice

Fixed Issues from Previous Release

> **Note**
>
> Detailed information about the fixed issues can be found in the release notes for v25.7.0.

- DPUSet and BFB removal leaves DPU CRs in OS Installing Phase indefinitely
- DPUFlavor nvconfig.hostPowerCycleRequired is not used
- [VPC OVN DPUService] E/W Traffic Not Passing within Same VPC after DPU Reprovisioning
- DPUServiceInterface doesn't wait for ServiceInterface to be deleted
- DPUDeployment disruptive upgrade handling is not always graceful

Supported DPU Services:

- OVN-Kubernetes
- DOCA Host-Based Networking (HBN)
- DOCA Telemetry Service (DTS)
- DOCA BlueMan
- DOCA SNAP
- DOCA Firefly
- DOCA Argus
- OVN VPC

Dependencies

Hardware and Software Requirements

> **Note**
>
> For Host Trusted mode: Refer to Host Trusted Prerequisites for detailed requirements.
>
> For Zero Trust mode: Refer to Zero Trust Prerequisites for detailed requirements. In addition, notice the additional requirements for deploying via Redfish.

- DPU Hardware: NVIDIA BlueField-3 DPUs
- Minimal DOCA BFB Image: DOCA v2.5 or higher (must be pre-installed on DPUs to support DPF provisioning)
- Supported DOCA BFB Image: `bf-bundle-3.2.1`

Installation & Upgrade Notes

Installation Notes

None.

Upgrade Notes

> **Note**
>
> DPF supports upgrades from the immediate previous GA release. For more information refer to Lifecycle Management

- **[Zero-Trust] DpuDiscovery no longer creates DPUNodes**:
  - **Description:** DpuDiscovery no longer creates DPUNodes by default. Site-admin can enable it by setting skipDPUNodeDiscovery to false in DPFOperatorConfig.
  - **Example:** Set skipDPUNodeDiscovery to false in DPFOperatorConfig:

```yaml
apiVersion: operator.dpu.nvidia.com/v1alpha1
kind: DPFOperatorConfig
metadata:
  name: dpfoperatorconfig
  namespace: dpf-operator-system
spec:
  provisioningController:
    installInterface:
      installViaRedfish:
        skipDPUNodeDiscovery: false
```

- **Required field changes in DPUSet:**
  - **Description:** The following fields in `DPUSet` are now required and must be specified:
    - `spec.dpuTemplate` is now required
    - `spec.dpuTemplate.spec.dpuFlavor` is now required
  - **Impact:** Existing DPUSet resources that do not specify these fields will need to be updated before upgrading.
  - **Note:** These fields were made required because DPUSet resources cannot function properly without them. This change enforces validation that should have been in place from the beginning, making this effectively a bug fix rather than a breaking change for properly configured resources.
- **MTU minimum value changes:**
  - **Description:** The minimum MTU value has been increased from 1000 to 1280 bytes across multiple API fields. Additionally, maximum MTU constraints have been added where applicable.
  - **Affected Fields:**
    - `DPUServiceChain.spec.template.spec.template.spec.switches` - minimum increased: 1000 → 1280

        - `DPUDeployment.spec.serviceChains.switches[*].serviceMTU` - minimum increased: 1000 → 1280
        - `ServiceChain.spec.switches[*].serviceMTU` - minimum increased: 1000 → 1280
        - `ServiceChainSet.spec.template.spec.switches[*].serviceMTU` - minimum increased: 1000 → 1280
        - `DPFOperatorConfig.spec.networking.controlPlaneMTU` - minimum increased: 1000 → 1280
        - `DPFOperatorConfig.spec.networking.highSpeedMTU` - minimum increased: 1000 → 1280
        - `DPUFlavor.spec.hostNetworkInterfaceConfigs[*].mtu` - minimum increased: 1000 → 1280
        - `DPUServiceNAD.spec.serviceMTU` - minimum constraint added: 1280, maximum constraint added: 9216
    - **Impact:** Existing resources with MTU values below 1280 will need to be updated. The system will reject values below the new minimum during validation.
- **DPUDeployment now requires at least one and up to 50 services:**
    - **Description:** Previously, it was possible to define a DPUDeployment without any service. This was a bug in the validation which is now fixed. DPUDeployment now requires more than 1 and up 50 services.
    - **Affected Fields:**
        - `DPUDeployment.spec.services` - minimum constraint added when there was none previously : 1
        - `DPUDeployment.spec.services` - maximum constraint added when there was none previously : 50
    - **Impact:** Existing DPUDeployments without services should be replaced with a DPUSet instead. The system will reject creation or update of a DPUDeployment

with none or more than 50 services.

Deprecated API Fields

The following API fields have been deprecated and will be removed in future releases. Users should migrate to the recommended alternatives.

- **DPFOperatorConfig Fields:**
  - **Description:** Multiple fields in DPFOperatorConfig have been deprecated.
    - The top-level `image` fields have been replaced with container-specific image override fields for more granular control.
    - Registry configuration fields under `installViaRedfish` have been deprecated in favor of the top-level `registry` field.
    - The `deployInTargetCluster` field is deprecated and not supported.
  - **Deprecated Fields:**
    - Image fields (replaced with container-specific overrides):
    - `spec.provisioningController.image` → Use `spec.provisioningController.controller.image` instead
    - `spec.dpuServiceController.image` → Use `spec.dpuServiceController.controller.image` instead
    - `spec.dpuDetector.image` → Use `spec.dpuDetector.daemon.image` instead
    - `spec.kamajiClusterManager.image` → Use `spec.kamajiClusterManager.controller.image` instead
    - `spec.staticClusterManager.image` → Use `spec.staticClusterManager.controller.image` instead
    - `spec.serviceSetController.image` → Use `spec.serviceSetController.controller.image` instead

- `spec.flannel.image` → Use `spec.flannel.cni.image` and `spec.flannel.daemon.image` instead
- `spec.multus.image` → Use `spec.multus.cni.image` instead
- `spec.nvipam.image` → Use `spec.nvipam.controller.image` instead
- `spec.sriovDevicePlugin.image` → Use `spec.sriovDevicePlugin.deviceplugin.image` instead
- `spec.ovsCNI.image` → Use `spec.ovsCNI.cni.image` instead
- `spec.sfcController.image` → Use `spec.sfcController.controller.image` instead
- ProvisioningController registry fields:
- `spec.provisioningController.installInterface.installViaRe` → Use `spec.provisioningController.registry.address` instead
- `spec.provisioningController.installInterface.installViaRe` → Use `spec.provisioningController.registry` instead
- DeployInTargetCluster fields (not supported):
- `spec.serviceSetController.deployInTargetCluster` → This field is not supported and will be removed
- `spec.flannel.deployInTargetCluster` → This field is not supported and will be removed
- `spec.multus.deployInTargetCluster` → This field is not supported and will be removed
- `spec.nvipam.deployInTargetCluster` → This field is not supported and will be removed
- `spec.sriovDevicePlugin.deployInTargetCluster` → This field is not supported and will be removed

    - `spec.ovsCNI.deployInTargetCluster` → This field is not supported and will be removed
    - `spec.sfcController.deployInTargetCluster` → This field is not supported and will be removed
    - `spec.cniInstaller.deployInTargetCluster` → This field is not supported and will be removed

- **DPUDevice Fields:**
  - **Description:** The following fields in `DPUDevice.spec` have been deprecated in favor of their corresponding status fields, which are automatically populated by the system.
  - **Deprecated Fields:**
    - `spec.psid` → Use `status.psid` instead
    - `spec.opn` → Use `status.opn` instead
    - `spec.pf0Name` → Use `status.pf0Name` instead
  - **Note:** The status fields are considered read-only and automatically populated by the system. Users should not attempt to modify these fields.

- **DPUNode Fields:**
  - **Description:** The `gNOI` reboot method and `nodeDMSAddress` field have been deprecated.
  - **Deprecated Fields:**
    - `spec.nodeRebootMethod.gNOI` → Use `spec.nodeRebootMethod.hostAgent` instead
    - `spec.nodeDMSAddress` → This field is no longer used

- **DPUSet Fields:**
  - **Description:** The `maxUnavailable` field in rolling update strategy has been deprecated.

- **Deprecated Fields:**
  - `spec.strategy.rollingUpdate.maxUnavailable` → Use `dpfoperatorconfig.spec.provisioningController.maxUnavaila` instead

- **DPUFlavor Fields:**
  - **Description:** The `hostPowerCycleRequired` field in NVConfig has been deprecated as it is unused.
  - **Deprecated Fields:**
    - `spec.nvconfig[].hostPowerCycleRequired` → This field is unused and deprecated

- **DPU Fields:**
  - **Description:** The `bmcIP` field in DPU spec should be obtained from the associated DPUDevice instead.
  - **Deprecated Fields:**
    - `spec.bmcIP` → Use `DPUDevice.spec.bmcIp` or `DPUDevice.status.bmcIp` instead

- **DPUServiceNAD Fields:**
  - **Description:** The `spec.metadata` field in DPUServiceNAD has been deprecated. The standard Kubernetes metadata field at the resource level remains available.
  - **Deprecated Fields:**
    - `spec.metadata` → This field is unused and deprecated

Known Issues and Limitations

- **On initial deployment DPF CRs maybe stuck in initial state (Pending, Initializing, etc.) and not progressing**
    - In case DPF CRs were created before DPF components are running they may be stuck in their initial state. DPF CRs need to be created after the DPF components have been deployed. In case CRs were created before they may remain in an initial state.
    - Internal Ref #4241297
    - **Workaround:** Delete any CRs that were created before the System components have been deployed and recreate them.
- **Stale ports after DPU reboot**
    - When rebooting DPU, the old DPU service ports won't get deleted from DPU's OVS and would be stale
    - Internal Ref #4174183
    - **Workaround:** No workaround, known issue, shouldn't affect performance.
- **DPU Cluster control-plane connectivity is lost when physical port P0 is down on the worker node**
    - Link down of p0 port on the DPU will result in DPU control plane connectivity loss of DPU components.
    - Internal Ref #3751863
    - **Workaround:** Make sure P0 link is up on the DPU, if down either restart DPU or refer to DOCA troubleshooting https://docs.nvidia.com/networking/display/bfswtroubleshooting.

      Note: This issue is relevant for Host Trusted deployments only.
- **nvidia-k8s-ipam and servicechainset-controller DPF system DPUServices are in "Pending" phase**
    - As long as there are no provisioned DPUs in the system, the nvidia-k8s-ipam and servicechainset-controller will appear as not ready / pending when querying dpuservices. This has no impact on performance or functionality since DPF system components are only relevant when there are DPUs to provision services on.

    - Internal Ref #4241324
    - **Workaround:** No workaround, known issue

- **System doesn't recover after DPU Reset**
    - When the user triggers a reset of a DPU in any way other than using DPF APIs (e.g. recreation of a DPU CR), the system may not recover.
    - Internal Ref #4521178, #4188044, #4732664
    - **Workaround:** Power cycle the host. Note that this operation is dangerous and there might be file system corruption which will require triggering the reprovisioning of the DPU. If the system does not recover, reprovision the DPU by deleting the DPU CR (it will be recreated and DPU provisioning will happen).

      Note: This issue is relevant for Host Trusted deployments where OVN-Kubernetes is used as a primary CNI.

- **Leftover CRs if worker is removed from the cluster permanently**
    - When a worker was added to a cluster, optionally had DPU provisioned and later was removed from the host cluster permanently, there may be leftover DPF related CRs in both the host cluster and the DPU cluster.
    - Internal Ref #4403130, #4571788
    - **Workaround:** No workaround, known issue.

- **DPUDeployment or DPUSet created in namespace different from the dpf-operator-system namespace do not trigger DPU provisioning**
    - When creating a DPUDeployment or a DPUSet in a namespace other than dpf-operator-system, there are no DPU CRs created due to the DPUNode CRs residing in the dpf-operator-system namespace.
    - Internal Ref #4427091
    - **Workaround:** Create the DPUDeployment or DPUSet in the dpf-operator-system namespace.

- **DPUService stops reconciling when DPUCluster is unavailable for long time**

    - When the DPUCluster is unavailable for long time (more than 5 mins), changes to DPUServices (also generated ones via DPUDeployment or DPFOperatorConfig) that have happened during that time might not be reflected to the DPUCluster.
    - Internal Ref #4359857
    - **Workaround:** Recreate DPUServices that are stuck.
- **DPUFlavor nvconfig limitations**
    - DPUFlavor doesn't support assigning nvconfig parameters to only a specific device(s). Currently, it assigns the same parameters across all devices (i.e., '*')
    - Internal Ref #4583584
- **[OVN-Kubernetes DPUService] Nodes marked as NotReady**
    - When installing OVN-Kubernetes as a CNI on a node running containerd version 1.7.0 and above the Node never becomes ready.
    - Internal Ref #4178221
    - **Workaround:**
        - Option 1: Use containerd version below v1.7.0 when using OVN-Kubernetes as a primary CNI.
        - Option 2: Manually restart containerd on the host.
- **[OVN-Kubernetes DPUService] Lost traffic from workloads to control plane components or Kubernetes services after dpu reboot, port flapping, ovs restart or manual network configuration**
    - Connectivity issues between workload pods to control plane components or Kubernetes services may occur after the following events: DPU reboot without host reboot, high speed port flapping (link down/up), ovs restart, DPU network configuration change (for example using "netplan apply" command on DPU). The issues are caused by network configuration that was applied by ovn CNI on DPUs and won't get reapplied automatically.
    - Internal Ref #4188044, #4521178

    - **Workaround:** Recreate the OVN-Kubernetes node pod on the host to reapply the configuration.

- **[OVN-Kubernetes DPUService] host network configuration may result in lost traffic from host workloads (on overlay)**

    - When changing host network (for example with netplan apply) custom network configuration which is done by the host CNI components may be lost and won't reapply automatically.

    - Internal Ref #4188044

    - **Workaround:** Recreate the OVN-Kubernetes node pod on the host to reapply the configuration.

- **[OVN-Kubernetes DPUService] InternalTrafficPolicy and ExternalTrafficPolicy Service options are not handled correctly**

    - When external traffic reaches a NodePort Kubernetes Service on the host cluster via the OOB Network of the host, the user won't see the relevant policy option working as expected. The same applies for traffic originating from a Pod on the overlay network hitting the same type of service.

    - Internal Ref #4320953

    - **Workaround:** No workaround, known issue.

- **[OVN-Kubernetes DPUService] No network connectivity for SR-IOV accelerated workload pods after DPU reboot**

    - SR-IOV accelerated workload pod is losing its VF interface upon DPU reboot. VF is available on the host however not injected back into the pod.

    - Internal Ref #4236521

    - **Workaround:** Recreate the SR-IOV accelerated workload pods.

- **[HBN DPUService] Invalid HBN configuration is not reflected to user in case it is syntactically valid**

    - If the HBN YAML configuration is valid but contains values that are illegal from an NVUE perspective then the HBN service will start with the last known valid configuration and it won't be reflected to the end user.

- Internal Ref #4555216
- **Workaround:** No workaround, known issue.

- **[Firefly DPUService] Deletion of the Firefly DPUService leaves stale flows in OVS**
  - When the Firefly DPUService is deleted after successful deployment or the labels of the serviceDaemonSet are modified, flows are not cleaned up from OVS.
  - Internal Ref #4382535
  - **Workaround:** Although these flows are very unlikely to cause an issue, reprovisioning the DPU or power cycling the host will bring the OVS in good state. Note that power cycling the host is dangerous and there might be file system corruption which will require triggering the reprovisioning of the DPU.

- **Changing DPUNode nodeRebootMethod during DPU provisioning is not supported**
  - When a DPU is not in Ready or Error phase, updating the DPUNode `spec.nodeRebootMethod` is not supported. Updating `spec.nodeRebootMethod` at this time may cause the DPU to get stuck in `Rebooting` phase.
  - Internal Ref: #4641260
  - **Workaround:** Delete the DPU object to trigger reprovisioning.

- **DPU may fail to reboot during Rebooting phase if firmware version is too old**
  - If the firmware version of the DPU is older than version 32.41.1000 (released April 2024), DPUs may fail to provision with an error during the DPU reboot phase.
  - Internal Ref: #4634335
  - **Workaround:** Manually upgrade the firmware version of the DPU to 32.41.1000 or later, or set `spec.nodeRebootMethod` to `external` (and reboot the node at the correct time).

- **[DPUDeployment] Pods of service referenced in spec.serviceChain are restarted when serviceChain changes, causing disruption**

- When updating DPUDeployment.spec.serviceChain with a disruptive upgrade, the affected pods are restarted but DPUDeployment does not remove the node effect from the corev1.Node in the host cluster. This causes the node to be marked as ready and accept workloads before DPU services are fully operational.
- Internal Ref: #4686635
- **Workaround:** No workaround, known issue.

- **[DPUDeployment] Invalid configuration in service and service chain related configuration may lead to DPUDeployment making no progress, even after reverting to a working configuration.**
  - When an invalid change is introduced in DPUServiceConfiguration/DPUServiceTemplate or the DPUDeployment.spec.serviceChain, the DPUDeployment may get stuck. This happens when one or more DPUs get to the Node Effect Removal state. The DPU is stuck because the underlying component doesn't become ready (Pod/ServiceChain). Even reverting to the working configuration won't work because the DPU controller doesn't adjust the labels on the corev1.Node in the DPUCluster when the DPU is in Node Effect Removal state.
  - Internal Ref: #4713965
  - **Workaround:** Revert to a working configuration and remove the requestors from the DPUNodeMaintenance objects.

- **No network traffic through DPU when the system is properly configured and all related DPF resources are ready**
  - In rare cases, a race condition during the initialization of the OVS instance running on the DPU may prevent traffic from passing through the DPU.
  - Internal Ref: #4719221
  - **Workaround:** The following workarounds are possible:
    - **Step 1:** Powercycle the host
    - **Step 2:** Add the following command to DPUFlavor `spec.ovs` field: `ovs-appctl dpctl/del-dp system@ovs-system`

- **DPU provisioning in Zero-Trust may fail if the provisioning controller has restarted during DPU provisioning**
    - If the provisioning controller restarts while a DPU is being provisioned, the DPU will transition to the `Error` phase. In the provisioning controller logs, a `400 Bad Request` error is seen with message `An update is in progress.` while communicating with the DPU BMC.
    - Internal Ref: #4723795
    - **Workaround:** Wait a few minutes for the current in-flight DPU provisioning process to be completed by the DPU BMC, then delete the DPU object to restart provisioning.
- **DOCA SNAP may fail to reattach a DPUVolumeAttachment that uses VirtioFS on hot-plugged PFs after a host power cycle**
    - After a host power cycle, a DPUVolumeAttachment that uses VirtioFS on hot-plugged PFs can become not ready and might not be restored automatically.
    - Internal Ref: #4734537
    - **Workaround:**
        - **Step 1:** Delete failed DPUVolumeAttachment.
        - **Step 2:** Create a new DPUVolumeAttachment with the same configuration.
- **DOCA SNAP may fail to reattach a DPUVolumeAttachment that uses emulated NVMe devices on hot-plugged PFs after a host power cycle**
    - After a host power cycle, a DPUVolumeAttachment that uses emulated NVMe devices on hot-plugged PFs can become not ready and might not be restored automatically.
    - Internal Ref: #4734514
    - **Workaround:**
        - **Step 1:** Mark the DPUVolumeAttachment for deletion. The object removal will be blocked by the finalizer at this step.

        - **Step 2:** Access the API of the DPU cluster and remove the finalizer from the VolumeAttachment CR that has the same name as the DPUVolumeAttachment CR in the host cluster.
        - **Step 3:** Wait for the DPUVolumeAttachment to be fully removed in the host cluster.
        - **Step 4:** Power cycle the host.
        - **Step 5:** After the host is fully booted, create a new DPUVolumeAttachment with the same configuration.

- **In rare cases, a DPUVolumeAttachment that uses emulated NVMe devices on hot-plugged PFs may reach the Ready state but have an invalid (00:00.0) PCI device address in the CR status**
    - If the host is still booting while a DPUVolumeAttachment is in progress, the DPUVolumeAttachment may reach the Ready state but have an invalid (00:00.0) PCI device address, which prevents the volume from being consumed by workloads on the host.
    - Internal Ref: #4734509
    - **Workaround:**
        - **Step 1:** Mark the DPUVolumeAttachment for deletion. The object removal will be blocked by the finalizer at this step.
        - **Step 2:** Access the API of the DPU cluster and remove the finalizer from the VolumeAttachment CR that has the same name as the DPUVolumeAttachment CR in the host cluster.
        - **Step 3:** Wait for the DPUVolumeAttachment to be fully removed in the host cluster.
        - **Step 4:** Power cycle the host.
        - **Step 5:** After the host is fully booted, create a new DPUVolumeAttachment with the same configuration.

- **Long DPU provisioning time when multiple DPUs are provisioned on the same node**
    - In some cases, it may take a long amount of time to provision multiple DPUs on the same node.

    - Internal Ref: #4757191
    - **Workaround:** If the DPU is stuck in `DPU Cluster Config` phase after a long installation, please reprovision the DPU and make sure no other DPUs on the same node are being provisioned at the same time

- **DPF does not support HBN with EVPN configuration together with OVN-Kubernetes or DPF OVN VPC services**
    - Mixing GENEVE(OVN) and VXLAN(EVPN) tunnel types in HBN will cause traffic failure
    - Internal Ref: #4776088
    - **Workaround:** No workaround. This configuration should be avoided.

Troubleshooting

Troubleshooting DOCA Platform Framework

This section provides comprehensive troubleshooting guidance for common issues you may encounter while deploying, configuring, or operating the DOCA Platform Framework (DPF).

Quick Diagnostic Tools

DPF CLI (dpfctl)

Command-line tool for visualizing, debugging, and troubleshooting DPU resources in Kubernetes. Essential for real-time visibility into resource states and conditions.

Use when:

- DPU provisioning is failing
- Need to understand resource dependencies
- Debugging component readiness issues

System Reports (sosreport)

Generate comprehensive system reports for deeper analysis and support requests.

Use when:

- Need detailed system information for support cases
- Investigating complex infrastructure issues
- Preparing diagnostic data for NVIDIA support

Escalation Path

If you cannot resolve the issue using the guides above:

1. **Collect Diagnostic Information** * Generate a sosreport for your environment
2. **Check Known Issues** * Review Release Notes for known issues * Search the GitHub repository for similar problems
3. **Contact Support** * Open an issue on the GitHub repository * Include diagnostic information and steps to reproduce * For enterprise customers, contact NVIDIA support with your diagnostic package

Additional Resources

- **User Guides** - Operational procedures and best practices
- **Architecture** - Understanding system design for better troubleshooting
- **API Reference** - Complete API documentation for debugging configurations

DPF CLI - dpfctl

`dpfctl` is a command-line tool designed to **visualize, debug, and troubleshoot DPU resources** in Kubernetes.

It simplifies debugging by extracting and presenting resource conditions in a structured, human-readable format.

It helps during provisioning and debugging of DPF by providing real-time visibility into resource states, conditions, and relationships.

For example, during DPU provisioning you can quickly identify which resources are ready, which are failing, and understand the dependency chain between components.

```
Every 2.0s: kubectl -n dpf-operator-system exec deploy/dpf-operator-controller-manager -- /dpfctl describe dpusets

NAME                                      NAMESPACE           STATUS  REASON         SINCE  MESSAGE
DPFOperatorConfig/dpfoperatorconfig       dpf-operator-system
           ├Ready                                             False   Pending        2m37s  The following conditions are not ready:
                                                                                            * SystemComponentsReady
           └SystemComponentsReady                             False   Error          82s    System components must be ready for DPF Operator to continue:
                                                                                              * nvidia-k8s-ipam: DPUService dpf-operator-system/nvidia-k8s-ipam is not ready
└DPUSets
  └DPUSet/dpuset-1                        dpf-operator-system
    ├BFB/bf-bundle-3.0.0-135              dpf-operator-system
    │         └Ready                                          False   Downloading    84s
    ├DPU/c-234-183-              0dhg     dpf-operator-system
    │         ├BFBReady                                       False   BFBIsNotReady  33s    BFB is not ready
    │         └Ready                                          False   Pending        33s
    └DPU/c-234-183-              0dfy     dpf-operator-system
              ├BFBReady                                       False   BFBIsNotReady  38s    BFB is not ready
              └Ready                                          False   Pending        38s
```

How to Run

There are 2 ways to run `dpfctl`:

- **Install**: You can download & run `dpfctl` locally on your machine
- **Execute Inside Kubernetes Pod**: With this way you can ensure that you are using the correct version of `dpfctl` that is compatible with the DPF Operator

Install

To download the latest version of `dpfctl`, you can find the latest version on NGC doca/dpfctl.

There are 3 different versions available for different architectures:

- `dpfctl-linux-amd64`
- `dpfctl-linux-arm64`
- `dpfctl-darwin-arm64`

You can also download it directly from the command line (in this example we are using `dpfctl-linux-amd64` with version `v25.10.0`):

```
curl -L -o /usr/local/bin/dpfctl
https://api.ngc.nvidia.com/v2/resources/nvidia/doca/dpfctl/versions/v25.10.0/files/dpfctl-linux-amd64
chmod +x /usr/local/bin/dpfctl
```

Execute Inside Kubernetes Pod

To execute `dpfctl` from the running container:

```
kubectl -n dpf-operator-system exec deploy/dpf-operator-controller-manager -- /dpfctl describe all
```

Creating a Shell Alias

For convenience, you can create a shell alias to simplify your commands:

```
alias dpfctl="kubectl -n dpf-operator-system exec deploy/dpf-operator-controller-manager -- /dpfctl "
```

If you want to use `watch` according to `man watch`:

> If the last character of the alias value is a blank, then the next command word following the alias is also checked for alias expansion.

To use `watch` with custom arguments (like `-n1` for interval), add both aliases to your shell's configuration file (e.g., `.bashrc`, `.zshrc`, or `.profile`):

```
# Add to your shell's configuration file
alias dpfctl="kubectl -n dpf-operator-system exec deploy/dpf-operator-controller-manager -- /dpfctl "
alias watch="watch -c "
```

Usage

```
dpfctl describe [command] [flags]
```

Available Commands:

Command	Description
all	Describe all DPF resources
dpuclusters	Describe DPF DPUClusters
dpudeployments	Describe DPF DPUDeployments
dpuservices	Describe DPF DPUServices
dpusets	Describe DPF DPUSets and DPU related resources
dpuvpcs	Describe DPF DPUVPCs and VPC related resources
storage	Describe DPF Storage resources

> **Note**
>
> Available flags can be found with `dpfctl describe --help`.

By default, `dpfctl describe` provides an overview of key **DPU-related resources and their conditions**.

Example output of `dpfctl describe all`:

```
NAME                                                    NAMESPACE
STATUS       REASON             SINCE  MESSAGE
DPFOperatorConfig/dpfoperatorconfig                     dpf-
operator-system  Ready: True  Success            23h
  DPUClusters
    DPUCluster/dpu-cplane-tenant1                       dpu-cplane-
tenant1   Ready: True  HealthCheckPassed  4d20h
  DPUDeployments
    DPUDeployment/vpc-ovn                               dpf-
operator-system  Ready: True  Success            23h
      DPUServiceChains
        DPUServiceChain/vpc-ovn-trsq6                   dpf-
operator-system  Ready: True  Success            23h
      DPUSets
        DPUSet/vpc-ovn-dpuset1                          dpf-
operator-system
          BFB/bf-bundle                                 dpf-
operator-system  Ready: True  Ready              3d17h  File: bf-
bundle-3.2.1-34_25.11_ubuntu-24.04_64k_prod.bfb, DOCA: 3.2.1
          DPUs
            4 DPUs...                                   dpf-
operator-system  Ready: True  DPUReady           3d17h  See dpu-
node-mt2310xz03lr-mt2310xz03lr, dpu-node-mt2310xz03m2-
mt2310xz03m2,

dpu-node-mt2425601x13-mt2425601x13, dpu-node-mt2425601xqy-
mt2425601xqy
      Services
        DPUServiceTemplates
          4 DPUServiceTemplates...                      dpf-
operator-system  Ready: True  Success            3d17h  See ovn-
central, ovn-controller, vpc-ovn-controller, vpc-ovn-node
        DPUServices
          4 DPUServices...                              dpf-
operator-system  Ready: True  Success            3d1h   See ovn-
```

```
central-9558p, ovn-controller-v5bkr, vpc-ovn-controller-7sbp6,
vpc-ovn-node-r84zn
  DPUServiceCredentialRequests
    DPUServiceCredentialRequest/servicesetcontroller  dpf-
operator-system  Ready: True  Success            4d20h
  DPUServiceIPAMs
    2 DPUServiceIPAMs...                              dpf-
operator-system  Ready: True  Success            3d17h  See vpc-
ippool-gateway, vpc-ippool-vtep
  DPUServiceInterfaces
    4 DPUServiceInterfaces...                         dpf-
operator-system  Ready: True  Success            3d17h  See ovn-
ext-patch, ovn-vtep-patch, p0, vf2-test
  DPUVPCs
    DPUVPC/myvpc                                      dpf-
operator-system  Ready: True  Success            2d
      IsolationClass/ovn.vpc.dpu.nvidia.com
Ready: True  Available          3d     Provisioner:
ovn.vpc.dpu.nvidia.com
      DPUVirtualNetwork/testnet1                      dpf-
operator-system  Ready: True  Success            2d
        DPUServiceInterface/vf2-test                  dpf-
operator-system  Ready: True  Success            2d
  System Components                                   dpf-
operator-system
    DPUServices
      8 DPUServices...                                dpf-
operator-system  Ready: True  Success            4d20h  See
flannel, multus, nvidia-k8s-ipam, ovs-cni, servicechainset-
controller,

servicechainset-rbac-and-crds, sfc-controller, sriov-device-
plugin
```

Storage resources are not included in the output of `dpfctl describe all` command by default. To display storage resources, use the `--show-storage` flag.

The `dpfctl describe storage` command can be used to display only storage resources. It acts as a shortcut for `dpfctl describe all --show-storage --show-resources dpuvolumes,dpuvolumeattachments,dpustoragevendors,dpustoragepolicies`.

Kubeconfig

By default, `dpfctl` uses the kubeconfig file at `~/.kube/config`. To use a different kubeconfig file, specify it with `--kubeconfig`, which is part of the global flags:

```
dpfctl describe all --kubeconfig /path/to/kubeconfig
```

Alternatively, set the `KUBECONFIG` environment variable:

```
export KUBECONFIG=/path/to/kubeconfig
dpfctl describe all
```

or

```
KUBECONFIG=/path/to/kubeconfig dpfctl describe all
```

Visualizing the Output

We can customize the output by using flags.

Show & Expand Resources

For example `dpfctl describe all --show-resources=dpuservice` will show only DPUService resources.

```
NAME                                    NAMESPACE            STATUS
REASON   SINCE  MESSAGE
DPFOperatorConfig/dpfoperatorconfig  dpf-operator-system  Ready:
True  Success  23h
  DPUServices
    4 DPUServices...                   dpf-operator-system  Ready:
True  Success  3d1h   See ovn-central-9558p, ovn-controller-
v5bkr, vpc-ovn-controller-7sbp6, vpc-ovn-node-r84zn
  System Components                    dpf-operator-system
    DPUServices
      8 DPUServices...                 dpf-operator-system  Ready:
True  Success  4d20h  See flannel, multus, nvidia-k8s-ipam, ovs-
cni, servicechainset-controller,

servicechainset-rbac-and-crds, sfc-controller, sriov-device-
plugin
```

If you want to show multiple different resources you can add a comma-separated list of resources to the `--show-resources` flag.

```
> dpfctl describe all --show-resources dpuservice,dpuset
NAME                                  NAMESPACE            STATUS
REASON    SINCE  MESSAGE
DPFOperatorConfig/dpfoperatorconfig  dpf-operator-system  Ready:
True  Success   23h
  DPUServices
    4 DPUServices...                 dpf-operator-system  Ready:
True  Success   3d1h   See ovn-central-9558p, ovn-controller-
v5bkr, vpc-ovn-controller-7sbp6, vpc-ovn-node-r84zn
  DPUSets
    DPUSet/vpc-ovn-dpuset1           dpf-operator-system
      BFB/bf-bundle                  dpf-operator-system  Ready:
True  Ready     3d17h  File: bf-bundle-3.2.1-34_25.11_ubuntu-
24.04_64k_prod.bfb, DOCA: 3.2.1
      DPUs
        4 DPUs...                    dpf-operator-system  Ready:
True  DPUReady  3d17h  See dpu-node-mt2310xz03lr-mt2310xz03lr,
dpu-node-mt2310xz03m2-mt2310xz03m2,

dpu-node-mt2425601x13-mt2425601x13, dpu-node-mt2425601xqy-
mt2425601xqy
  System Components                  dpf-operator-system
    DPUServices
      8 DPUServices...               dpf-operator-system  Ready:
True  Success   4d20h  See flannel, multus, nvidia-k8s-ipam, ovs-
cni, servicechainset-controller,

servicechainset-rbac-and-crds, sfc-controller, sriov-device-
plugin
```

To expand child objects:

```
> dpfctl describe all --show-resources dpuservice --expand-
resources dpuservice --grouping=false
NAME
NAMESPACE            READY        REASON   SINCE  MESSAGE
[...]
    DPUService/ovn-central-9558p
dpf-operator-system  Ready: True  Success  26h
      Application/in-cluster-ovn-central-9558p
dpf-operator-system
                    Deployment/in-cluster-ovn-central-9558p
Synced       Healthy  31s
                    Ready
True         Success  31s
    DPUService/ovn-controller-v5bkr
dpf-operator-system  Ready: True  Success  47h
      Application/dpu-cplane-tenant1-ovn-controller-v5bkr
dpf-operator-system
                    DaemonSet/dpu-cplane-tenant1-ovn-controller-
v5bkr                       Synced       Healthy  31s
                    Ready
True         Success  31s
```

Note

The flag `--expand-resources` is currently supported only for `DPUServices`. Further support for other resources will be added in future releases.

Grouping

Grouping combines resources of the same kind. To disable grouping:

```
> dpfctl describe all --show-resources dpuservice --grouping=false
NAME                                                NAMESPACE
STATUS       REASON   SINCE  MESSAGE
DPFOperatorConfig/dpfoperatorconfig                 dpf-operator-
system  Ready: True  Success  7m14s
  DPUServices
    DPUService/ovn-central-9558p                    dpf-operator-
system  Ready: True  Success  26h
    DPUService/ovn-controller-v5bkr                 dpf-operator-
system  Ready: True  Success  47h
    DPUService/vpc-ovn-controller-7sbp6             dpf-operator-
system  Ready: True  Success  2d2h
    DPUService/vpc-ovn-node-r84zn                   dpf-operator-
system  Ready: True  Success  34m
  System Components                                 dpf-operator-
system
    DPUServices
      DPUService/flannel                            dpf-operator-
system  Ready: True  Success  47h
      DPUService/multus                             dpf-operator-
system  Ready: True  Success  47h
      DPUService/nvidia-k8s-ipam                    dpf-operator-
system  Ready: True  Success  47h
      DPUService/ovs-cni                            dpf-operator-
system  Ready: True  Success  25h
      DPUService/servicechainset-controller         dpf-operator-
system  Ready: True  Success  3d21h
      DPUService/servicechainset-rbac-and-crds      dpf-operator-
system  Ready: True  Success  3d21h
      DPUService/sfc-controller                     dpf-operator-
system  Ready: True  Success  26h
      DPUService/sriov-device-plugin                dpf-operator-
system  Ready: True  Success  47h
```

Conditions

To show resource conditions:

```
dpfctl describe all --show-conditions=all
```

Example output:

```
NAME                                                        NAMESPACE
STATUS  REASON                SINCE  MESSAGE
[...]
  System Components                                         dpf-
operator-system
    DPUServices
      DPUService/flannel                                    dpf-
operator-system
                    Ready
True    Success               47h
                    ApplicationPrereqsReconciled
True    Success               3d21h
                    ApplicationsReady
True    Success               47h
                    ApplicationsReconciled
True    Success               3d21h
                    ConfigPortsReconciled
True    Success               3d21h
                    DPUServiceInterfaceReconciled
True    Success               3d21h
```

Reporting Issues with sosreport

Environment

- DOCA Platform Framework Kubernetes cluster

Issue

- What is the recommended way for generating a sosreport in DOCA Platform Framework
- It may not be possible to connect to DOCA Platform Framework nodes via SSH from outside the cluster by default but sosreport may need to be run for troubleshooting purposes.

Generating a SOS Report with a Debug Pod

Target Host Cluster

Create a secret containing the kubeconfig

In order to run `sosreport`, a `kubeconfig` is needed to access the API Server.

1. Create a secret containing the `kubeconfig`

```
kubectl create secret generic admin-config --from-
file=kubeconfig=<path_to_kubeconfig>
```

Deploy sos-report

1. Display the list of nodes in the cluster and export the selected node. The following command will display the list of nodes:

```
kubectl get nodes
```

2. Then create a debug pod by deploying the following manifest:

```
cat <<EOF | kubectl create -f -
apiVersion: v1
kind: Pod
metadata:
  name: dpf-sosreport
spec:
  nodeName: ${NODE_NAME}
  containers:
  - name: sosreport
    image: ghcr.io/nvidia/sosreport:latest
    env:
    - name: CASE_ID
      value: "${CASE_ID}"
    imagePullPolicy: IfNotPresent
    securityContext:
      privileged: true
      runAsUser: 0
    volumeMounts:
      - mountPath: /host
        name: host
      - mountPath: /run
        name: run
      - mountPath: /var/log
        name: varlog
        # sosreport check if this file exist before executing the
kubernetes plugin,
        # without it no kubernetes output will be available.
      - mountPath: /etc/kubernetes/admin.conf
        name: adminconf
        subPath: kubeconfig
      - mountPath: /etc/localtime
        name: localtime
      - mountPath: /etc/machine-id
        name: machineid
      - mountPath: /boot
```

```
        name: boot
      - mountPath: /usr/lib/modules/
        name: modules
  volumes:
    - hostPath:
        path: /
      name: host
    - hostPath:
        path: /run
      name: run
    - hostPath:
        path: /boot
      name: boot
    - hostPath:
        path: /usr/lib/modules/
      name: modules
    - hostPath:
        path: /var/log
      name: varlog
    - secret:
        secretName: admin-config
      name: adminconf
    - hostPath:
        path: /etc/localtime
      name: localtime
    - hostPath:
        path: /etc/machine-id
      name: machineid
  restartPolicy: Never
  hostIPC: true
  hostNetwork: true
  hostPID: true
EOF
```

Target Tenant Cluster

Find the tenant cluster kubeconfig

In order to run `sosreport`, a `kubeconfig` is needed to access the API Server. When the report has to be generated for a tenant cluster, we have to retrieve the `kubeconfig` from the host cluster.

1. Get the `kubeconfig` name from the dpucluster `spec`.

```
export KUBECONFIG_NAME=$(kubectl get dpucluster -n ${NAMESPACE}
${CLUSTER_NAME} -o jsonpath='{.spec.kubeconfig}')
```

2. Create the `kubeconfig` from the secret data

```
kubectl get secrets -n ${NAMESPACE} ${KUBECONFIG_NAME} -o json \
  | jq -r '.data["super-admin.conf"]' \
  | base64 --decode \
  > /tmp/${NAMESPACE}-${CLUSTER_NAME}.kubeconfig
```

3. Create a secret containing the `kubeconfig` in the tenant cluster

```
kubectl create secret generic admin-config --from-
file=kubeconfig=/tmp/${NAMESPACE}-${CLUSTER_NAME}.kubeconfig \
  --kubeconfig=/tmp/${NAMESPACE}-${CLUSTER_NAME}.kubeconfig
```

Deploy sos-report

1. Display the list of nodes in the cluster and export the selected node. The following command will display the list of nodes:

```
kubectl get nodes
```

2. Then create a debug pod by deploying the following manifest:

```
cat <<EOF | kubectl --
kubeconfig=/tmp/${NAMESPACE}-${CLUSTER_NAME}.kubeconfig create -f
-
apiVersion: v1
kind: Pod
metadata:
  name: dpf-sosreport
spec:
  nodeName: ${NODE_NAME}
  containers:
  - name: sosreport
    image: ghcr.io/nvidia/sosreport:latest
    env:
    - name: CASE_ID
      value: "${CASE_ID}"
    imagePullPolicy: IfNotPresent
    securityContext:
      privileged: true
      runAsUser: 0
    volumeMounts:
      - mountPath: /host
        name: host
      - mountPath: /run
        name: run
      - mountPath: /var/log
        name: varlog
        # sosreport check if this file exist before executing the
kubernetes plugin,
        # without it no kubernetes output will be available.
      - mountPath: /etc/kubernetes/admin.conf
        name: adminconf
        subPath: kubeconfig
      - mountPath: /etc/localtime
        name: localtime
      - mountPath: /etc/machine-id
```

```
        name: machineid
      - mountPath: /boot
        name: boot
      - mountPath: /usr/lib/modules/
        name: modules
  volumes:
    - hostPath:
        path: /
      name: host
    - hostPath:
        path: /run
      name: run
    - hostPath:
        path: /boot
      name: boot
    - hostPath:
        path: /usr/lib/modules/
      name: modules
    - hostPath:
        path: /var/log
      name: varlog
    - secret:
        secretName: admin-config
      name: adminconf
    - hostPath:
        path: /etc/localtime
      name: localtime
    - hostPath:
        path: /etc/machine-id
      name: machineid
  restartPolicy: Never
  hostIPC: true
  hostNetwork: true
  hostPID: true
EOF
```

Retrieve the Generated Report

The final repost archive is available under `/tmp` in the node filesystem.

In order to untar it, run :

```
tar -x --xz -f sosreport-<node_name>-<case_id>-<date>-xxx.tar.xz
```

User Guides

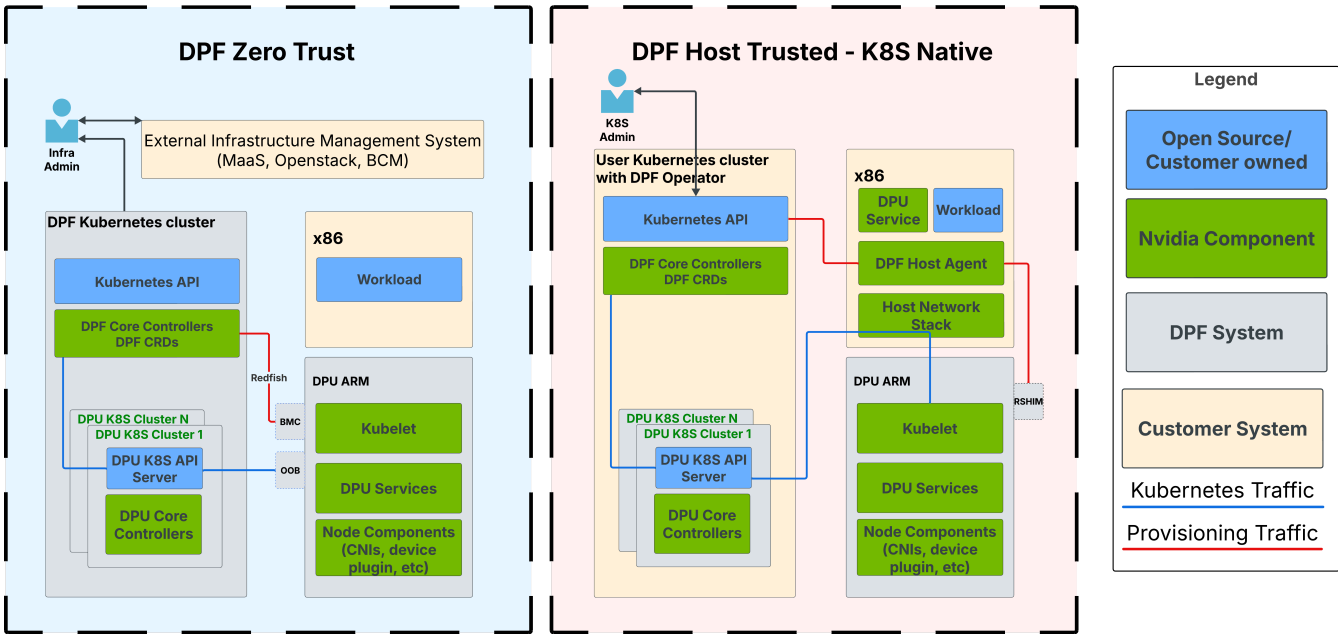
DPF User Guides

These guides walk you through deploying and configuring the DOCA Platform Framework (DPF) for different scenarios.

Deployment Scenarios

We have two primary deployment scenarios for DPF - Zero Trust and Host Trusted. Each scenario has its own deployment guide and configuration details.

In the below diagram, you can see the differences between the two deployment modes:



Zero Trust Deployment

Zero Trust Guide

In Zero Trust deployments, the DPU acts as a host accelerator card managed by DPF via BMC and Redfish. Your host machines stay outside the DPF management cluster, while the DPU provides secure acceleration for those hosts.

This approach gives you:

- DPU managed through DPF via BMC and Redfish
- Secure isolation between management plane and host systems
- DPU services managed via Kubernetes APIs within the DPF cluster
- Ideal for isolation use cases: HBN, VPC, SNAP isolated storage, and Argus

Host Trusted Deployment

Host Trusted Guide

In Host Trusted deployments, the DPU acts as a host accelerator card managed by the host. Your host machines function as worker nodes within the DPF management cluster, with the DPU serving as an integrated acceleration resource.

This approach gives you:

- Host-managed DPU
- Standard Kubernetes APIs for both host workloads and DPU services
- End-to-end Kubernetes-native experience
- Support for HBN, OVN Kubernetes, and HBN+OVN Kubernetes use cases
- DPU configuration through DOCA Management Service (DMS)

Host Trusted Deployment

Overview

The **Host Trusted** deployment scenario represents DPF for Host Trusted environments, where the DPU is treated as a **host accelerator card** and is **managed by the host**. In this deployment model, host machines function as worker nodes within the DPF management cluster, with the DPU serving as an integrated acceleration resource that enhances the host's capabilities.

This architecture allows users to leverage standard Kubernetes APIs for managing both host workloads and DPU services, creating an end to end Kubernetes-native experience.

The DOCA Management Service(DMS) is used to install and configure the DPU.

Use Cases

The Host trusted model supports various use cases, each optimized for specific networking and acceleration requirements.

Each use case may require tailored configurations on the DPU for optimal performance. Below are the validated user guides with corresponding deployment guides.

HBN and OVN Kubernetes

- Host Based Networking and OVN Kubernetes

DPU Services	Comments
DOCA Host-Based Networking (HBN)	Accelerates underlay BGP routing with ECMP from the DPU
OVN Kubernetes with DPU offload	Provides SDN overlay services and Kubernetes CNI offloads
DOCA Telemetry Service (DTS)	Provides enhanced DPU telemetry
DOCA BlueMan	Offers a user-friendly GUI for DTS

OVN Kubernetes

- OVN Kubernetes on its own

DPU Services	Comments
OVN Kubernetes with DPU offload	Provides SDN overlay services and Kubernetes CNI offloads
DOCA Telemetry Service (DTS)	Provides enhanced DPU telemetry
DOCA BlueMan	Offers a user-friendly GUI for DTS

HBN

- Host Based Networking on its own

DPU Services	Comments
DOCA Host-Based Networking (HBN)	Accelerates underlay BGP routing with ECMP and EVPN-based overlays

DPU Passthrough

- DPU Passthrough

No DPUServices are necessary for this use case.

HBN and SNAP Storage

- Host Based Networking and SNAP Storage

DPU Services	Comments
DOCA Host-Based Networking (HBN)	Accelerates underlay BGP routing with ECMP and EVPN-based overlays
DOCA SNAP	Provides shared storage for Kubernetes workloads

HBN and OVN Kubernetes and SNAP Storage

- Host Based Networking and OVN Kubernetes and SNAP Storage

DPU Services	Comments
DOCA Host-Based Networking (HBN)	Accelerates underlay BGP routing with ECMP and EVPN-based overlays
OVN Kubernetes with DPU offload	Provides SDN overlay services and Kubernetes CNI offloads

DPU Services	Comments
DOCA Telemetry Service (DTS)	Provides enhanced DPU telemetry
DOCA BlueMan	Offers a user-friendly GUI for DTS
DOCA SNAP	Provides shared storage for Kubernetes workloads

Prerequisites

Before deploying DPF in Host Trusted mode, ensure you have the necessary hardware and software prerequisites in place. This includes compatible DPU hardware, host operating system configurations, and network settings. For detailed requirements, refer to the prerequisites guide.

Support and Troubleshooting

For assistance with Host Trusted deployments:

- Review the specific use case documentation
- Check the DPF troubleshooting guide
- Consult the release notes for known issues
- Engage with the NVIDIA support team for further assistance

Next Steps

Once you've familiarized yourself with the Host Trusted concept, proceed to:

1. Review the prerequisites.
2. Select an appropriate use case.
3. Begin your DPF Host Trusted deployment.

DPF Prerequisites for Host Trusted

Before deploying DPF, ensure the following prerequisites are met:

- DPF System Prerequisites for Host Trusted
- DPF Operator Helm Prerequisites
- Host Network Configuration Prerequisites

DPF System Prerequisites for Host Trusted

DPF makes a number of assumptions about the hardware, software and networking of the machines it runs on. Some of the specific user guides add their own requirements.

Hardware Setup

There is a high availability control plane machines serving many worker nodes in a cluster running DPF.

Control Plane Machines

Each control plane machine:

- May be virtualized
- x86_64 architecture
- 16 GB RAM
- 8 CPUs
- DPUs are not installed

Worker Machines

Each worker machine:

- Bare metal - no virtualization
- x86_64 architecture

- 16 GB RAM
- 8 CPUs
- Any number of DPUs

DPUs

- Bluefield 3
- 32 GB memory
- Flashed with NVIDIA BFB with DOCA version 2.5 or higher
- Out-of-band management port is not used

System Software Setup

Control Plane Machines

- NFS client packages - i.e. `nfs-common`
- NFS server available with `/mnt/dpf_share` readable and writable by any user

Worker Machines

- `In-Band Manageability Interface` enabled in BIOS
- NFS client packages - i.e. `nfs-common`
- NFS server available with `/mnt/dpf_share` readable and writable by any user
- rshim package is not installed

Kubernetes

- Kubernetes 1.32 - 1.34
- Control plane nodes have the labels `"node-role.kubernetes.io/control-plane" : ""`

Network Setup

- All nodes have full internet access - both from the host out-of-band and DPU high speed interfaces
- Virtual IP from the management subnet reserved for internal DPF usage
- The out-of-band management and high-speed networks are routable to each other
- The control plane nodes hosting the DPU control plane pods must be located on the same L2 broadcast domain
- The out-of-band management fabric on which control plane nodes are connected should allow MultiCast traffic (used for VRRP)

Host Network Configuration Prerequisites

Overview

To facilitate connectivity between the Data Processing Unit (DPU) and the Kubernetes control plane (API server) and the DHCP server on the management network, a fundamental network configuration is necessary on the host. The appropriate solution may vary depending on the specific environment. This guide provides a basic example for a host with Ubuntu 24.04 OS.

Assumption

This configuration is intended to be implemented during the host provisioning phase, prior to its integration into the Kubernetes cluster.

Host-DPU Network Diagram

The solution involves setting up a linux bridge with an IP address and connecting the Out-of-Band (OOB) uplink (a 1 Gbps Ethernet port) to it. During the DPU provisioning process, a Virtual Function (VF) will be created for the DPU and connected to the bridge to enable connectivity.

```
             +--------------------------------+  +-------------+
             | Host                           |  | DPU         |
             |        +---------+             |  |             |
Management---|--OOB---|  br-dpu |--ensXf0vf0--|--|--pf0vf0     |
  Network    | uplink +---------+             |  | representor |
             | (No IP)    IP                  |  |             |
             |                                |  |             |
             +--------------------------------+  +-------------+
```

Example Solution - Set Up The Bridge Using Netplan

Utilize Netplan to configure the necessary network interfaces. Netplan configuration files are executed at boot time before the kubelet process.

The following YAML illustrates the requirements for the Netplan configuration. It ensures the following:

1. No IP, gateway or DNS nameservers is configured on the `oob` interface.
2. The `oob` interface is connected to the bridge `br-dpu`.
3. The bridge is always named `br-dpu`. The name of the OOB interface does not matter.
4. All network connectivity requirements, such as IP, gateway, and nameservers, are configured on br-dpu.

```
network:
  ethernets:
    oob:
      # dhcp4 is always no for oob
      dhcp4: no
      match:
        # the mac address of the oob
        macaddress: xx:xx:xx:xx:xx:xx
      set-name: oob
  bridges:
    br-dpu:
      interfaces: [oob]
      critical: true
      # other bridge configurations
  # uncomment the line below to use NetworkManager as backend
renderer
  # renderer: NetworkManager
  version: 2
```

In addition to the above configuration, an IP address and other settings are required to make br-dpu work. You can either set them statically or configure them through DHCP.

Option 1: Set br-dpu through DHCP

To let the `br-dpu` get its configuration via DHCP, add `dhcp4: yes` like the following:

```
network:
  ethernets:
    oob:
      # dhcp4 is always no for oob
      dhcp4: no
      match:
        # the mac address of the oob
        macaddress: xx:xx:xx:xx:xx:xx
      set-name: oob
  bridges:
    br-dpu:
      interfaces: [oob]
      # the mac address of the oob
      macaddress: xx:xx:xx:xx:xx:xx
      critical: true
      dhcp4: yes
      dhcp4-overrides:
        route-metric: 5
  version: 2
```

Note

Please ensure that the configuration on `br-dpu` matches the configuration on your DHCP server. You may need additional configurations to make the DHCP procedure work. For example: 1. Set `macaddress` to explicitly assign a MAC address to `br-dpu`. 2. Set `dhcp-identifier: mac` to use MAC address as the DHCP client identifier.

Option 2: Set br-dpu Statically

If DHCP is disabled on br-dpu, you must assign an IP address and a default gateway to `br-dpu`. `nameservers` are optional.

```
network:
  ethernets:
    oob:
      #dhcp4 is always no for oob
      dhcp4: no
      match:
        # the mac address of the oob
        macaddress: xx:xx:xx:xx:xx:xx
      set-name: oob
  bridges:
    br-dpu:
      interfaces: [oob]
      critical: true
      # the addresses of the bridge
      addresses: ["xx.xx.xx.xx"]
      routes:
        # the default gateway for the bridge
        - to: default
          via: "xx.xx.xx.xx"
          metric: 5
      # DNS nameservers for the bridge
      nameservers:
        addresses:
          - xx.xx.xx.xx
        search:
          - "xxx"
  version: 2
```

> **Note**
>
> The metric of the default route must be at least 2.

Host Routes and network acceleration

When the hosts are connected both to the management network via the OOB interface, and to the high-speed network via the Bluefield PFs or VFs, the default route should be configured via the OOB. This means that the outgoing traffic will by default go through the management network.

This is an issue in case some external traffic is reaching the host via the high-speed IP address. This causes asymmetric traffic routing. In addition, if the outgoing traffic is via the OOB port instead of high-speed port, the performance are far worse since the traffic is not accelerated. Similarly, traffic from high-speed network reaching the host via the OOB network will cause asymmetric traffic. To fix this, some source-based routing on the host is necessary, so that traffic coming to the high speed network will leave the host via the high speed network, and traffic coming from the OOB interface will also leave the host via the OOB port.

The script below demonstrates an example of source-based routing to ensure proper network acceleration. It assumes that :

- 192.168.0.10/24 is the host IP on the management network (OOB interface) and the gateway is 192.168.0.1
- 10.0.0.10/22 is the host IP address on the high speed network (interface ens1f0np0) and the gateway is 10.0.0.1

```
# change routing table for traffic to high speed network from host
OOB to use OOB port
ip rule add priority 200 from 192.168.0.10/32 to 10.0.0.0/22 table 200
# Route via OOB Gateway
ip r a default via 192.168.0.1 dev br-dpu table 200

# change routing table for traffic to anywhere from host high-
speed IP
ip rule add priority 201 from 10.0.0.10/32 table 201
# route via the high speed gateway
ip r a default via 10.0.0.1 dev ens1f0np0 table 201
```

Host Trusted Use Cases

Host Trusted Use Cases

The **Host Trusted** scenario supports the following use cases, each tailored to specific networking and acceleration needs. Each use case may require specific configurations on the DPU for optimal performance.

- Host Based Networking and OVN Kubernetes
- OVN Kubernetes
- Host Based Networking
- DPU Passthrough in DPF Host Trusted
- Host Based Networking and SNAP Storage
- Host Based Networking and OVN Kubernetes and SNAP Storage

DPU Passthrough in DPF Host Trusted

> **Note**
>
> Follow this guide from the source GitHub repo at github.com/NVIDIA/doca-platform and moving to the `docs/public/user-guides/host-trusted/use-cases/passthrough/README.md` for better formatting of the code.

This configuration provides instructions for deploying the NVIDIA DOCA Platform Framework (DPF) on high-performance, bare-metal infrastructure in Host Trusted mode. It focuses on provisioning NVIDIA® BlueField®-3 DPUs using DPF and enabling them to act as passthrough devices.

Prerequisites

This guide should be run by cloning the repo from github.com/NVIDIA/doca-platform and moving to the `docs/public/user-guides/host-trusted/use-cases/passthrough/README.md` directory.

The system is set up as described in the system prerequisites.

Software Prerequisites

The following tools must be installed on the machine where the commands contained in this guide run:

- kubectl
- helm
- envsubst

Installation Guide

0. Required Variables

The following variables are required by this guide. A sensible default is provided where it makes sense, but many will be specific to the target infrastructure.

Commands in this guide are run in the same directory that contains this readme.

Environment variables file

```
## Virtual IP used by the load balancer for the DPU Cluster. Must
be a reserved IP from the management subnet and not allocated by
DHCP.
export DPUCLUSTER_VIP=

## Interface on which the DPUCluster load balancer will listen.
Should be the management interface of the control plane node.
export DPUCLUSTER_INTERFACE=

## IP address to the NFS server used as storage for the BFB.
export NFS_SERVER_IP=

## The DPF REGISTRY is the Helm repository URL where the DPF
Operator Chart resides.
## Usually this is the NVIDIA Helm NGC registry. For development
purposes, this can be set to a different repository.
export REGISTRY=https://helm.ngc.nvidia.com/nvidia/doca

## The DPF TAG is the version of the DPF components which will be
deployed in this guide.
export TAG=v25.10.0

## URL to the BFB used in the `bfb.yaml` and linked by the
DPUSet.
export BFB_URL="https://content.mellanox.com/BlueField/BFBs/Ubuntu24.04/bf-bundle-3.2.1-
34_25.11_ubuntu-24.04_64k_prod.bfb"
```

Modify the variables in `manifests/00-env-vars/envvars.env` to fit your environment, then source the file:

```
source manifests/00-env-vars/envvars.env
```

1. DPF Operator Installation

Create storage required by the DPF Operator

A number of environment variables must be set before running this command.

```
kubectl create ns dpf-operator-system
cat manifests/01-dpf-operator-installation/*.yaml | envsubst |
kubectl apply -f -
```

This deploys the following objects:

PersistentVolume and PersistentVolumeClaim for the provisioning controller

```
---
apiVersion: v1
kind: PersistentVolume
metadata:
  name: bfb-pv
spec:
  capacity:
    storage: 10Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteMany
  nfs:
    path: /mnt/dpf_share/bfb
    server: $NFS_SERVER_IP
  persistentVolumeReclaimPolicy: Delete
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: bfb-pvc
  namespace: dpf-operator-system
spec:
  accessModes:
  - ReadWriteMany
  resources:
    requests:
      storage: 10Gi
  volumeMode: Filesystem
  storageClassName: ""
```

Additional Dependencies

Before deploying the DPF Operator, ensure that Helm is properly configured according to the Helm prerequisites.

> **Warning**
>
> This is a critical prerequisite step that must be completed for the DPF Operator to function properly.

Deploy the DPF Operator

A number of environment variables must be set before running this command.

HTTP Registry (default)

If the $REGISTRY is an HTTP Registry (default value) use this command:

```
helm repo add --force-update dpf-repository ${REGISTRY}
helm repo update
helm upgrade --install -n dpf-operator-system dpf-operator dpf-
repository/dpf-operator --version=$TAG
```

OCI Registry

For development purposes, if the $REGISTRY is an OCI Registry use this command:

```
helm upgrade --install -n dpf-operator-system dpf-operator
$REGISTRY/dpf-operator --version=$TAG
```

Verification

These verification commands may need to be run multiple times to ensure the condition is met.

Verify the DPF Operator installation with:

```
## Ensure the DPF Operator deployment is available.
kubectl rollout status deployment --namespace dpf-operator-system
dpf-operator-controller-manager
## Ensure all pods in the DPF Operator system are ready.
kubectl wait --for=condition=ready --namespace dpf-operator-system
pods --all
```

2. DPF System Installation

This section involves creating the DPF system components and some basic infrastructure required for a functioning DPF-enabled cluster.

Deploy the DPF System components

A number of environment variables must be set before running this command.

```
kubectl create ns dpu-cplane-tenant1
cat manifests/02-dpf-system-installation/*.yaml | envsubst |
kubectl apply -f -
```

This will create the following objects:

DPFOperatorConfig to install the DPF System components

```yaml
---
apiVersion: operator.dpu.nvidia.com/v1alpha1
kind: DPFOperatorConfig
metadata:
  name: dpfoperatorconfig
  namespace: dpf-operator-system
spec:
  provisioningController:
    bfbPVCName: "bfb-pvc"
    dmsTimeout: 900
  kamajiClusterManager:
    disable: false
```

DPUCluster to serve as Kubernetes control plane for DPU nodes

```
---
apiVersion: provisioning.dpu.nvidia.com/v1alpha1
kind: DPUCluster
metadata:
  name: dpu-cplane-tenant1
  namespace: dpu-cplane-tenant1
spec:
  type: kamaji
  maxNodes: 10
  clusterEndpoint:
    # deploy keepalived instances on the nodes that match the
given nodeSelector.
    keepalived:
      # interface on which keepalived will listen. Should be the oob
interface of the control plane node.
      interface: $DPUCLUSTER_INTERFACE
      # Virtual IP reserved for the DPU Cluster load balancer.
Must not be allocatable by DHCP.
      vip: $DPUCLUSTER_VIP
      # virtualRouterID must be in range [1,255], make sure the
given virtualRouterID does not duplicate with any existing
keepalived process running on the host
      virtualRouterID: 126
      nodeSelector:
        node-role.kubernetes.io/control-plane: ""
```

Verification

These verification commands may need to be run multiple times to ensure the condition is met.

Verify the DPF System with:

```
## Ensure the provisioning and DPUService controller manager
deployments are available.
kubectl rollout status deployment --namespace dpf-operator-system
dpf-provisioning-controller-manager dpuservice-controller-manager
## Ensure all other deployments in the DPF Operator system are
Available.
kubectl rollout status deployment --namespace dpf-operator-system
## Ensure the DPUCluster is ready for nodes to join.
kubectl wait --for=condition=ready --namespace dpu-cplane-tenant1
dpucluster --all
```

3. DPU Provisioning and Interface Plumbing

> ⚠ **Warning**
>
> In case more than 1 DPU exists per node, the relevant selector should be applied in the DPUSet to select the appropriate DPU. See DPUSet - DPU Selection to understand more about the selectors.

In this step we provision our DPUs and we do the nessecary interface plumbing to enable the DPU to act as a passthrough device.

The user is expected to create a DPUSet object to provision the DPUs and a DPUServiceChain to enable the nessecary connectivity between the host and DPU interfaces.

> Check the DPUSet documentation and DPUServiceChain documentation for more information about these objects.

Create the BFB, DPUSet and DPUServiceChain

A number of environment variables must be set before running this command.

```
cat manifests/03-dpf-object-installation/*.yaml | envsubst |
kubectl apply -f -
```

This will deploy the following objects:

BFB to download the BFB to a shared volume

```
---
apiVersion: provisioning.dpu.nvidia.com/v1alpha1
kind: BFB
metadata:
  name: bf-bundle-$TAG
  namespace: dpf-operator-system
spec:
  url: $BFB_URL
```

DPUFlavor used for provisioning the DPUs

```
---
apiVersion: provisioning.dpu.nvidia.com/v1alpha1
kind: DPUFlavor
metadata:
  name: passthrough-$TAG
  namespace: dpf-operator-system
spec:
  grub:
    kernelParameters:
    - console=hvc0
    - console=ttyAMA0
    - earlycon=pl011,0x13010000
    - fixrttc
    - net.ifnames=0
    - biosdevname=0
    - iommu.passthrough=1
    - cgroup_no_v1=net_prio,net_cls
    - hugepagesz=2048kB
    - hugepages=3072
  nvconfig:
  - device: "*"
    parameters:
    - PF_BAR2_ENABLE=0
    - PER_PF_NUM_SF=1
    - PF_TOTAL_SF=20
    - PF_SF_BAR_SIZE=10
    - NUM_PF_MSIX_VALID=0
    - PF_NUM_PF_MSIX_VALID=1
    - PF_NUM_PF_MSIX=228
    - INTERNAL_CPU_MODEL=1
    - INTERNAL_CPU_OFFLOAD_ENGINE=0
    - SRIOV_EN=1
    - NUM_OF_VFS=46
    - LAG_RESOURCE_ALLOCATION=1
    - LINK_TYPE_P1=ETH
```

```
    - LINK_TYPE_P2=ETH
  ovs:
    rawConfigScript: |
      _ovs-vsctl() {
        ovs-vsctl --no-wait --timeout 15 "$@"
      }

      _ovs-vsctl set Open_vSwitch . other_config:doca-init=true
      _ovs-vsctl set Open_vSwitch . other_config:dpdk-max-
memzones=50000
      _ovs-vsctl set Open_vSwitch . other_config:hw-offload=true
      _ovs-vsctl set Open_vSwitch . other_config:pmd-quiet-
idle=true
      _ovs-vsctl set Open_vSwitch . other_config:max-idle=20000
      _ovs-vsctl set Open_vSwitch . other_config:max-
revalidator=5000
      _ovs-vsctl set Open_vSwitch . other_config:ctl-pipe-
size=1024
      _ovs-vsctl --if-exists del-br ovsbr1
      _ovs-vsctl --if-exists del-br ovsbr2
      _ovs-vsctl --may-exist add-br br-sfc
      _ovs-vsctl set bridge br-sfc datapath_type=netdev
      _ovs-vsctl set bridge br-sfc fail_mode=secure
      _ovs-vsctl --may-exist add-port br-sfc p0
      _ovs-vsctl set Interface p0 type=dpdk
      _ovs-vsctl set Interface p0 mtu_request=9216
      _ovs-vsctl set Port p0 external_ids:dpf-type=physical

  bfcfgParameters:
  - UPDATE_ATF_UEFI=yes
  - UPDATE_DPU_OS=yes
  - WITH_NIC_FW_UPDATE=yes

  configFiles:
  - path: /etc/mellanox/mlnx-bf.conf
    operation: override
```

```
    raw: |
        ALLOW_SHARED_RQ="no"
        IPSEC_FULL_OFFLOAD="no"
        ENABLE_ESWITCH_MULTIPORT="yes"
    permissions: "0644"
  - path: /etc/mellanox/mlnx-ovs.conf
    operation: override
    raw: |
        CREATE_OVS_BRIDGES="no"
        OVS_DOCA="yes"
    permissions: "0644"
  - path: /etc/mellanox/mlnx-sf.conf
    operation: override
    raw: ""
    permissions: "0644"
```

DPUSet to provision DPUs on worker nodes

```
---
apiVersion: provisioning.dpu.nvidia.com/v1alpha1
kind: DPUSet
metadata:
  name: passthrough
  namespace: dpf-operator-system
spec:
  dpuNodeSelector:
    matchLabels:
      feature.node.kubernetes.io/dpu-enabled: "true"
  dpuTemplate:
    spec:
      dpuFlavor: passthrough-$TAG
      bfb:
        name: bf-bundle-$TAG
```

DPUServiceInterfaces used by the DPUServiceChain

```yaml
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceInterface
metadata:
  name: p0
  namespace: dpf-operator-system
spec:
  template:
    spec:
      template:
        metadata:
          labels:
            interface: "p0"
        spec:
          interfaceType: physical
          physical:
            interfaceName: p0
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceInterface
metadata:
  name: p1
  namespace: dpf-operator-system
spec:
  template:
    spec:
      template:
        metadata:
          labels:
            interface: "p1"
        spec:
          interfaceType: physical
          physical:
            interfaceName: p1
---
```

```yaml
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceInterface
metadata:
  name: pf0hpf
  namespace: dpf-operator-system
spec:
  template:
    spec:
      template:
        metadata:
          labels:
            interface: "pf0hpf"
        spec:
          interfaceType: pf
          pf:
            pfID: 0
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceInterface
metadata:
  name: pf1hpf
  namespace: dpf-operator-system
spec:
  template:
    spec:
      template:
        metadata:
          labels:
            interface: "pf1hpf"
        spec:
          interfaceType: pf
          pf:
            pfID: 1
```

DPUServiceChain to make the device act as passthrough device

```yaml
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceChain
metadata:
  name: passthrough
  namespace: dpf-operator-system
spec:
  template:
    spec:
      template:
        spec:
          switches:
            - ports:
              - serviceInterface:
                  matchLabels:
                    interface: p0
              - serviceInterface:
                  matchLabels:
                    interface: pf0hpf
            - ports:
              - serviceInterface:
                  matchLabels:
                    interface: p1
              - serviceInterface:
                  matchLabels:
                    interface: pf1hpf
```

Verification

The following verification commands may need to be run multiple times to ensure the condition is met.

```
## Ensure the DPUServiceChain is ready
kubectl wait --for=condition=ready --namespace dpf-operator-system
dpuservicechain passthrough
## Ensure the DPUServiceInterfaces are ready
kubectl wait --for=condition=ready --namespace dpf-operator-system
dpuserviceinterface p0 p1 pf0hpf pf1hpf
## Ensure the BFB is ready
kubectl wait --for="jsonpath={.status.phase}=Ready" --namespace dpf-operator-
system bfb bf-bundle
## Ensure the DPUs have the condition Initialized (this may take
time)
kubectl wait --for=condition=Initialized --namespace dpf-operator-
system dpu --all
```

Wait for the DPUs to be provisioned

The DPUs will take some time to be provisioned and the OS to be installed. You can check the status of the DPUs with

```
kubectl wait --for=condition=Ready --namespace dpf-operator-system
dpu --all
```

or with `dpfctl`:

```
$ kubectl -n dpf-operator-system exec deploy/dpf-operator-
controller-manager -- /dpfctl describe dpusets
NAME                                                NAMESPACE
STATUS       REASON         SINCE  MESSAGE
DPFOperatorConfig/dpfoperatorconfig                 dpf-operator-
system  Ready: True  Success        24m
  DPUServiceChains
    DPUServiceChain/passthrough                     dpf-operator-
system  Ready: True  Success        7s
  DPUServiceInterfaces
    4 DPUServiceInterfaces...                       dpf-operator-
system  Ready: True  Success        78m    See p0, p1, pf0hpf,
pf1hpf
  DPUSets
    DPUSet/passthrough                              dpf-operator-
system
      BFB/bf-bundle                                 dpf-operator-
system  Ready: True  Ready          78m    File: bf-bundle-3.2.1-
34_25.11_ubuntu-24.04_64k_prod.bfb, DOCA: 3.2.1
      DPUs
        DPU/dpu-node-mt2306xz0370-mt2306xz0370  dpf-operator-
system
                      Ready
False        OS Installing  1s
        DPU/dpu-node-mt2333xz0xq3-mt2333xz0xq3  dpf-operator-
system
                      Ready
False        OS Installing  1s
        DPU/dpu-node-mt2402xz0f6v-mt2402xz0f6v  dpf-operator-
system
                      Ready
False        OS Installing  1s
        DPU/dpu-node-mt2404xz0c98-mt2404xz0c98  dpf-operator-
system
```

```
                        Ready
False        OS Installing  1s
```

Test Traffic

After the DPUs are provisioned and the rest of the objects are Ready, we can test traffic by assigning an IP on one of the PFs on the host for each DPU, and run a simple ping. This assumes that the high speed ports of the DPUs are connected and the DPUs can reach each other. Assuming the PF is named `ens0f0np0` then:

Host 1:

```
ip addr add 192.168.1.1/24 dev ens0f0np0
```

Host 2:

```
ip addr add 192.168.1.2/24 dev ens0f0np0
```

From Host 1:

```
$ ping 192.168.1.2 -c3
PING 192.168.1.2 (192.168.1.2) 56(84) bytes of data.
64 bytes from 192.168.1.2: icmp_seq=1 ttl=64 time=0.387 ms
64 bytes from 192.168.1.2: icmp_seq=2 ttl=64 time=0.344 ms
64 bytes from 192.168.1.2: icmp_seq=3 ttl=64 time=0.396 ms

--- 192.168.1.2 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2053ms
rtt min/avg/max/mdev = 0.344/0.375/0.396/0.022 ms
```

Uninstall

This section covers only the DPF related components and not the prerequisites as these must be managed by the admin.

Delete the DPF Operator system and DPF Operator

```
kubectl delete -n dpf-operator-system dpfoperatorconfig
dpfoperatorconfig --wait
helm uninstall -n dpf-operator-system dpf-operator --wait
```

Delete DPF Operator PVC

```
kubectl -n dpf-operator-system delete pvc bfb-pvc
kubectl delete pv bfb-pv
```

Note: there can be a race condition with deleting the underlying Kamaji cluster which runs the DPU cluster control plane in this guide. If that happens it may be necessary to remove finalizers manually from `DPUCluster` and `Datastore` objects.

Host Based Networking

> **Note**
>
> Follow this guide from the source GitHub repo at github.com/NVIDIA/doca-platform and moving to the `docs/public/user-guides/host-trusted/use-cases/hbn/README.md` for better formatting of the code.

In this configuration NVIDIA Host Based Networking (HBN) is installed as a DPUService.

Prerequisites

This guide should be run by cloning the repo from github.com/NVIDIA/doca-platform and moving to the `docs/public/user-guides/hbn_only` directory.

The system is set up as described in the system prerequisites. The HBN DPUService has the additional requirements:

Software Prerequisites

This guide uses the following tools which must be installed on the machine where the commands contained in this guide run.

- kubectl
- helm
- envsubst

Kubernetes Prerequisites

- Control plane setup is complete before starting this guide
- CNI installed before starting this guide
- Worker nodes are not added until indicated by this guide

- High-speed ports are used for secondary workload network and not for primary CNI

Virtual Functions

A number of virtual functions (VFs) will be created on hosts when provisioning DPUs. Certain of these VFs are marked for specific usage:

- The first VF (vf0) is used by provisioning components.
- The remaining VFs are allocated by SR-IOV Device Plugin.

Installation Guide

0. Required Variables

The following variables are required by this guide. A sensible default is provided where it makes sense, but many will be specific to the target infrastructure.

Commands in this guide are run in the same directory that contains this readme.

Environment variables file

```
## Virtual IP used by the load balancer for the DPU Cluster. Must
be a reserved IP from the management subnet and not allocated by
DHCP.
export DPUCLUSTER_VIP=

## DPU_P0 is the name of the first port of the DPU. This name
must be the same on all worker nodes.
export DPU_P0=

## Interface on which the DPUCluster load balancer will listen.
Should be the management interface of the control plane node.
export DPUCLUSTER_INTERFACE=

## IP address to the NFS server used as storage for the BFB.
export NFS_SERVER_IP=

## The repository URL for the NVIDIA Helm chart registry.
## Usually this is the NVIDIA Helm NGC registry. For development
purposes, this can be set to a different repository.
export HELM_REGISTRY_REPO_URL=https://helm.ngc.nvidia.com/nvidia/doca

## The repository URL for the HBN container image.
## Usually this is the NVIDIA NGC registry. For development
purposes, this can be set to a different repository.
export HBN_NGC_IMAGE_URL=nvcr.io/nvidia/doca/doca_hbn

## The DPF REGISTRY is the Helm repository URL where the DPF
Operator Chart resides.
## Usually this is the NVIDIA Helm NGC registry. For development
purposes, this can be set to a different repository.
export REGISTRY=https://helm.ngc.nvidia.com/nvidia/doca

## The DPF TAG is the version of the DPF components which will be
deployed in this guide.
export TAG=v25.10.0
```

```
## URL to the BFB used in the `bfb.yaml` and linked by the
DPUSet.
export BFB_URL="https://content.mellanox.com/BlueField/BFBs/Ubuntu24.04/bf-bundle-3.2.1-
34_25.11_ubuntu-24.04_64k_prod.bfb"
```

Modify the variables in `manifests/00-env-vars/envvars.env` to fit your environment, then source the file:

```
source manifests/00-env-vars/envvars.env
```

1. DPF Operator Installation

Create storage required by the DPF Operator

A number of environment variables must be set before running this command.

```
kubectl create ns dpf-operator-system
cat manifests/01-dpf-operator-installation/*.yaml | envsubst |
kubectl apply -f -
```

This deploys the following objects:

PersistentVolume and PersistentVolumeClaim for the provisioning controller

```
---
apiVersion: v1
kind: PersistentVolume
metadata:
  name: bfb-pv
spec:
  capacity:
    storage: 10Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteMany
  nfs:
    path: /mnt/dpf_share/bfb
    server: $NFS_SERVER_IP
  persistentVolumeReclaimPolicy: Delete
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: bfb-pvc
  namespace: dpf-operator-system
spec:
  accessModes:
  - ReadWriteMany
  resources:
    requests:
      storage: 10Gi
  volumeMode: Filesystem
  storageClassName: ""
```

Additional Dependencies

Before deploying the DPF Operator, ensure that Helm is properly configured according to the Helm prerequisites.

> **Warning**
>
> This is a critical prerequisite step that must be completed for the DPF Operator to function properly.

Deploy the DPF Operator

A number of environment variables must be set before running this command.

HTTP Registry (default)

If the $REGISTRY is an HTTP Registry (default value) use this command:

```
helm repo add --force-update dpf-repository ${REGISTRY}
helm repo update
helm upgrade --install -n dpf-operator-system dpf-operator dpf-
repository/dpf-operator --version=$TAG
```

OCI Registry

For development purposes, if the $REGISTRY is an OCI Registry use this command:

```
helm upgrade --install -n dpf-operator-system dpf-operator
$REGISTRY/dpf-operator --version=$TAG
```

Verification

These verification commands may need to be run multiple times to ensure the condition is met.

Verify the DPF Operator installation with:

```
## Ensure the DPF Operator deployment is available.
kubectl rollout status deployment --namespace dpf-operator-system
dpf-operator-controller-manager
## Ensure all pods in the DPF Operator system are ready.
kubectl wait --for=condition=ready --namespace dpf-operator-system
pods --all
```

2. DPF system installation

This section involves creating the DPF system components and some basic infrastructure required for a functioning DPF-enabled cluster.

Deploy the DPF System components

A number of environment variables must be set before running this command.

```
kubectl create ns dpu-cplane-tenant1
cat manifests/02-dpf-system-installation/*.yaml | envsubst |
kubectl apply -f -
```

This will create the following objects:

DPFOperatorConfig to install the DPF System components

```yaml
---
apiVersion: operator.dpu.nvidia.com/v1alpha1
kind: DPFOperatorConfig
metadata:
  name: dpfoperatorconfig
  namespace: dpf-operator-system
spec:
  provisioningController:
    bfbPVCName: "bfb-pvc"
    dmsTimeout: 900
  kamajiClusterManager:
    disable: false
```

DPUCluster to serve as Kubernetes control plane for DPU nodes

```
---
apiVersion: provisioning.dpu.nvidia.com/v1alpha1
kind: DPUCluster
metadata:
  name: dpu-cplane-tenant1
  namespace: dpu-cplane-tenant1
spec:
  type: kamaji
  maxNodes: 10
  clusterEndpoint:
    # deploy keepalived instances on the nodes that match the
given nodeSelector.
    keepalived:
      # interface on which keepalived will listen. Should be the oob
interface of the control plane node.
      interface: $DPUCLUSTER_INTERFACE
      # Virtual IP reserved for the DPU Cluster load balancer.
Must not be allocatable by DHCP.
      vip: $DPUCLUSTER_VIP
      # virtualRouterID must be in range [1,255], make sure the
given virtualRouterID does not duplicate with any existing
keepalived process running on the host
      virtualRouterID: 126
      nodeSelector:
        node-role.kubernetes.io/control-plane: ""
```

Verification

These verification commands may need to be run multiple times to ensure the condition is met.

Verify the DPF System with:

```
## Ensure the provisioning and DPUService controller manager
deployments are available.
kubectl rollout status deployment --namespace dpf-operator-system
dpf-provisioning-controller-manager dpuservice-controller-manager
## Ensure all other deployments in the DPF Operator system are
Available.
kubectl rollout status deployment --namespace dpf-operator-system
## Ensure the DPUCluster is ready for nodes to join.
kubectl wait --for=condition=ready --namespace dpu-cplane-tenant1
dpucluster --all
```

3. Enable Accelerated Interfaces

Traffic can be routed through HBN on the worker node by mounting the DPU physical interface into a pod.

Install Multus and SRIOV Network Operator using NVIDIA Network Operator

```
helm repo add nvidia https://helm.ngc.nvidia.com/nvidia --force-update
helm upgrade --no-hooks --install --create-namespace --namespace
nvidia-network-operator network-operator nvidia/network-operator
--version 25.7.0 -f ./manifests/03-enable-accelerated-
interfaces/helm-values/network-operator.yml
```

NVIDIA Network Operator Helm values

```yaml
nfd:
  enabled: false
  deployNodeFeatureRules: false
sriovNetworkOperator:
  enabled: true
sriov-network-operator:
  operator:
    affinity:
      nodeAffinity:
        requiredDuringSchedulingIgnoredDuringExecution:
          nodeSelectorTerms:
            - matchExpressions:
                - key: node-role.kubernetes.io/master
                  operator: Exists
            - matchExpressions:
                - key: node-role.kubernetes.io/control-plane
                  operator: Exists
  crds:
    enabled: true
  sriovOperatorConfig:
    deploy: true
    configDaemonNodeSelector: null
operator:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
          - matchExpressions:
              - key: node-role.kubernetes.io/master
                operator: Exists
          - matchExpressions:
              - key: node-role.kubernetes.io/control-plane
                operator: Exists
```

Apply the NICClusterConfiguration and SriovNetworkNodePolicy

```
cat manifests/03-enable-accelerated-interfaces/*.yaml | envsubst
| kubectl apply -f -
```

This will deploy the following objects:

NICClusterPolicy for the NVIDIA Network Operator

```
---
apiVersion: mellanox.com/v1alpha1
kind: NicClusterPolicy
metadata:
  name: nic-cluster-policy
spec:
  secondaryNetwork:
    multus:
      image: multus-cni
      imagePullSecrets: []
      repository: ghcr.io/k8snetworkplumbingwg
      version: v3.9.3
```

SriovNetworkNodePolicy for the SR-IOV Network Operator

```
---
apiVersion: sriovnetwork.openshift.io/v1
kind: SriovNetworkNodePolicy
metadata:
  name: bf3-p0-vfs
  namespace: nvidia-network-operator
spec:
  nicSelector:
    deviceID: "a2dc"
    vendor: "15b3"
    pfNames:
    - $DPU_P0#2-45
  nodeSelector:
    node-role.kubernetes.io/worker: ""
  numVfs: 46
  resourceName: bf3-p0-vfs
  isRdma: true
  externallyManaged: true
  deviceType: netdevice
  linkType: eth
```

Verification

These verification commands may need to be run multiple times to ensure the condition is met.

Verify the DPF System with:

```
## Ensure the provisioning and DPUService controller manager
deployments are available.
kubectl wait --for=condition=Ready --namespace nvidia-network-
operator pods --all
## Expect the following Daemonsets to be successfully rolled out.
kubectl rollout status daemonset --namespace nvidia-network-
operator kube-multus-ds sriov-network-config-daemon sriov-device-
plugin
```

4. DPU Provisioning and Service Installation

In this step we deploy our DPUs and the services that will run on them.

The user is expected to create a DPUDeployment object that reflects a set of DPUServices that should run on a set of DPUs.

> If you want to learn more about `DPUDeployments`, feel free to check the DPUDeployment documentation.

Create the DPUDeployment, DPUServiceConfig, DPUServiceTemplate and other necessary objects

> ⚠ **Warning**
>
> In case more than 1 DPU exists per node, the relevant selector should be applied in the DPUDeployment to select the appropriate DPU. See DPUDeployment - DPUs Configuration to understand more about the selectors.

A number of environment variables must be set before running this command.

```
cat manifests/04-dpudeployment-installation/*.yaml | envsubst |
kubectl apply -f -
```

This will deploy the following objects:

BFB to download Bluefield Bitstream to a shared volume

```
---
apiVersion: provisioning.dpu.nvidia.com/v1alpha1
kind: BFB
metadata:
  name: bf-bundle-$TAG
  namespace: dpf-operator-system
spec:
  url: $BFB_URL
```

HBN DPUFlavor to correctly configure the DPUs on provisioning

```
---
apiVersion: provisioning.dpu.nvidia.com/v1alpha1
kind: DPUFlavor
metadata:
  name: hbn-$TAG
  namespace: dpf-operator-system
spec:
  bfcfgParameters:
  - UPDATE_ATF_UEFI=yes
  - UPDATE_DPU_OS=yes
  - WITH_NIC_FW_UPDATE=yes
  configFiles:
  - operation: override
    path: /etc/mellanox/mlnx-bf.conf
    permissions: "0644"
    raw: |
      ALLOW_SHARED_RQ="no"
      IPSEC_FULL_OFFLOAD="no"
      ENABLE_ESWITCH_MULTIPORT="yes"
  - operation: override
    path: /etc/mellanox/mlnx-ovs.conf
    permissions: "0644"
    raw: |
      CREATE_OVS_BRIDGES="no"
      OVS_DOCA="yes"
  - operation: override
    path: /etc/mellanox/mlnx-sf.conf
    permissions: "0644"
    raw: ""
  grub:
    kernelParameters:
    - console=hvc0
    - console=ttyAMA0
    - earlycon=pl011,0x13010000
    - fixrttc
```

```
    - net.ifnames=0
    - biosdevname=0
    - iommu.passthrough=1
    - cgroup_no_v1=net_prio,net_cls
    - hugepagesz=2048kB
    - hugepages=3072
  nvconfig:
  - device: '*'
    parameters:
    - PF_BAR2_ENABLE=0
    - PER_PF_NUM_SF=1
    - PF_TOTAL_SF=20
    - PF_SF_BAR_SIZE=10
    - NUM_PF_MSIX_VALID=0
    - PF_NUM_PF_MSIX_VALID=1
    - PF_NUM_PF_MSIX=228
    - INTERNAL_CPU_MODEL=1
    - INTERNAL_CPU_OFFLOAD_ENGINE=0
    - SRIOV_EN=1
    - NUM_OF_VFS=46
    - LAG_RESOURCE_ALLOCATION=1
    - LINK_TYPE_P1=ETH
    - LINK_TYPE_P2=ETH
  ovs:
    rawConfigScript: |
      _ovs-vsctl() {
        ovs-vsctl --no-wait --timeout 15 "$@"
      }

      _ovs-vsctl set Open_vSwitch . other_config:doca-init=true
      _ovs-vsctl set Open_vSwitch . other_config:dpdk-max-
memzones=50000
      _ovs-vsctl set Open_vSwitch . other_config:hw-offload=true
      _ovs-vsctl set Open_vSwitch . other_config:pmd-quiet-
idle=true
      _ovs-vsctl set Open_vSwitch . other_config:max-idle=20000
```

```
      _ovs-vsctl set Open_vSwitch . other_config:max-
revalidator=5000
      _ovs-vsctl --if-exists del-br ovsbr1
      _ovs-vsctl --if-exists del-br ovsbr2
      _ovs-vsctl --may-exist add-br br-sfc
      _ovs-vsctl set bridge br-sfc datapath_type=netdev
      _ovs-vsctl set bridge br-sfc fail_mode=secure
      _ovs-vsctl --may-exist add-port br-sfc p0
      _ovs-vsctl set Interface p0 type=dpdk
      _ovs-vsctl set Interface p0 mtu_request=9216
      _ovs-vsctl set Port p0 external_ids:dpf-type=physical
      _ovs-vsctl --may-exist add-br br-hbn
      _ovs-vsctl set bridge br-hbn datapath_type=netdev
      _ovs-vsctl set bridge br-hbn fail_mode=secure
```

DPUDeployment to provision DPUs on worker nodes

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUDeployment
metadata:
  name: hbn-only
  namespace: dpf-operator-system
spec:
  dpus:
    bfb: bf-bundle-$TAG
    flavor: hbn-$TAG
    dpuSets:
    - nameSuffix: "dpuset1"
      nodeSelector:
        matchLabels:
          feature.node.kubernetes.io/dpu-enabled: "true"
      dpuSelector:
        provisioning.dpu.nvidia.com/dpudevice-pf0-name: $DPU_P0
  services:
    doca-hbn:
      serviceTemplate: doca-hbn
      serviceConfiguration: doca-hbn
  serviceChains:
    switches:
      - ports:
        - serviceInterface:
            matchLabels:
              uplink: p0
        - service:
            name: doca-hbn
            interface: p0_if
      - ports:
        - serviceInterface:
            matchLabels:
              uplink: p1
        - service:
```

```
            name: doca-hbn
            interface: p1_if
        - ports:
          - serviceInterface:
              matchLabels:
                vf: pf0vf10
          - service:
              name: doca-hbn
              interface: pf0vf10_if
        - ports:
          - serviceInterface:
              matchLabels:
                vf: pf1vf10
          - service:
              name: doca-hbn
              interface: pf1vf10_if
```

DPUServiceConfig and DPUServiceTemplate to deploy HBN workloads to the DPUs

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceConfiguration
metadata:
  name: doca-hbn
  namespace: dpf-operator-system
spec:
  deploymentServiceName: "doca-hbn"
  serviceConfiguration:
    serviceDaemonSet:
      annotations:
        k8s.v1.cni.cncf.io/networks: |-
          [
          {"name": "iprequest", "interface": "ip_lo", "cni-args": {"poolNames": ["loopback"], "poolType": "cidrpool"}},
          {"name": "iprequest", "interface": "ip_pf0vf10", "cni-args": {"poolNames": ["pool1"], "poolType": "cidrpool", "allocateDefaultGateway": true}},
          {"name": "iprequest", "interface": "ip_pf1vf10", "cni-args": {"poolNames": ["pool2"], "poolType": "cidrpool", "allocateDefaultGateway": true}}
          ]
    helmChart:
      values:
        configuration:
          perDPUValuesYAML: |
            - hostnamePattern: "*"
              values:
                bgp_peer_group: hbn
                vrf1: RED
                vrf2: BLUE
                l3vni1: 100001
                l3vni2: 100002
            - hostnamePattern: "worker1*"
              values:
                bgp_autonomous_system: 65101
            - hostnamePattern: "worker2*"
```

```
            values:
              bgp_autonomous_system: 65201
        startupYAMLJ2: |
          - header:
              model: bluefield
              nvue-api-version: nvue_v1
              rev-id: 1.0
              version: HBN 2.4.0
          - set:
              evpn:
                enable: on
                route-advertise: {}
              interface:
                lo:
                  ip:
                    address:
                      {{ ipaddresses.ip_lo.ip }}/32: {}
                  type: loopback
                p0_if,p1_if,pf0vf10_if,pf1vf10_if:
                  type: swp
                  link:
                    mtu: 9000
                pf0vf10_if:
                  ip:
                    address:
                      {{ ipaddresses.ip_pf0vf10.cidr }}: {}
                    vrf: {{ config.vrf1 }}
                pf1vf10_if:
                  ip:
                    address:
                      {{ ipaddresses.ip_pf1vf10.cidr }}: {}
                    vrf: {{ config.vrf2 }}
              nve:
                vxlan:
                  arp-nd-suppress: on
                  enable: on
```

```
                    source:
                      address: {{ ipaddresses.ip_lo.ip }}
                router:
                  bgp:
                    enable: on
                    graceful-restart:
                      mode: full
                vrf:
                  default:
                    router:
                      bgp:
                        address-family:
                          ipv4-unicast:
                            enable: on
                            redistribute:
                              connected:
                                enable: on
                            multipaths:
                              ebgp: 16
                          l2vpn-evpn:
                            enable: on
                        autonomous-system: {{
config.bgp_autonomous_system }}
                        enable: on
                        neighbor:
                          p0_if:
                            peer-group: {{ config.bgp_peer_group
}}
                            type: unnumbered
                            address-family:
                              l2vpn-evpn:
                                enable: on
                                add-path-tx: off
                          p1_if:
                            peer-group: {{ config.bgp_peer_group
}}
```

```
                            type: unnumbered
                            address-family:
                              l2vpn-evpn:
                                enable: on
                                add-path-tx: off
                        path-selection:
                          multipath:
                            aspath-ignore: on
                        peer-group:
                          {{ config.bgp_peer_group }}:
                            address-family:
                              ipv4-unicast:
                                enable: on
                              l2vpn-evpn:
                                enable: on
                            remote-as: external
                        router-id: {{ ipaddresses.ip_lo.ip }}
                    {{ config.vrf1 }}:
                      evpn:
                        enable: on
                        vni:
                          {{ config.l3vni1 }}: {}
                      router:
                        bgp:
                          address-family:
                            ipv4-unicast:
                              enable: on
                              redistribute:
                                connected:
                                  enable: on
                              route-export:
                                to-evpn:
                                  enable: on
                          autonomous-system: {{
config.bgp_autonomous_system }}
                          enable: on
```

```
                    {{ config.vrf2 }}:
                      evpn:
                        enable: on
                        vni:
                          {{ config.l3vni2 }}: {}
                      router:
                        bgp:
                          address-family:
                            ipv4-unicast:
                              enable: on
                              redistribute:
                                connected:
                                  enable: on
                              route-export:
                                to-evpn:
                                  enable: on
                          autonomous-system: {{
config.bgp_autonomous_system }}
                          enable: on

  interfaces:
  - name: p0_if
    network: mybrhbn
  - name: p1_if
    network: mybrhbn
  - name: pf0vf10_if
    network: mybrhbn
  - name: pf1vf10_if
    network: mybrhbn
```

```yaml
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceTemplate
metadata:
  name: doca-hbn
  namespace: dpf-operator-system
spec:
  deploymentServiceName: "doca-hbn"
  helmChart:
    source:
      repoURL: $HELM_REGISTRY_REPO_URL
      version: 1.0.5
      chart: doca-hbn
    values:
      image:
        repository: $HBN_NGC_IMAGE_URL
        tag: 3.2.1-doca3.2.1
      resources:
        memory: 6Gi
        nvidia.com/bf_sf: 4
```

DPUServiceInterfaces for physical ports on the DPU

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceInterface
metadata:
  name: p0
  namespace: dpf-operator-system
spec:
  template:
    spec:
      template:
        metadata:
          labels:
            uplink: "p0"
        spec:
          interfaceType: physical
          physical:
            interfaceName: p0
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceInterface
metadata:
  name: p1
  namespace: dpf-operator-system
spec:
  template:
    spec:
      template:
        metadata:
          labels:
            uplink: "p1"
        spec:
          interfaceType: physical
          physical:
            interfaceName: p1
---
```

```yaml
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceInterface
metadata:
  name: pf0vf10-rep
  namespace: dpf-operator-system
spec:
  template:
    spec:
      template:
        metadata:
          labels:
            vf: "pf0vf10"
        spec:
          interfaceType: vf
          vf:
            parentInterfaceRef: p0
            pfID: 0
            vfID: 10
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceInterface
metadata:
  name: pf1vf10-rep
  namespace: dpf-operator-system
spec:
  template:
    spec:
      template:
        metadata:
          labels:
            vf: "pf1vf10"
        spec:
          interfaceType: vf
          vf:
            parentInterfaceRef: p1
            pfID: 1
```

```
            vfID: 10
```

DPUServiceIPAM to set up IP Address Management on the DPUCluster

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceIPAM
metadata:
  name: pool1
  namespace: dpf-operator-system
spec:
  ipv4Network:
    network: "10.0.121.0/24"
    gatewayIndex: 2
    prefixSize: 29
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceIPAM
metadata:
  name: pool2
  namespace: dpf-operator-system
spec:
  ipv4Network:
    network: "10.0.122.0/24"
    gatewayIndex: 2
    prefixSize: 29
```

DPUServiceIPAM for the loopback interface in HBN

```yaml
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceIPAM
metadata:
  name: loopback
  namespace: dpf-operator-system
spec:
  ipv4Network:
    network: "11.0.0.0/24"
    prefixSize: 32
```

Verification

These verification commands may need to be run multiple times to ensure the condition is met.

Note that the DPUService name will have a random suffix. For example, `doca-hbn-l2xsl`.

Verify the DPU and Service installation with:

```
## Ensure the DPUServices are created and have been reconciled.
kubectl wait --for=condition=ApplicationsReconciled --namespace
dpf-operator-system dpuservices -l svc.dpu.nvidia.com/owned-by-
dpudeployment=dpf-operator-system_hbn-only
## Ensure the DPUServiceIPAMs have been reconciled
kubectl wait --for=condition=DPUIPAMObjectReconciled --namespace
dpf-operator-system dpuserviceipam --all
## Ensure the DPUServiceInterfaces have been reconciled
kubectl wait --for=condition=ServiceInterfaceSetReconciled --
namespace dpf-operator-system dpuserviceinterface --all
## Ensure the DPUServiceChains have been reconciled
kubectl wait --for=condition=ServiceChainSetReconciled --namespace
dpf-operator-system dpuservicechain --all
```

5. Test Traffic

Add worker nodes to the cluster

At this point workers should be added to the cluster. Each worker node should be configured in line with the prerequisites. As workers are added to the cluster DPUs will be provisioned and DPUServices will begin to be spun up.

You can verify the status of the DPUDeployment and its components with the following command:

```
$ kubectl -n dpf-operator-system exec deploy/dpf-operator-
controller-manager -- /dpfctl describe dpudeployments
NAME                                          NAMESPACE
STATUS        REASON     SINCE  MESSAGE
DPFOperatorConfig/dpfoperatorconfig           dpf-operator-system
Ready: True   Success    2h
  DPUDeployments
    DPUDeployment/hbn-only                    dpf-operator-system
Ready: True   Success    2h
      DPUServiceChains
        DPUServiceChain/hbn-only-wkdhz        dpf-operator-system
Ready: True   Success    2h
      DPUSets
        DPUSet/hbn-only-dpuset1               dpf-operator-system
          BFB/bf-bundle                       dpf-operator-system
Ready: True   Ready      2h     File: bf-bundle-3.2.1-
34_25.11_ubuntu-24.04_64k_prod.bfb, DOCA: 3.2.1
          DPUs
            2 DPUs...                         dpf-operator-system
Ready: True   DPUReady   2h     See dpu-node-mt2310xz03lr-
mt2310xz03lr, dpu-node-mt2310xz03m2-mt2310xz03m2
      Services
        DPUServiceTemplates
          DPUServiceTemplate/doca-hbn         dpf-operator-system
Ready: True   Success    2h
        DPUServices
          DPUService/doca-hbn-l2xsl           dpf-operator-system
Ready: True   Success    2h
```

Deploy Test Pods

```
kubectl apply -f manifests/05-test-traffic
```

HBN functionality can be tested by pinging between the pods and services deployed in the default namespace.

Uninstall

This section describes how to clean up the DPF components installed in this guide. It is recommended to run this section only after the DPF Operator and DPUCluster are no longer needed.

Delete DPF CNI acceleration components

```
kubectl delete -f manifests/03-enable-accelerated-interfaces --
wait
helm uninstall -n nvidia-network-operator network-operator --wait
```

Delete the DPF Operator system and DPF Operator

```
kubectl delete -n dpf-operator-system dpfoperatorconfig
dpfoperatorconfig --wait
helm uninstall -n dpf-operator-system dpf-operator --wait
```

Delete DPF Operator PVC

```
kubectl -n dpf-operator-system delete pvc bfb-pvc
kubectl delete pv bfb-pv
```

Note: there can be a race condition with deleting the underlying Kamaji cluster which runs the DPU cluster control plane in this guide. If that happens it may be necessary to remove finalizers manually from `DPUCluster` and `Datastore` objects.

OVN Kubernetes

> **Note**
>
> Follow this guide from the source GitHub repo at github.com/NVIDIA/doca-platform and moving to the `docs/public/user-guides/host-trusted/use-cases/ovnk/README.md` for better formatting of the code.

In this configuration OVN Kubernetes is offloaded to the DPU.

> **Warning**
>
> Due to several known issues regarding the stability of this specific deployment use-case (OVN Kubernetes), it should be considered a proof-of-concept in this release.
>
> Please use it at your own risk!

Prerequisites

This guide should be run by cloning the repo from github.com/NVIDIA/doca-platform and moving to the `docs/public/user-guides/ovn_only` directory.

The system is set up as described in the system prerequisites. The OVN Kubernetes deployment has these additional requirements:

Software Prerequisites

This guide uses the following tools which must be installed on the machine where the commands contained in this guide run.

- kubectl
- helm
- envsubst

Network Prerequisites

Control Plane Nodes

- Open vSwitch (OVS) packages installed - i.e. `openvswitch-switch` for Ubuntu 24.04
- out-of-band management port should be configured as OVS bridge port with "bridge-uplink" OVS metadata This addresses a known issue.
- DNS stub resolver should be disabled if using systemd resolvd

Worker Nodes

- Open vSwitch (OVS) packages not installed
- Only a single DPU uplink is used with this deployment (p0).
- All worker nodes are connected to the same L2 broadcast domain (VLAN) on the high-speed network.
- Host high-speed port (Host PF0) must have DHCP enabled.

- MTU of the port should be statically set to 1500
- **Note**: These settings can be configured automatically via DPUFlavor `hostNetworkInterfaceConfigs`
- An external DHCP Server should be used for the high-speed network:
    - The DHCP server must not assign a default gateway to the DHCP clients.
    - The DHCP server should assign a special route (option 121) for a "dummy" IP subnet with a next hop address of the actual default gateway router serving the high-speed network.
    - The special route (which is configurable) is used by DPF to inject the default gateway into the overlay network. By default, DPF is looking for the subnet 169.254.99.100/32 in the special route and extracts the gateway address.
    - The gateway value sent using option 121 should be calculated according to RFC3442 (An online calculator exists). For example, the value of "20:a9:fe:63:64:0a:00:7b:fe" represents a route to 169.254.99.100/32 via 10.0.123.254.

Kubernetes prerequisites

- CNI not installed
- kube-proxy not installed
- coreDNS should be configured to run only on control plane nodes - e.g. using NodeAffinity.
- control plane setup is complete before starting this guide
- worker nodes are not added until indicated by this guide

Control Plane Nodes

- Have the labels:
    - `"k8s.ovn.org/zone-name": $KUBERNETES_NODE_NAME`

Worker Nodes

- Have the labels:
  - `"k8s.ovn.org/dpu-host": ""`
  - `"k8s.ovn.org/zone-name": $KUBERNETES_NODE_NAME`
- Have the annotations:
  - `"k8s.ovn.org/remote-zone-migrated": $KUBERNETES_NODE_NAME`

Virtual Functions

A number of virtual functions (VFs) will be created on hosts when provisioning DPUs. Certain of these VFs are marked for specific usage:

- The first VF (vf0) is used by provisioning components
- The second VF (vf1) is used by ovn-kubernetes
- The remaining VFs are allocated by SR-IOV Device Plugin. Each pod using OVN Kubernetes in DPU mode as its primary CNI will have one of these VFs injected at Pod creation time

Installation Guide

0. Required Variables

The following variables are required by this guide. A sensible default is provided where it makes sense, but many will be specific to the target infrastructure.

Commands in this guide are run in the same directory that contains this readme.

Environment variables file

```
## IP Address for the Kubernetes API server of the target cluster
on which DPF is installed.
## This should never include a scheme or a port.
## e.g. 10.10.10.10
export TARGETCLUSTER_API_SERVER_HOST=

## Port for the Kubernetes API server of the target cluster on
which DPF is installed.
export TARGETCLUSTER_API_SERVER_PORT=6443

## IP address range for hosts in the target cluster on which DPF
is installed.
## This is a CIDR in the form e.g. 10.10.10.0/24
export TARGETCLUSTER_NODE_CIDR=

## IP address range for VTEPs used by OVN Kubernetes. This should
align with the VTEP CIDR used in the DHCP server that
## serves the high speed fabric. In configurations where
different ranges are used per rack, the value should be set to
## the superset CIDR that includes all these ranges.
## This is a CIDR in the form e.g. 10.0.120.0/22
export VTEP_CIDR=

## Virtual IP used by the load balancer for the DPU Cluster. Must
be a reserved IP from the management subnet and not allocated by
DHCP.
export DPUCLUSTER_VIP=

## DPU_P0 is the name of the first port of the DPU. This name
must be the same on all worker nodes.
export DPU_P0=

## DPU_P0_VF1 is the name of the second Virtual Function (VF) of
the first port of the DPU. This name must be the same on all
worker nodes.
```

```
## Note: The VF will be created after the DPU is provisioned and
the phase "Host Network Configuration" is completed.
export DPU_P0_VF1=

## Interface/bridge on which the DPUCluster load balancer will
listen. Should be the management interface/bridge of the control
plane node.
export DPUCLUSTER_INTERFACE=

## IP address to the NFS server used as storage for the BFB.
export NFS_SERVER_IP=

## The repository URL for the NVIDIA Helm chart registry.
## Usually this is the NVIDIA Helm NGC registry. For development
purposes, this can be set to a different repository.
export HELM_REGISTRY_REPO_URL=https://helm.ngc.nvidia.com/nvidia/doca

## The repository URL for the OVN-Kubernetes Helm chart.
## Usually this is the NVIDIA GHCR repository. For development
purposes, this can be set to a different repository.
export OVN_KUBERNETES_REPO_URL=oci://ghcr.io/nvidia

## POD_CIDR is the CIDR used for pods in the target Kubernetes
cluster.
export POD_CIDR=10.233.64.0/18

## SERVICE_CIDR is the CIDR used for services in the target
Kubernetes cluster.
## This is a CIDR in the form e.g. 10.10.10.0/24
export SERVICE_CIDR=10.233.0.0/18

## The DPF REGISTRY is the Helm repository URL where the DPF
Operator Chart resides.
## Usually this is the NVIDIA Helm NGC registry. For development
purposes, this can be set to a different repository.
export REGISTRY=https://helm.ngc.nvidia.com/nvidia/doca
```

```
## The DPF TAG is the version of the DPF components which will be deployed in this guide.
export TAG=v25.10.0

## URL to the BFB used in the `bfb.yaml` and linked by the DPUSet.
export BFB_URL="https://content.mellanox.com/BlueField/BFBs/Ubuntu24.04/bf-bundle-3.2.1-34_25.11_ubuntu-24.04_64k_prod.bfb"
```

Modify the variables in `manifests/00-env-vars/envvars.env` to fit your environment, then source the file:

```
source manifests/00-env-vars/envvars.env
```

1. CNI Installation

OVN Kubernetes is used as the primary CNI for the cluster. On worker nodes the primary CNI will be accelerated by offloading work to the DPU. On control plane nodes OVN Kubernetes will run without offloading.

Create the Namespace

```
kubectl create ns ovn-kubernetes
```

Install OVN Kubernetes from the helm chart

Install the OVN Kubernetes CNI components from the helm chart. A number of environment variables must be set before running this command.

```
envsubst < manifests/01-cni-installation/helm-values/ovn-
kubernetes.yml | helm upgrade --install -n ovn-kubernetes ovn-
kubernetes ${OVN_KUBERNETES_REPO_URL}/ovn-kubernetes-chart --
version $TAG --values -
```

OVN-Kubernetes Helm values

```
commonManifests:
  enabled: true
nodeWithoutDPUManifests:
  enabled: true
controlPlaneManifests:
  enabled: true
nodeWithDPUManifests:
  enabled: true
  nodeMgmtPortNetdev: $DPU_P0_VF1
  dpuServiceAccountNamespace: dpf-operator-system
gatewayOpts: --gateway-interface=$DPU_P0
k8sAPIServer:
https://$TARGETCLUSTER_API_SERVER_HOST:$TARGETCLUSTER_API_SERVER_PORT
## Note this CIDR is followed by a trailing /24 which informs OVN
Kubernetes on how to split the CIDR per node.
podNetwork: $POD_CIDR/24
serviceNetwork: $SERVICE_CIDR
```

Verification

These verification commands may need to be run multiple times to ensure the condition is met.

Verify the CNI installation with:

```
## Ensure all nodes in the cluster are ready.
kubectl wait --for=condition=ready nodes --all
## Ensure all pods in the ovn-kubernetes namespace are ready.
kubectl wait --for=condition=ready --namespace ovn-kubernetes pods
--all --timeout=300s
```

2. DPF Operator Installation

Create storage required by the DPF Operator

A number of environment variables must be set before running this command.

```
kubectl create namespace dpf-operator-system
cat manifests/02-dpf-operator-installation/*.yaml | envsubst |
kubectl apply -f -
```

This deploys the following objects:

PersistentVolume and PersistentVolumeClaim for the provisioning controller

```
---
apiVersion: v1
kind: PersistentVolume
metadata:
  name: bfb-pv
spec:
  capacity:
    storage: 10Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteMany
  nfs:
    path: /mnt/dpf_share/bfb
    server: $NFS_SERVER_IP
  persistentVolumeReclaimPolicy: Delete
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: bfb-pvc
  namespace: dpf-operator-system
spec:
  accessModes:
  - ReadWriteMany
  resources:
    requests:
      storage: 10Gi
  volumeMode: Filesystem
  storageClassName: ""
```

Additional Dependencies

Before deploying the DPF Operator, ensure that Helm is properly configured according to the Helm prerequisites.

> **Warning**
>
> This is a critical prerequisite step that must be completed for the DPF Operator to function properly.

After applying the additional dependencies you MUST ensure that the KUBERNETES_SERVICE_HOST and KUBERNETES_SERVICE_PORT environment variables are set in the node-feature-discovery-worker DaemonSet.

NFD needs to target the VIP because it needs to be up before cluster services can work.

Example commands to set the environment variables:

```
kubectl -n dpf-operator-system set env daemonset/node-feature-
discovery-worker \
KUBERNETES_SERVICE_HOST=$TARGETCLUSTER_API_SERVER_HOST \
KUBERNETES_SERVICE_PORT=$TARGETCLUSTER_API_SERVER_PORT
```

Deploy the DPF Operator

A number of environment variables must be set before running this command.

HTTP Registry (default)

If the $REGISTRY is an HTTP Registry (default value) use this command:

```
helm repo add --force-update dpf-repository ${REGISTRY}
helm repo update
helm upgrade --install -n dpf-operator-system dpf-operator dpf-
repository/dpf-operator --version=$TAG
```

OCI Registry

For development purposes, if the $REGISTRY is an OCI Registry use this command:

```
helm upgrade --install -n dpf-operator-system dpf-operator
$REGISTRY/dpf-operator --version=$TAG
```

Verification

These verification commands may need to be run multiple times to ensure the condition is met.

Verify the DPF Operator installation with:

```
## Ensure the DPF Operator deployment is available.
kubectl rollout status deployment --namespace dpf-operator-system
dpf-operator-controller-manager
## Ensure all pods in the DPF Operator system are ready.
kubectl wait --for=condition=ready --namespace dpf-operator-system
pods --all
```

3. DPF System Installation

This section involves creating the DPF system components and some basic infrastructure required for a functioning DPF-enabled cluster.

Deploy the DPF System components

A number of environment variables must be set before running this command.

```
kubectl create ns dpu-cplane-tenant1
cat manifests/03-dpf-system-installation/*.yaml | envsubst |
kubectl apply -f -
```

This will create the following objects:

DPFOperatorConfig to install the DPF System components

```
---
apiVersion: operator.dpu.nvidia.com/v1alpha1
kind: DPFOperatorConfig
metadata:
  name: dpfoperatorconfig
  namespace: dpf-operator-system
spec:
  overrides:
    kubernetesAPIServerVIP: $TARGETCLUSTER_API_SERVER_HOST
    kubernetesAPIServerPort: $TARGETCLUSTER_API_SERVER_PORT
  provisioningController:
    bfbPVCName: "bfb-pvc"
    dmsTimeout: 900
  kamajiClusterManager:
    disable: false
```

DPUCluster to serve as Kubernetes control plane for DPU nodes

```
---
apiVersion: provisioning.dpu.nvidia.com/v1alpha1
kind: DPUCluster
metadata:
  name: dpu-cplane-tenant1
  namespace: dpu-cplane-tenant1
spec:
  type: kamaji
  maxNodes: 10
  clusterEndpoint:
    # deploy keepalived instances on the nodes that match the
given nodeSelector.
    keepalived:
      # interface on which keepalived will listen. Should be the oob
interface of the control plane node.
      interface: $DPUCLUSTER_INTERFACE
      # Virtual IP reserved for the DPU Cluster load balancer.
Must not be allocatable by DHCP.
      vip: $DPUCLUSTER_VIP
      # virtualRouterID must be in range [1,255], make sure the
given virtualRouterID does not duplicate with any existing
keepalived process running on the host
      virtualRouterID: 126
      nodeSelector:
        node-role.kubernetes.io/control-plane: ""
```

Verification

These verification commands may need to be run multiple times to ensure the condition is met.

Verify the DPF System with:

```
## Ensure the provisioning and DPUService controller manager
deployments are available.
kubectl rollout status deployment --namespace dpf-operator-system
dpf-provisioning-controller-manager dpuservice-controller-manager
## Ensure all other deployments in the DPF Operator system are
Available.
kubectl rollout status deployment --namespace dpf-operator-system
## Ensure the DPUCluster is ready for nodes to join.
kubectl wait --for=condition=ready --namespace dpu-cplane-tenant1
dpucluster --all
```

4. Install Components to Enable Accelerated CNI Nodes

OVN Kubernetes will accelerate traffic by attaching a VF to each pod using the primary CNI. This VF is used to offload flows to the DPU. This section details the components needed to connect pods to the offloaded OVN Kubernetes CNI.

Install Multus and SRIOV Network Operator using NVIDIA Network Operator

```
helm repo add nvidia https://helm.ngc.nvidia.com/nvidia --force-update
helm upgrade --no-hooks --install --create-namespace --namespace
nvidia-network-operator network-operator nvidia/network-operator
--version 25.7.0 -f ./manifests/04-enable-accelerated-cni/helm-
values/network-operator.yml
```

NVIDIA Network Operator Helm values

```yaml
nfd:
  enabled: false
  deployNodeFeatureRules: false
sriovNetworkOperator:
  enabled: true
sriov-network-operator:
  operator:
    affinity:
      nodeAffinity:
        requiredDuringSchedulingIgnoredDuringExecution:
          nodeSelectorTerms:
            - matchExpressions:
                - key: node-role.kubernetes.io/master
                  operator: Exists
            - matchExpressions:
                - key: node-role.kubernetes.io/control-plane
                  operator: Exists
  crds:
    enabled: true
  sriovOperatorConfig:
    deploy: true
    configDaemonNodeSelector: null
operator:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
          - matchExpressions:
              - key: node-role.kubernetes.io/master
                operator: Exists
          - matchExpressions:
              - key: node-role.kubernetes.io/control-plane
                operator: Exists
```

Install the OVN Kubernetes resource injection webhook

The OVN Kubernetes resource injection webhook injected each pod scheduled to a worker node with a request for a VF and a Network Attachment Definition. This webhook is part of the same helm chart as the other components of the OVN Kubernetes CNI. Here it is installed by adjusting the existing helm installation to add the webhook component to the installation.

```
envsubst < manifests/04-enable-accelerated-cni/helm-values/ovn-
kubernetes.yml | helm upgrade --install -n ovn-kubernetes ovn-
kubernetes-resource-injector ${OVN_KUBERNETES_REPO_URL}/ovn-
kubernetes-chart --version $TAG --values -
```

OVN Kubernetes Resource Injector Helm values

```
ovn-kubernetes-resource-injector:
  ## Enable the ovn-kubernetes-resource-injector
  enabled: true
```

Apply the NICClusterConfiguration and SriovNetworkNodePolicy

```
cat manifests/04-enable-accelerated-cni/*.yaml | envsubst |
kubectl apply -f -
```

This will deploy the following objects:

NICClusterPolicy for the NVIDIA Network Operator

```yaml
---
apiVersion: mellanox.com/v1alpha1
kind: NicClusterPolicy
metadata:
  name: nic-cluster-policy
spec:
  secondaryNetwork:
    multus:
      image: multus-cni
      imagePullSecrets: []
      repository: ghcr.io/k8snetworkplumbingwg
      version: v3.9.3
```

SriovNetworkNodePolicy for the SR-IOV Network Operator

```
---
apiVersion: sriovnetwork.openshift.io/v1
kind: SriovNetworkNodePolicy
metadata:
  name: bf3-p0-vfs
  namespace: nvidia-network-operator
spec:
  nicSelector:
    deviceID: "a2dc"
    vendor: "15b3"
    pfNames:
    - $DPU_P0#2-45
  nodeSelector:
    node-role.kubernetes.io/worker: ""
  numVfs: 46
  resourceName: bf3-p0-vfs
  isRdma: true
  externallyManaged: true
  deviceType: netdevice
  linkType: eth
```

Verification

These verification commands may need to be run multiple times to ensure the condition is met.

Verify that the accelerated CNI is enabled with:

```
## Ensure the nvidia-network-operator pods are ready.
kubectl wait --for=condition=Ready --namespace nvidia-network-
operator pods --all
## Expect the following Daemonsets to be successfully rolled out.
kubectl rollout status daemonset --namespace nvidia-network-
operator kube-multus-ds sriov-network-config-daemon sriov-device-
plugin
## Expect the network injector to be successfully rolled out.
kubectl rollout status deployment --namespace ovn-kubernetes ovn-
kubernetes-resource-injector
```

5. DPU Provisioning and Service Installation

In this step we deploy our DPUs and the services that will run on them.

The user is expected to create a DPUDeployment object that reflects a set of DPUServices that should run on a set of DPUs.

> If you want to learn more about `DPUDeployments`, feel free to check the DPUDeployment documentation.

Create the DPUDeployment, DPUServiceConfig, DPUServiceTemplate and other necessary objects

> ⚠ **Warning**
>
> In case more than 1 DPU exists per node, the relevant selector should be applied in the DPUDeployment to select the appropriate DPU. See DPUDeployment - DPUs Configuration to understand more about the selectors.

A number of environment variables must be set before running this command.

```
cat manifests/05-dpudeployment-installation/*.yaml | envsubst |
kubectl apply -f -
```

This will deploy the following objects:

BFB to download Bluefield Bitstream to a shared volume

```
---
apiVersion: provisioning.dpu.nvidia.com/v1alpha1
kind: BFB
metadata:
  name: bf-bundle-$TAG
  namespace: dpf-operator-system
spec:
  url: $BFB_URL
```

OVN DPUFlavor to correctly configure the DPUs on provisioning

```
---
apiVersion: provisioning.dpu.nvidia.com/v1alpha1
kind: DPUFlavor
metadata:
  name: ovnk-$TAG
  namespace: dpf-operator-system
spec:
  grub:
    kernelParameters:
      - console=hvc0
      - console=ttyAMA0
      - earlycon=pl011,0x13010000
      - fixrttc
      - net.ifnames=0
      - biosdevname=0
      - iommu.passthrough=1
      - cgroup_no_v1=net_prio,net_cls
      - hugepagesz=2048kB
      - hugepages=3072
  nvconfig:
    - device: "*"
      parameters:
      - PF_BAR2_ENABLE=0
      - PER_PF_NUM_SF=1
      - PF_TOTAL_SF=20
      - PF_SF_BAR_SIZE=10
      - NUM_PF_MSIX_VALID=0
      - PF_NUM_PF_MSIX_VALID=1
      - PF_NUM_PF_MSIX=228
      - INTERNAL_CPU_MODEL=1
      - INTERNAL_CPU_OFFLOAD_ENGINE=0
      - SRIOV_EN=1
      - NUM_OF_VFS=46
      - LAG_RESOURCE_ALLOCATION=1
      - LINK_TYPE_P1=ETH
```

```
      - LINK_TYPE_P2=ETH
  ovs:
    rawConfigScript: |
      _ovs-vsctl() {
        ovs-vsctl --no-wait --timeout 15 "$@"
      }

      _ovs-vsctl set Open_vSwitch . other_config:doca-init=true
      _ovs-vsctl set Open_vSwitch . other_config:dpdk-max-
memzones=50000
      _ovs-vsctl set Open_vSwitch . other_config:hw-offload=true
      _ovs-vsctl set Open_vSwitch . other_config:pmd-quiet-
idle=true
      _ovs-vsctl set Open_vSwitch . other_config:max-idle=20000
      _ovs-vsctl set Open_vSwitch . other_config:max-
revalidator=5000
      _ovs-vsctl set Open_vSwitch . other_config:ctl-pipe-
size=1024
      _ovs-vsctl --if-exists del-br ovsbr1
      _ovs-vsctl --if-exists del-br ovsbr2
      _ovs-vsctl --may-exist add-br br-sfc
      _ovs-vsctl set bridge br-sfc datapath_type=netdev
      _ovs-vsctl set bridge br-sfc fail_mode=secure
      _ovs-vsctl --may-exist add-port br-sfc p0
      _ovs-vsctl set Interface p0 type=dpdk
      _ovs-vsctl set Interface p0 mtu_request=9216
      _ovs-vsctl set Port p0 external_ids:dpf-type=physical

      _ovs-vsctl set Open_vSwitch . external-ids:ovn-bridge-
datapath-type=netdev
      _ovs-vsctl --may-exist add-br br-ovn
      _ovs-vsctl set bridge br-ovn datapath_type=netdev
      _ovs-vsctl br-set-external-id br-ovn bridge-id br-ovn
      _ovs-vsctl br-set-external-id br-ovn bridge-uplink
puplinkbrovntobrsfc
      _ovs-vsctl set Interface br-ovn mtu_request=9216
```

```
    _ovs-vsctl --may-exist add-port br-ovn pf0hpf
    _ovs-vsctl set Interface pf0hpf type=dpdk
    _ovs-vsctl set Interface pf0hpf mtu_request=9216

bfcfgParameters:
  - UPDATE_ATF_UEFI=yes
  - UPDATE_DPU_OS=yes
  - WITH_NIC_FW_UPDATE=yes

hostNetworkInterfaceConfigs:
  - portNumber: 0
    dhcp: true
    mtu: 1500

configFiles:
- path: /etc/mellanox/mlnx-bf.conf
  operation: override
  raw: |
      ALLOW_SHARED_RQ="no"
      IPSEC_FULL_OFFLOAD="no"
      ENABLE_ESWITCH_MULTIPORT="yes"
  permissions: "0644"
- path: /etc/mellanox/mlnx-ovs.conf
  operation: override
  raw: |
      CREATE_OVS_BRIDGES="no"
      OVS_DOCA="yes"
  permissions: "0644"
- path: /etc/mellanox/mlnx-sf.conf
  operation: override
  raw: ""
  permissions: "0644"
```

DPUDeployment to provision DPUs on worker nodes

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUDeployment
metadata:
  name: ovn
  namespace: dpf-operator-system
spec:
  dpus:
    bfb: bf-bundle-$TAG
    flavor: ovnk-$TAG
    dpuSets:
    - nameSuffix: "dpuset1"
      nodeSelector:
        matchLabels:
          feature.node.kubernetes.io/dpu-enabled: "true"
      dpuSelector:
        provisioning.dpu.nvidia.com/dpudevice-pf0-name: $DPU_P0
  services:
    ovn:
      serviceTemplate: ovn
      serviceConfiguration: ovn
    dts:
      serviceTemplate: dts
      serviceConfiguration: dts
    blueman:
      serviceTemplate: blueman
      serviceConfiguration: blueman
  serviceChains:
    switches:
      - ports:
        - serviceInterface:
            matchLabels:
              uplink: p0
        - serviceInterface:
            matchLabels:
```

```
                port: ovn
```

OVN DPUServiceConfig and DPUServiceTemplate to deploy OVN workloads to the DPUs

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceConfiguration
metadata:
  name: ovn
  namespace: dpf-operator-system
spec:
  deploymentServiceName: "ovn"
  serviceConfiguration:
    helmChart:
      values:
        k8sAPIServer:
https://$TARGETCLUSTER_API_SERVER_HOST:$TARGETCLUSTER_API_SERVER_PORT
        podNetwork: $POD_CIDR/24
        serviceNetwork: $SERVICE_CIDR
        dpuManifests:
          kubernetesSecretName: "ovn-dpu" # user needs to populate
based on DPUServiceCredentialRequest
          vtepCIDR: $VTEP_CIDR
          hostCIDR: $TARGETCLUSTER_NODE_CIDR
          externalDHCP: true
          gatewayDiscoveryNetwork: "169.254.99.100/32" # This is a
"dummy" subnet used to get the default gateway address from DHCP
server (via option 121)
```

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceTemplate
metadata:
  name: ovn
  namespace: dpf-operator-system
spec:
  deploymentServiceName: "ovn"
  helmChart:
    source:
      repoURL: $OVN_KUBERNETES_REPO_URL
      chart: ovn-kubernetes-chart
      version: $TAG
    values:
      commonManifests:
        enabled: true
      dpuManifests:
        enabled: true
      leaseNamespace: "ovn-kubernetes"
      gatewayOpts: "--gateway-interface=br-ovn"
```

DOCA Telemetry Service DPUServiceConfig and DPUServiceTemplate to deploy DTS to the DPUs

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceConfiguration
metadata:
  name: dts
  namespace: dpf-operator-system
spec:
  deploymentServiceName: "dts"
```

```yaml
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceTemplate
metadata:
  name: dts
  namespace: dpf-operator-system
spec:
  deploymentServiceName: "dts"
  helmChart:
    source:
      repoURL: $HELM_REGISTRY_REPO_URL
      version: 1.23.4
      chart: doca-telemetry
```

Blueman DPUServiceConfig and DPUServiceTemplate to deploy Blueman to the DPUs

```yaml
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceConfiguration
metadata:
  name: blueman
  namespace: dpf-operator-system
spec:
  deploymentServiceName: "blueman"
```

```yaml
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceTemplate
metadata:
  name: blueman
  namespace: dpf-operator-system
spec:
  deploymentServiceName: "blueman"
  helmChart:
    source:
      repoURL: $HELM_REGISTRY_REPO_URL
      version: 1.0.8
      chart: doca-blueman
```

OVN DPUServiceCredentialRequest to allow cross cluster communication

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceCredentialRequest
metadata:
  name: ovn-dpu
  namespace: dpf-operator-system
spec:
  serviceAccount:
    name: ovn-dpu
    namespace: dpf-operator-system
  duration: 24h
  type: tokenFile
  secret:
    name: ovn-dpu
    namespace: dpf-operator-system
  metadata:
    labels:
      dpu.nvidia.com/image-pull-secret: ""
```

DPUServiceInterfaces for physical ports on the DPU

```yaml
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceInterface
metadata:
  name: p0
  namespace: dpf-operator-system
spec:
  template:
    spec:
      template:
        metadata:
          labels:
            uplink: "p0"
        spec:
          interfaceType: physical
          physical:
            interfaceName: p0
```

OVN DPUServiceInterface to define the ports attached to OVN workloads on the DPU

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceInterface
metadata:
  name: ovn
  namespace: dpf-operator-system
spec:
  template:
    spec:
      template:
        metadata:
          labels:
            port: ovn
        spec:
          interfaceType: ovn
```

Verification

These verification commands may need to be run multiple times to ensure the condition is met.

Note that the DPUService name will have a random suffix. For example, `ovn-ovn-l2xsl`.

Verify the DPU and Service installation with:

```
## Ensure the DPUServices are created and have been reconciled.
kubectl wait --for=condition=ApplicationsReconciled --namespace
dpf-operator-system dpuservices -l svc.dpu.nvidia.com/owned-by-
dpudeployment=dpf-operator-system_ovn
## Ensure the DPUServiceIPAMs have been reconciled
kubectl wait --for=condition=DPUIPAMObjectReconciled --namespace
dpf-operator-system dpuserviceipam --all
## Ensure the DPUServiceInterfaces have been reconciled
kubectl wait --for=condition=ServiceInterfaceSetReconciled --
namespace dpf-operator-system dpuserviceinterface --all
## Ensure the DPUServiceChains have been reconciled
kubectl wait --for=condition=ServiceChainSetReconciled --namespace
dpf-operator-system dpuservicechain --all
```

6. Test Traffic

Add worker nodes to the cluster

At this point workers should be added to the cluster. Each worker node should be configured in line with the prerequisites. As workers are added to the cluster DPUs will be provisioned and DPUServices will begin to be spun up.

You can verify the status of the DPUDeployment and its components with the following command:

```
$ kubectl -n dpf-operator-system exec deploy/dpf-operator-
controller-manager -- /dpfctl describe dpudeployments
NAME                                          NAMESPACE
STATUS        REASON        SINCE  MESSAGE
DPFOperatorConfig/dpfoperatorconfig           dpf-operator-system
Ready: True   Success       2h
  DPUDeployments
    DPUDeployment/ovn                         dpf-operator-system
Ready: True   Success       2h
      DPUServiceChains
        DPUServiceChain/ovn-wkdhz             dpf-operator-system
Ready: True   Success       2h
      DPUSets
        DPUSet/hbn-only-dpuset1               dpf-operator-system
          BFB/bf-bundle                       dpf-operator-system
Ready: True   Ready       2h     File: bf-bundle-3.2.1-
34_25.11_ubuntu-24.04_64k_prod.bfb, DOCA: 3.2.1
          DPUs
            2 DPUs...                         dpf-operator-system
Ready: True   DPUReady    2h     See dpu-node-mt2310xz03lr-
mt2310xz03lr, dpu-node-mt2310xz03m2-mt2310xz03m2
      Services
        DPUServiceTemplates
          4 DPUServiceTemplates...            dpf-operator-system
Ready: True   Success     2h     See ovn-blueman, ovn-dts, ovn-
ovn
        DPUServices
          4 DPUServices...                    dpf-operator-system
Ready: True   Success     2h     See ovn-blueman-4tlmp, ovn-dts-
49282, ovn-ovn-l2xsl
```

Deploy test pods

```
kubectl apply -f manifests/06-test-traffic
```

OVN functionality can be tested by pinging between the pods and services deployed in the default namespace.

Uninstall

This section describes how to clean up the cluster after the DPF setup is no longer needed. It is important to follow the order of deletion to ensure that all components are removed cleanly and that the cluster remains functional.

Delete the test pods

```
kubectl delete -f manifests/06-test-traffic --wait
```

Delete DPF CNI acceleration components

```
kubectl delete -f manifests/04-enable-accelerated-cni --wait
helm uninstall -n nvidia-network-operator network-operator --wait

## Note: Uninstalling OVN Kubernetes as primary CNI is not
supported but this command must be run to remove the webhook and
restore a functioning cluster.
helm uninstall -n ovn-kubernetes ovn-kubernetes-resource-injector
--wait
```

Delete the DPF Operator system and DPF Operator

First we have to delete some DPUServiceInterfaces. This is necessary because of a known issue during uninstallation.

```
kubectl delete -n dpf-operator-system dpuserviceinterface p0 ovn
--wait
```

Then we can delete the config and system namespace.

```
kubectl delete -n dpf-operator-system dpfoperatorconfig
dpfoperatorconfig --wait
helm uninstall -n dpf-operator-system dpf-operator --wait
```

Delete DPF Operator PVC

```
kubectl -n dpf-operator-system delete pvc bfb-pvc
kubectl delete pv bfb-pv
```

Note: there can be a race condition with deleting the underlying Kamaji cluster which runs the DPU cluster control plane in this guide. If that happens it may be necessary to remove finalizers manually from `DPUCluster` and `Datastore` objects.

Limitations of DPF Setup

Host network pod services

The Kubelet process on the Kubernetes nodes use the OOB interface IP address to register in Kubernetes. This means that the nodes have the OOB IP addresses as node IP addresses. This means that pods using host networking have the OOB IP address of the hosts as pod IP address. However, that interface is not accelerated. This means that any

component using the addresses of the pods using host networking will not benefit from hardware acceleration and high-speed ports.

For example, this means that when creating a Kubernetes NodePort service selecting pods using host networking, even if the user uses the high-speed IP of the host, the traffic will not be accelerated. In order to solve this, it is possible to create dedicated endpointSlices that contain the host high-speed port IP addresses instead of OOB port IP addresses. This way, the entire path to the pods will be accelerated and benefit from high performances, if the user uses the high speed IP address of the host with the nodePort port. This requires the workload running on the pod with host networking to also listen on the high-speed port IP address.

OVN Kubernetes with Host Based Networking

> **Note**
>
> Follow this guide from the source GitHub repo at github.com/NVIDIA/doca-platform and moving to the `docs/public/user-guides/host-trusted/use-cases/hbn-ovnk/README.md` for better formatting of the code.

In this configuration OVN Kubernetes is offloaded to the DPU and combined with NVIDIA Host Based Networking (HBN).

Prerequisites

This guide should be run by cloning the repo from github.com/NVIDIA/doca-platform and moving to the `docs/public/user-guides/hbn_ovn` directory.

The system is set up as described in the system prerequisites. The OVN Kubernetes with HBN case has the additional requirements:

DPU Prerequisites

- Bluefield 3 with 32GB of RAM

Software Prerequisites

This guide uses the following tools which must be installed on the machine where the commands contained in this guide run.

- kubectl
- helm
- envsubst

Network Prerequisites

Control Plane Nodes

- Open vSwitch (OVS) packages installed - i.e. `openvswitch-switch` for Ubuntu 24.04
- out-of-band management port should be configured as OVS bridge port with "bridge-uplink" OVS metadata This addresses a known issue
- DNS stub resolver should be disabled if using systemd resolvd

Worker Nodes

- Open vSwitch (OVS) packages not installed
- Host high-speed port (Host PF0) must have DHCP enabled
- MTU of the port should be statically set to 1500

- **Note**: These settings can be configured automatically via DPUFlavor `hostNetworkInterfaceConfigs`

Kubernetes prerequisites

- CNI not installed
- Kube-proxy not installed
- CoreDNS should be configured to run only on control plane nodes - e.g. using NodeAffinity. This addresses a known issue.
- Control plane setup is complete before starting this guide
- Worker nodes are not added until indicated by this guide

Control Plane Nodes

- Have the labels:
  - `"k8s.ovn.org/zone-name": $KUBERNETES_NODE_NAME`

Worker Nodes

- Have the labels:
  - `"k8s.ovn.org/dpu-host": ""`
  - `"k8s.ovn.org/zone-name": $KUBERNETES_NODE_NAME`
- Have the annotations:
  - `"k8s.ovn.org/remote-zone-migrated": $KUBERNETES_NODE_NAME`

Virtual Functions

A number of virtual functions (VFs) will be created on hosts when provisioning DPUs. Certain of these VFs are marked for specific usage:

- The first VF (vf0) is used by provisioning components
- The second VF (vf1) is used by ovn-kubernetes
- The remaining VFs are allocated by SR-IOV Device Plugin. Each pod using OVN Kubernetes in DPU mode as its primary CNI will have one of these VFs injected at Pod creation time

Installation Guide

0. Required Variables

The following variables are required by this guide. A sensible default is provided where it makes sense, but many will be specific to the target infrastructure.

Commands in this guide are run in the same directory that contains this readme.

Environment variables file

```
## IP Address for the Kubernetes API server of the target cluster
on which DPF is installed.
## This should never include a scheme or a port.
## e.g. 10.10.10.10
export TARGETCLUSTER_API_SERVER_HOST=

## Port for the Kubernetes API server of the target cluster on
which DPF is installed.
export TARGETCLUSTER_API_SERVER_PORT=6443

## IP address range for hosts in the target cluster on which DPF
is installed.
## This is a CIDR in the form e.g. 10.10.10.0/24
export TARGETCLUSTER_NODE_CIDR=

## Virtual IP used by the load balancer for the DPU Cluster. Must
be a reserved IP from the management subnet and not allocated by
DHCP.
export DPUCLUSTER_VIP=

## DPU_P0 is the name of the first port of the DPU. This name
must be the same on all worker nodes.
export DPU_P0=

## DPU_P0_VF1 is the name of the second Virtual Function (VF) of
the first port of the DPU. This name must be the same on all
worker nodes.
## Note: The VF will be created after the DPU is provisioned and
the phase "Host Network Configuration" is completed.
export DPU_P0_VF1=

## Interface on which the DPUCluster load balancer will listen.
Should be the management interface of the control plane node.
export DPUCLUSTER_INTERFACE=
```

```
## IP address to the NFS server used as storage for the BFB.
export NFS_SERVER_IP=

## The repository URL for the NVIDIA Helm chart registry.
## Usually this is the NVIDIA Helm NGC registry. For development
purposes, this can be set to a different repository.
export HELM_REGISTRY_REPO_URL=https://helm.ngc.nvidia.com/nvidia/doca

## The repository URL for the HBN container image.
## Usually this is the NVIDIA NGC registry. For development
purposes, this can be set to a different repository.
export HBN_NGC_IMAGE_URL=nvcr.io/nvidia/doca/doca_hbn

## The repository URL for the OVN-Kubernetes Helm chart.
## Usually this is the NVIDIA GHCR repository. For development
purposes, this can be set to a different repository.
export OVN_KUBERNETES_REPO_URL=oci://ghcr.io/nvidia

## POD_CIDR is the CIDR used for pods in the target Kubernetes
cluster.
export POD_CIDR=10.233.64.0/18

## SERVICE_CIDR is the CIDR used for services in the target
Kubernetes cluster.
## This is a CIDR in the form e.g. 10.10.10.0/24
export SERVICE_CIDR=10.233.0.0/18

## The DPF REGISTRY is the Helm repository URL where the DPF
Operator Chart resides.
## Usually this is the NVIDIA Helm NGC registry. For development
purposes, this can be set to a different repository.
export REGISTRY=https://helm.ngc.nvidia.com/nvidia/doca

## The DPF TAG is the version of the DPF components which will be
deployed in this guide.
export TAG=v25.10.0
```

```
## URL to the BFB used in the `bfb.yaml` and linked by the
DPUSet.
export BFB_URL="https://content.mellanox.com/BlueField/BFBs/Ubuntu24.04/bf-bundle-3.2.1-
34_25.11_ubuntu-24.04_64k_prod.bfb"
```

Modify the variables in `manifests/00-env-vars/envvars.env` to fit your environment, then source the file:

```
source manifests/00-env-vars/envvars.env
```

1. CNI Installation

OVN Kubernetes is used as the primary CNI for the cluster. On worker nodes the primary CNI will be accelerated by offloading work to the DPU. On control plane nodes OVN Kubernetes will run without offloading.

Create the Namespace

```
kubectl create ns ovn-kubernetes
```

Install OVN Kubernetes from the helm chart

Install the OVN Kubernetes CNI components from the helm chart. A number of environment variables must be set before running this command.

```
envsubst < manifests/01-cni-installation/helm-values/ovn-
kubernetes.yml | helm upgrade --install -n ovn-kubernetes ovn-
kubernetes ${OVN_KUBERNETES_REPO_URL}/ovn-kubernetes-chart --
version $TAG --values -
```

OVN-Kubernetes Helm values

```
commonManifests:
  enabled: true
nodeWithoutDPUManifests:
  enabled: true
controlPlaneManifests:
  enabled: true
nodeWithDPUManifests:
  enabled: true
  nodeMgmtPortNetdev: $DPU_P0_VF1
  dpuServiceAccountNamespace: dpf-operator-system
gatewayOpts: --gateway-interface=$DPU_P0
## Note this CIDR is followed by a trailing /24 which informs OVN
Kubernetes on how to split the CIDR per node.
podNetwork: $POD_CIDR/24
serviceNetwork: $SERVICE_CIDR
k8sAPIServer:
https://$TARGETCLUSTER_API_SERVER_HOST:$TARGETCLUSTER_API_SERVER_PORT
```

Verification

These verification commands may need to be run multiple times to ensure the condition is met.

Verify the CNI installation with:

```
## Ensure all nodes in the cluster are ready.
kubectl wait --for=condition=ready nodes --all
## Ensure all pods in the ovn-kubernetes namespace are ready.
kubectl wait --for=condition=ready --namespace ovn-kubernetes pods
--all --timeout=300s
```

2. DPF Operator Installation

Create storage required by the DPF Operator

A number of environment variables must be set before running this command.

```
kubectl create ns dpf-operator-system
cat manifests/02-dpf-operator-installation/*.yaml | envsubst |
kubectl apply -f -
```

This deploys the following objects:

PersistentVolume and PersistentVolumeClaim for the provisioning controller

```
---
apiVersion: v1
kind: PersistentVolume
metadata:
  name: bfb-pv
spec:
  capacity:
    storage: 10Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteMany
  nfs:
    path: /mnt/dpf_share/bfb
    server: $NFS_SERVER_IP
  persistentVolumeReclaimPolicy: Delete
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: bfb-pvc
  namespace: dpf-operator-system
spec:
  accessModes:
  - ReadWriteMany
  resources:
    requests:
      storage: 10Gi
  volumeMode: Filesystem
  storageClassName: ""
```

Additional Dependencies

Before deploying the DPF Operator, ensure that Helm is properly configured according to the Helm prerequisites.

> **Warning**
>
> This is a critical prerequisite step that must be completed for the DPF Operator to function properly.

After applying the additional dependencies you MUST ensure that the KUBERNETES_SERVICE_HOST and KUBERNETES_SERVICE_PORT environment variables are set in the node-feature-discovery-worker DaemonSet.

NFD needs to target the VIP because it needs to be up before cluster services can work.

Example commands to set the environment variables:

```
kubectl -n dpf-operator-system set env daemonset/node-feature-
discovery-worker \
KUBERNETES_SERVICE_HOST=$TARGETCLUSTER_API_SERVER_HOST \
KUBERNETES_SERVICE_PORT=$TARGETCLUSTER_API_SERVER_PORT
```

Deploy the DPF Operator

A number of environment variables must be set before running this command.

HTTP Registry (default)

If the $REGISTRY is an HTTP Registry (default value) use this command:

```
helm repo add --force-update dpf-repository ${REGISTRY}
helm repo update
helm upgrade --install -n dpf-operator-system dpf-operator dpf-
repository/dpf-operator --version=$TAG
```

OCI Registry

For development purposes, if the $REGISTRY is an OCI Registry use this command:

```
helm upgrade --install -n dpf-operator-system dpf-operator
$REGISTRY/dpf-operator --version=$TAG
```

Verification

These verification commands may need to be run multiple times to ensure the condition is met.

Verify the DPF Operator installation with:

```
## Ensure the DPF Operator deployment is available.
kubectl rollout status deployment --namespace dpf-operator-system
dpf-operator-controller-manager
## Ensure all pods in the DPF Operator system are ready.
kubectl wait --for=condition=ready --namespace dpf-operator-system
pods --all
```

3. DPF System installation

This section involves creating the DPF system components and some basic infrastructure required for a functioning DPF-enabled cluster.

Deploy the DPF System components

A number of environment variables must be set before running this command.

```
kubectl create ns dpu-cplane-tenant1
cat manifests/03-dpf-system-installation/*.yaml | envsubst |
kubectl apply -f -
```

This will create the following objects:

DPFOperatorConfig to install the DPF System components

```
---
apiVersion: operator.dpu.nvidia.com/v1alpha1
kind: DPFOperatorConfig
metadata:
  name: dpfoperatorconfig
  namespace: dpf-operator-system
spec:
  overrides:
    kubernetesAPIServerVIP: $TARGETCLUSTER_API_SERVER_HOST
    kubernetesAPIServerPort: $TARGETCLUSTER_API_SERVER_PORT
  provisioningController:
    bfbPVCName: "bfb-pvc"
    dmsTimeout: 900
  kamajiClusterManager:
    disable: false
```

DPUCluster to serve as Kubernetes control plane for DPU nodes

```
---
apiVersion: provisioning.dpu.nvidia.com/v1alpha1
kind: DPUCluster
metadata:
  name: dpu-cplane-tenant1
  namespace: dpu-cplane-tenant1
spec:
  type: kamaji
  maxNodes: 10
  clusterEndpoint:
    # deploy keepalived instances on the nodes that match the
given nodeSelector.
    keepalived:
      # interface on which keepalived will listen. Should be the oob
interface of the control plane node.
      interface: $DPUCLUSTER_INTERFACE
      # Virtual IP reserved for the DPU Cluster load balancer.
Must not be allocatable by DHCP.
      vip: $DPUCLUSTER_VIP
      # virtualRouterID must be in range [1,255], make sure the
given virtualRouterID does not duplicate with any existing
keepalived process running on the host
      virtualRouterID: 126
      nodeSelector:
        node-role.kubernetes.io/control-plane: ""
```

Verification

These verification commands may need to be run multiple times to ensure the condition is met.

Verify the DPF System with:

```
## Ensure the provisioning and DPUService controller manager
deployments are available.
kubectl rollout status deployment --namespace dpf-operator-system
dpf-provisioning-controller-manager dpuservice-controller-manager
## Ensure all other deployments in the DPF Operator system are
Available.
kubectl rollout status deployment --namespace dpf-operator-system
## Ensure the DPUCluster is ready for nodes to join.
kubectl wait --for=condition=ready --namespace dpu-cplane-tenant1
dpucluster --all
```

4. Install components to enable accelerated CNI nodes

OVN Kubernetes will accelerate traffic by attaching a VF to each pod using the primary CNI. This VF is used to offload flows to the DPU. This section details the components needed to connect pods to the offloaded OVN Kubernetes CNI.

Install Multus and SRIOV Network Operator using NVIDIA Network Operator

```
helm repo add nvidia https://helm.ngc.nvidia.com/nvidia --force-update
helm upgrade --no-hooks --install --create-namespace --namespace
nvidia-network-operator network-operator nvidia/network-operator
--version 25.7.0 -f ./manifests/04-enable-accelerated-cni/helm-
values/network-operator.yml
```

NVIDIA Network Operator Helm values

```yaml
nfd:
  enabled: false
  deployNodeFeatureRules: false
sriovNetworkOperator:
  enabled: true
sriov-network-operator:
  operator:
    affinity:
      nodeAffinity:
        requiredDuringSchedulingIgnoredDuringExecution:
          nodeSelectorTerms:
            - matchExpressions:
                - key: node-role.kubernetes.io/master
                  operator: Exists
            - matchExpressions:
                - key: node-role.kubernetes.io/control-plane
                  operator: Exists
  crds:
    enabled: true
  sriovOperatorConfig:
    deploy: true
    configDaemonNodeSelector: null
operator:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
          - matchExpressions:
              - key: node-role.kubernetes.io/master
                operator: Exists
          - matchExpressions:
              - key: node-role.kubernetes.io/control-plane
                operator: Exists
```

Install the OVN Kubernetes resource injection webhook

The OVN Kubernetes resource injection webhook injected each pod scheduled to a worker node with a request for a VF and a Network Attachment Definition. This webhook is part of the same helm chart as the other components of the OVN Kubernetes CNI. Here it is installed by adjusting the existing helm installation to add the webhook component to the installation.

```
envsubst < manifests/04-enable-accelerated-cni/helm-values/ovn-
kubernetes.yml | helm upgrade --install -n ovn-kubernetes ovn-
kubernetes-resource-injector ${OVN_KUBERNETES_REPO_URL}/ovn-
kubernetes-chart --version $TAG --values -
```

OVN Kubernetes Resource Injector Helm values

```
ovn-kubernetes-resource-injector:
  ## Enable the ovn-kubernetes-resource-injector
  enabled: true
```

Apply the NICClusterConfiguration and SriovNetworkNodePolicy

```
cat manifests/04-enable-accelerated-cni/*.yaml | envsubst |
kubectl apply -f -
```

This will deploy the following objects:

NICClusterPolicy for the NVIDIA Network Operator

```yaml
---
apiVersion: mellanox.com/v1alpha1
kind: NicClusterPolicy
metadata:
  name: nic-cluster-policy
spec:
  secondaryNetwork:
    multus:
      image: multus-cni
      imagePullSecrets: []
      repository: ghcr.io/k8snetworkplumbingwg
      version: v3.9.3
```

SriovNetworkNodePolicy for the SR-IOV Network Operator

```
---
apiVersion: sriovnetwork.openshift.io/v1
kind: SriovNetworkNodePolicy
metadata:
  name: bf3-p0-vfs
  namespace: nvidia-network-operator
spec:
  nicSelector:
    deviceID: "a2dc"
    vendor: "15b3"
    pfNames:
    - $DPU_P0#2-45
  nodeSelector:
    node-role.kubernetes.io/worker: ""
  numVfs: 46
  resourceName: bf3-p0-vfs
  isRdma: true
  externallyManaged: true
  deviceType: netdevice
  linkType: eth
```

Verification

These verification commands may need to be run multiple times to ensure the condition is met.

Verify that the accelerated CNI is enabled with:

```
## Ensure the nvidia-network-operator pods are ready.
kubectl wait --for=condition=Ready --namespace nvidia-network-
operator pods --all
## Expect the following Daemonsets to be successfully rolled out.
kubectl rollout status daemonset --namespace nvidia-network-
operator kube-multus-ds sriov-network-config-daemon sriov-device-
plugin
## Expect the network injector to be successfully rolled out.
kubectl rollout status deployment --namespace ovn-kubernetes ovn-
kubernetes-resource-injector
```

5. DPU Provisioning and Service Installation

In this step we deploy our DPUs and the services that will run on them.

The user is expected to create a DPUDeployment object that reflects a set of DPUServices that should run on a set of DPUs.

> If you want to learn more about `DPUDeployments`, feel free to check the DPUDeployment documentation.

Create the DPUDeployment, DPUServiceConfig, DPUServiceTemplate and other necessary objects

> ⚠ **Warning**
>
> In case more than 1 DPU exists per node, the relevant selector should be applied in the DPUDeployment to select the appropriate DPU. See DPUDeployment - DPUs Configuration to understand more about the selectors.

A number of environment variables must be set before running this command.

```
cat manifests/05-dpudeployment-installation/*.yaml | envsubst |
kubectl apply -f -
```

This will deploy the following objects:

BFB to download Bluefield Bitstream to a shared volume

```
---
apiVersion: provisioning.dpu.nvidia.com/v1alpha1
kind: BFB
metadata:
  name: bf-bundle-$TAG
  namespace: dpf-operator-system
spec:
  url: $BFB_URL
```

HBN OVN DPUFlavor to correctly configure the DPUs on provisioning

```
---
apiVersion: provisioning.dpu.nvidia.com/v1alpha1
kind: DPUFlavor
metadata:
  name: hbn-ovnk-$TAG
  namespace: dpf-operator-system
spec:
  grub:
    kernelParameters:
      - console=hvc0
      - console=ttyAMA0
      - earlycon=pl011,0x13010000
      - fixrttc
      - net.ifnames=0
      - biosdevname=0
      - iommu.passthrough=1
      - cgroup_no_v1=net_prio,net_cls
      - hugepagesz=2048kB
      - hugepages=3072
  nvconfig:
    - device: "*"
      parameters:
        - PF_BAR2_ENABLE=0
        - PER_PF_NUM_SF=1
        - PF_TOTAL_SF=20
        - PF_SF_BAR_SIZE=10
        - NUM_PF_MSIX_VALID=0
        - PF_NUM_PF_MSIX_VALID=1
        - PF_NUM_PF_MSIX=228
        - INTERNAL_CPU_MODEL=1
        - INTERNAL_CPU_OFFLOAD_ENGINE=0
        - SRIOV_EN=1
        - NUM_OF_VFS=46
        - LAG_RESOURCE_ALLOCATION=1
        - LINK_TYPE_P1=ETH
```

```
        - LINK_TYPE_P2=ETH
  ovs:
    rawConfigScript: |
      _ovs-vsctl() {
        ovs-vsctl --no-wait --timeout 15 "$@"
      }

      _ovs-vsctl set Open_vSwitch . other_config:doca-init=true
      _ovs-vsctl set Open_vSwitch . other_config:dpdk-max-
memzones=50000
      _ovs-vsctl set Open_vSwitch . other_config:hw-offload=true
      _ovs-vsctl set Open_vSwitch . other_config:pmd-quiet-
idle=true
      _ovs-vsctl set Open_vSwitch . other_config:max-idle=20000
      _ovs-vsctl set Open_vSwitch . other_config:max-
revalidator=5000
      _ovs-vsctl set Open_vSwitch . other_config:ctl-pipe-
size=1024
      _ovs-vsctl --if-exists del-br ovsbr1
      _ovs-vsctl --if-exists del-br ovsbr2
      _ovs-vsctl --may-exist add-br br-sfc
      _ovs-vsctl set bridge br-sfc datapath_type=netdev
      _ovs-vsctl set bridge br-sfc fail_mode=secure
      _ovs-vsctl --may-exist add-br br-hbn
      _ovs-vsctl set bridge br-hbn datapath_type=netdev
      _ovs-vsctl set bridge br-hbn fail_mode=secure
      _ovs-vsctl --may-exist add-port br-sfc p0
      _ovs-vsctl set Interface p0 type=dpdk
      _ovs-vsctl set Interface p0 mtu_request=9216
      _ovs-vsctl set Port p0 external_ids:dpf-type=physical

      _ovs-vsctl set Open_vSwitch . external-ids:ovn-bridge-
datapath-type=netdev
      _ovs-vsctl --may-exist add-br br-ovn
      _ovs-vsctl set bridge br-ovn datapath_type=netdev
      _ovs-vsctl br-set-external-id br-ovn bridge-id br-ovn
```

```
      _ovs-vsctl br-set-external-id br-ovn bridge-uplink
puplinkbrovntobrsfc
      _ovs-vsctl --may-exist add-port br-ovn pf0hpf
      _ovs-vsctl set Interface pf0hpf type=dpdk
      _ovs-vsctl set Interface pf0hpf mtu_request=9216

  bfcfgParameters:
    - UPDATE_ATF_UEFI=yes
    - UPDATE_DPU_OS=yes
    - WITH_NIC_FW_UPDATE=yes

  hostNetworkInterfaceConfigs:
    - portNumber: 0
      dhcp: true
      mtu: 1500

  configFiles:
  - path: /etc/mellanox/mlnx-bf.conf
    operation: override
    raw: |
        ALLOW_SHARED_RQ="no"
        IPSEC_FULL_OFFLOAD="no"
        ENABLE_ESWITCH_MULTIPORT="yes"
    permissions: "0644"
  - path: /etc/mellanox/mlnx-ovs.conf
    operation: override
    raw: |
        CREATE_OVS_BRIDGES="no"
        OVS_DOCA="yes"
    permissions: "0644"
  - path: /etc/mellanox/mlnx-sf.conf
    operation: override
    raw: ""
    permissions: "0644"
```

DPUDeployment to provision DPUs on worker nodes

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUDeployment
metadata:
  name: ovn-hbn
  namespace: dpf-operator-system
spec:
  dpus:
    bfb: bf-bundle-$TAG
    flavor: hbn-ovnk-$TAG
    dpuSets:
    - nameSuffix: "dpuset1"
      nodeSelector:
        matchLabels:
          feature.node.kubernetes.io/dpu-enabled: "true"
      dpuSelector:
        provisioning.dpu.nvidia.com/dpudevice-pf0-name: $DPU_P0
  services:
    ovn:
      serviceTemplate: ovn
      serviceConfiguration: ovn
    hbn:
      serviceTemplate: hbn
      serviceConfiguration: hbn
    dts:
      serviceTemplate: dts
      serviceConfiguration: dts
    blueman:
      serviceTemplate: blueman
      serviceConfiguration: blueman
  serviceChains:
    switches:
      - ports:
        - serviceInterface:
            matchLabels:
```

```
            uplink: p0
        - service:
            name: hbn
            interface: p0_if
      - ports:
        - serviceInterface:
            matchLabels:
              uplink: p1
        - service:
            name: hbn
            interface: p1_if
      - ports:
        - serviceInterface:
            matchLabels:
              port: ovn
        - service:
            name: hbn
            interface: pf2dpu2_if
```

OVN DPUServiceConfig and DPUServiceTemplate to deploy OVN workloads to the DPUs

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceConfiguration
metadata:
  name: ovn
  namespace: dpf-operator-system
spec:
  deploymentServiceName: "ovn"
  serviceConfiguration:
    helmChart:
      values:
        k8sAPIServer:
https://$TARGETCLUSTER_API_SERVER_HOST:$TARGETCLUSTER_API_SERVER_PORT
        podNetwork: $POD_CIDR/24
        serviceNetwork: $SERVICE_CIDR
        dpuManifests:
          kubernetesSecretName: "ovn-dpu" # user needs to populate
based on DPUServiceCredentialRequest
          vtepCIDR: "10.0.120.0/22" # user needs to populate based on
DPUServiceIPAM
          hostCIDR: $TARGETCLUSTER_NODE_CIDR # user needs to
populate
          ipamPool: "pool1" # user needs to populate based on
DPUServiceIPAM
          ipamPoolType: "cidrpool" # user needs to populate based on
DPUServiceIPAM
          ipamVTEPIPIndex: 0
          ipamPFIPIndex: 1
```

```yaml
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceTemplate
metadata:
  name: ovn
  namespace: dpf-operator-system
spec:
  deploymentServiceName: "ovn"
  helmChart:
    source:
      repoURL: $OVN_KUBERNETES_REPO_URL
      chart: ovn-kubernetes-chart
      version: $TAG
    values:
      commonManifests:
        enabled: true
      dpuManifests:
        enabled: true
      leaseNamespace: "ovn-kubernetes"
      gatewayOpts: "--gateway-interface=br-ovn"
```

HBN DPUServiceConfig and DPUServiceTemplate to deploy HBN workloads to the DPUs

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceConfiguration
metadata:
  name: hbn
  namespace: dpf-operator-system
spec:
  deploymentServiceName: "hbn"
  serviceConfiguration:
    serviceDaemonSet:
      annotations:
        k8s.v1.cni.cncf.io/networks: |-
          [
          {"name": "iprequest", "interface": "ip_lo", "cni-args": {"poolNames":
["loopback"], "poolType": "cidrpool"}},
          {"name": "iprequest", "interface": "ip_pf2dpu2", "cni-args": {"poolNames":
["pool1"], "poolType": "cidrpool", "allocateDefaultGateway": true}}
          ]
    helmChart:
      values:
        configuration:
          perDPUValuesYAML: |
            - hostnamePattern: "*"
              values:
                bgp_peer_group: hbn
            - hostnamePattern: "worker1*"
              values:
                bgp_autonomous_system: 65101
            - hostnamePattern: "worker2*"
              values:
                bgp_autonomous_system: 65201
          startupYAMLJ2: |
            - header:
                model: BLUEFIELD
                nvue-api-version: nvue_v1
```

```
                rev-id: 1.0
                version: HBN 2.4.0
            - set:
                interface:
                  lo:
                    ip:
                      address:
                        {{ ipaddresses.ip_lo.ip }}/32: {}
                    type: loopback
                  p0_if,p1_if:
                    type: swp
                    link:
                      mtu: 9000
                  pf2dpu2_if:
                    ip:
                      address:
                        {{ ipaddresses.ip_pf2dpu2.cidr }}: {}
                    type: swp
                    link:
                      mtu: 9000
                router:
                  bgp:
                    autonomous-system: {{
config.bgp_autonomous_system }}
                    enable: on
                    graceful-restart:
                      mode: full
                    router-id: {{ ipaddresses.ip_lo.ip }}
                vrf:
                  default:
                    router:
                      bgp:
                        address-family:
                          ipv4-unicast:
                            enable: on
                            redistribute:
```

```
                                connected:
                                  enable: on
                            ipv6-unicast:
                              enable: on
                              redistribute:
                                connected:
                                  enable: on
                          enable: on
                          neighbor:
                            p0_if:
                              peer-group: {{ config.bgp_peer_group
}}
                              type: unnumbered
                            p1_if:
                              peer-group: {{ config.bgp_peer_group
}}
                              type: unnumbered
                          path-selection:
                            multipath:
                              aspath-ignore: on
                          peer-group:
                            {{ config.bgp_peer_group }}:
                              remote-as: external

  interfaces:
    ## NOTE: Interfaces inside the HBN pod must have the `_if`
suffix due to a naming convention in HBN.
  - name: p0_if
    network: mybrhbn
  - name: p1_if
    network: mybrhbn
  - name: pf2dpu2_if
    network: mybrhbn
```

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceTemplate
metadata:
  name: hbn
  namespace: dpf-operator-system
spec:
  deploymentServiceName: "hbn"
  helmChart:
    source:
      repoURL: $HELM_REGISTRY_REPO_URL
      version: 1.0.5
      chart: doca-hbn
    values:
      image:
        repository: $HBN_NGC_IMAGE_URL
        tag: 3.2.1-doca3.2.1
      resources:
        memory: 6Gi
        nvidia.com/bf_sf: 3
```

DOCA Telemetry Service DPUServiceConfig and DPUServiceTemplate to deploy DTS to the DPUs

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceConfiguration
metadata:
  name: dts
  namespace: dpf-operator-system
spec:
  deploymentServiceName: "dts"
```

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceTemplate
metadata:
  name: dts
  namespace: dpf-operator-system
spec:
  deploymentServiceName: "dts"
  helmChart:
    source:
      repoURL: $HELM_REGISTRY_REPO_URL
      version: 1.23.4
      chart: doca-telemetry
```

Blueman DPUServiceConfig and DPUServiceTemplate to deploy Blueman to the DPUs

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceConfiguration
metadata:
  name: blueman
  namespace: dpf-operator-system
spec:
  deploymentServiceName: "blueman"
```

```yaml
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceTemplate
metadata:
  name: blueman
  namespace: dpf-operator-system
spec:
  deploymentServiceName: "blueman"
  helmChart:
    source:
      repoURL: $HELM_REGISTRY_REPO_URL
      version: 1.0.8
      chart: doca-blueman
```

OVN DPUServiceCredentialRequest to allow cross cluster communication

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceCredentialRequest
metadata:
  name: ovn-dpu
  namespace: dpf-operator-system
spec:
  serviceAccount:
    name: ovn-dpu
    namespace: dpf-operator-system
  duration: 24h
  type: tokenFile
  secret:
    name: ovn-dpu
    namespace: dpf-operator-system
  metadata:
    labels:
      dpu.nvidia.com/image-pull-secret: ""
```

DPUServiceInterfaces for physical ports on the DPU

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceInterface
metadata:
  name: p0
  namespace: dpf-operator-system
spec:
  template:
    spec:
      template:
        metadata:
          labels:
            uplink: "p0"
        spec:
          interfaceType: physical
          physical:
            interfaceName: p0
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceInterface
metadata:
  name: p1
  namespace: dpf-operator-system
spec:
  template:
    spec:
      template:
        metadata:
          labels:
            uplink: "p1"
        spec:
          interfaceType: physical
          physical:
```

```
            interfaceName: p1
```

OVN DPUServiceInterface to define the ports attached to OVN workloads on the DPU

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceInterface
metadata:
  name: ovn
  namespace: dpf-operator-system
spec:
  template:
    spec:
      template:
        metadata:
          labels:
            port: ovn
        spec:
          interfaceType: ovn
```

DPUServiceIPAM to set up IP Address Management on the DPUCluster

```yaml
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceIPAM
metadata:
  name: pool1
  namespace: dpf-operator-system
spec:
  ipv4Network:
    network: "10.0.120.0/22"
    gatewayIndex: 3
    prefixSize: 29
```

DPUServiceIPAM for the loopback interface in HBN

```yaml
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceIPAM
metadata:
  name: loopback
  namespace: dpf-operator-system
spec:
  ipv4Network:
    network: "11.0.0.0/24"
    prefixSize: 32
```

Verification

These verification commands may need to be run multiple times to ensure the condition is met.

Note that the DPUService name will have a random suffix. For example, `ovn-hbn-doca-hbn-l2xsl`. Use the correct name for the verification.

Verify the DPU and Service installation with:

```
## Ensure the DPUServices are created and have been reconciled.
kubectl wait --for=condition=ApplicationsReconciled --namespace
dpf-operator-system dpuservices -l svc.dpu.nvidia.com/owned-by-
dpudeployment=dpf-operator-system_ovn-hbn
## Ensure the DPUServiceIPAMs have been reconciled
kubectl wait --for=condition=DPUIPAMObjectReconciled --namespace
dpf-operator-system dpuserviceipam --all
## Ensure the DPUServiceInterfaces have been reconciled
kubectl wait --for=condition=ServiceInterfaceSetReconciled --
namespace dpf-operator-system dpuserviceinterface --all
## Ensure the DPUServiceChains have been reconciled
kubectl wait --for=condition=ServiceChainSetReconciled --namespace
dpf-operator-system dpuservicechain --all
```

6. Test Traffic

Add worker nodes to the cluster

At this point workers should be added to the cluster. Each worker node should be configured in line with the prerequisites. As workers are added to the cluster DPUs will be provisioned and DPUServices will begin to be spun up.

You can verify the status of the DPUDeployment and its components with the following command:

```
$ kubectl -n dpf-operator-system exec deploy/dpf-operator-
controller-manager -- /dpfctl describe dpudeployments
NAME                                          NAMESPACE
STATUS        REASON     SINCE  MESSAGE
DPFOperatorConfig/dpfoperatorconfig           dpf-operator-system
Ready: True   Success    2h
  DPUDeployments
    DPUDeployment/ovn-hbn                     dpf-operator-system
Ready: True   Success    2h
      DPUServiceChains
        DPUServiceChain/ovn-hbn-wkdhz         dpf-operator-system
Ready: True   Success    2h
      DPUSets
        DPUSet/hbn-only-dpuset1               dpf-operator-system
          BFB/bf-bundle                       dpf-operator-system
Ready: True   Ready      2h     File: bf-bundle-3.2.1-
34_25.11_ubuntu-24.04_64k_prod.bfb, DOCA: 3.2.1
          DPUs
            2 DPUs...                         dpf-operator-system
Ready: True   DPUReady   2h     See dpu-node-mt2310xz03lr-
mt2310xz03lr, dpu-node-mt2310xz03m2-mt2310xz03m2
      Services
        DPUServiceTemplates
          4 DPUServiceTemplates...            dpf-operator-system
Ready: True   Success    2h     See ovn-hbn-blueman, ovn-hbn-dts,
ovn-hbn-hbn, ovn-hbn-ovn
        DPUServices
          4 DPUServices...                    dpf-operator-system
Ready: True   Success    2h     See ovn-hbn-blueman-4tlmp, ovn-
hbn-dts-49282, ovn-hbn-hbn-hd92m, ovn-hbn-ovn-l2xsl
```

Deploy test pods

```
kubectl apply -f manifests/06-test-traffic
```

HBN and OVN functionality can be tested by pinging between the pods and services deployed in the default namespace.

Uninstall

This section describes how to clean up the cluster after the DPF setup has been completed. It is important to follow the steps in the correct order to ensure that all components are removed cleanly and that the cluster remains functional.

Delete the test pods

```
kubectl delete -f manifests/06-test-traffic --wait
```

Delete DPF CNI acceleration components

```
kubectl delete -f manifests/04-enable-accelerated-cni --wait
helm uninstall -n nvidia-network-operator network-operator --wait

## Note: Uninstalling OVN Kubernetes as primary CNI is not
supported but this command must be run to remove the webhook and
restore a functioning cluster.
helm uninstall -n ovn-kubernetes ovn-kubernetes-resource-injector
--wait
```

Delete the DPF Operator system and DPF Operator

First we have to delete some DPUServiceInterfaces. This is necessary because of a known issue during uninstallation.

```
kubectl delete -n dpf-operator-system dpuserviceinterface p0 p1
ovn --wait
```

Then we can delete the config and system namespace.

```
kubectl delete -n dpf-operator-system dpfoperatorconfig
dpfoperatorconfig --wait
helm uninstall -n dpf-operator-system dpf-operator --wait
```

Delete DPF Operator PVC

```
kubectl -n dpf-operator-system delete pvc bfb-pvc
kubectl delete pv bfb-pv
```

Note: there can be a race condition with deleting the underlying Kamaji cluster which runs the DPU cluster control plane in this guide. If that happens it may be necessary to remove finalizers manually from `DPUCluster` and `Datastore` objects.

Limitations of DPF Setup

Host network pod services

The Kubelet process on the Kubernetes nodes use the OOB interface IP address to register in Kubernetes. This means that the nodes have the OOB IP addresses as node IP addresses. This means that pods using host networking have the OOB IP address of the hosts as pod IP address. However, that interface is not accelerated. This means that any

component using the addresses of the pods using host networking will not benefit from hardware acceleration and high-speed ports.

For example, this means that when creating a Kubernetes NodePort service selecting pods using host networking, even if the user uses the high-speed IP of the host, the traffic will not be accelerated. In order to solve this, it is possible to create dedicated endpointSlices that contain the host high-speed port IP addresses instead of OOB port IP addresses. This way, the entire path to the pods will be accelerated and benefit from high performances, if the user uses the high speed IP address of the host with the nodePort port. This requires the workload running on the pod with host networking to also listen on the high-speed port IP address.

OVN Kubernetes with Host Based Networking and SNAP Block Storage

> **Note**
>
> Follow this guide from the source GitHub repo at github.com/NVIDIA/doca-platform and moving to the `docs/public/user-guides/host-trusted/use-cases/hbn-ovnk-snap/README.md` for better formatting of the code.

In this configuration OVN Kubernetes is offloaded to the DPU and combined with NVIDIA Host Based Networking (HBN) and DOCA SNAP for Block (NVMe) storage.

Prerequisites

This guide should be run by cloning the repo from github.com/NVIDIA/doca-platform and moving to the `docs/public/user-guides/host-trusted/use-cases/hbn-ovnk-snap` directory.

The system is set up as described in the system prerequisites. The OVN Kubernetes with HBN and SNAP Block Storage use case has the additional requirements:

DPU Prerequisites

- Bluefield 3 with 32GB of RAM

Infrastructure Prerequisites

- A remote SPDK target should be set up to provide persistent storage for SNAP Block Storage
- The SPDK target should be reachable from the DPUs
- The management interface of the SPDK target should be reachable from the control plane nodes
- Make sure to check Host OS Configuration Section in SNAP service documentation to validate the host OS configuration on the worker nodes

Software Prerequisites

This guide uses the following tools which must be installed on the machine where the commands contained in this guide run.

- kubectl
- helm
- envsubst

Network Prerequisites

Control Plane Nodes

- Open vSwitch (OVS) packages installed - i.e. `openvswitch-switch` for Ubuntu 24.04

- out-of-band management port should be configured as OVS bridge port with "bridge-uplink" OVS metadata This addresses a known issue.
- DNS stub resolver should be disabled if using systemd resolvd

Worker Nodes

- Open vSwitch (OVS) packages not installed
- Host high-speed port (Host PF0) must have DHCP enabled
- MTU of the port should be statically set to 1500
- **Note**: These settings can be configured automatically via DPUFlavor `hostNetworkInterfaceConfigs`

Kubernetes Prerequisites

- CNI not installed
- kube-proxy not installed
- coreDNS should be configured to run only on control plane nodes - e.g. using NodeAffinity. This addresses a known issue.
- control plane setup is complete before starting this guide
- worker nodes are not added until indicated by this guide

Control Plane Nodes

- Have the labels:
  - `"k8s.ovn.org/zone-name": $KUBERNETES_NODE_NAME`

Worker Nodes

- Have the labels:
  - `"k8s.ovn.org/dpu-host": ""`
  - `"k8s.ovn.org/zone-name": $KUBERNETES_NODE_NAME`
- Have the annotations:
  - `"k8s.ovn.org/remote-zone-migrated": $KUBERNETES_NODE_NAME`

Virtual Functions

A number of virtual functions (VFs) will be created on hosts when provisioning DPUs. Certain of these VFs are marked for specific usage:

- The first VF (vf0) is used by provisioning components.
- The second VF (vf1) is used by ovn-kubernetes.
- The remaining VFs are allocated by SR-IOV Device Plugin. Each pod using OVN Kubernetes in DPU mode as its primary CNI will have one of these VFs injected at Pod creation time.

Installation Guide

0. Required Variables

The following variables are required by this guide. A sensible default is provided where it makes sense, but many will be specific to the target infrastructure.

Commands in this guide are run in the same directory that contains this readme.

Environment variables file

```
## IP Address for the Kubernetes API server of the target cluster
on which DPF is installed.
## This should never include a scheme or a port.
## e.g. 10.10.10.10
export TARGETCLUSTER_API_SERVER_HOST=

## Port for the Kubernetes API server of the target cluster on
which DPF is installed.
export TARGETCLUSTER_API_SERVER_PORT=6443

## IP address range for hosts in the target cluster on which DPF
is installed.
## This is a CIDR in the form e.g. 10.10.10.0/24
export TARGETCLUSTER_NODE_CIDR=

## Virtual IP used by the load balancer for the DPU Cluster. Must
be a reserved IP from the management subnet and not allocated by
DHCP.
export DPUCLUSTER_VIP=

## DPU_P0 is the name of the first port of the DPU. This name
must be the same on all worker nodes.
export DPU_P0=

## DPU_P0_VF1 is the name of the second Virtual Function (VF) of
the first port of the DPU. This name must be the same on all
worker nodes.
## Note: The VF will be created after the DPU is provisioned and
the phase "Host Network Configuration" is completed.
export DPU_P0_VF1=

## Interface on which the DPUCluster load balancer will listen.
Should be the management interface of the control plane node.
export DPUCLUSTER_INTERFACE=
```

```
## IP address to the NFS server used as storage for the BFB.
export NFS_SERVER_IP=

## The repository URL for the NVIDIA Helm chart registry.
## Usually this is the NVIDIA Helm NGC registry. For development
purposes, this can be set to a different repository.
export HELM_REGISTRY_REPO_URL=https://helm.ngc.nvidia.com/nvidia/doca

## The repository URL for the HBN container image.
## Usually this is the NVIDIA NGC registry. For development
purposes, this can be set to a different repository.
export HBN_NGC_IMAGE_URL=nvcr.io/nvidia/doca/doca_hbn

## The repository URL for the SNAP VFS container image.
## Usually this is the NVIDIA NGC registry. For development
purposes, this can be set to a different repository.
export SNAP_NGC_IMAGE_URL=nvcr.io/nvidia/doca/doca_vfs

## The repository URL for the OVN-Kubernetes Helm chart.
## Usually this is the NVIDIA GHCR repository. For development
purposes, this can be set to a different repository.
export OVN_KUBERNETES_REPO_URL=oci://ghcr.io/nvidia

## POD_CIDR is the CIDR used for pods in the target Kubernetes
cluster.
export POD_CIDR=10.233.64.0/18

## SERVICE_CIDR is the CIDR used for services in the target
Kubernetes cluster.
## This is a CIDR in the form e.g. 10.10.10.0/24
export SERVICE_CIDR=10.233.0.0/18

## The DPF REGISTRY is the Helm repository URL where the DPF
Operator Chart resides.
## Usually this is the NVIDIA Helm NGC registry. For development
purposes, this can be set to a different repository.
```

```
export REGISTRY=https://helm.ngc.nvidia.com/nvidia/doca

## The DPF TAG is the version of the DPF components which will be deployed in this guide.
export TAG=v25.10.0

## URL to the BFB used in the `bfb.yaml` and linked by the DPUSet.
export BFB_URL="https://content.mellanox.com/BlueField/BFBs/Ubuntu24.04/bf-bundle-3.2.1-34_25.11_ubuntu-24.04_64k_prod.bfb"
```

Modify the variables in `manifests/00-env-vars/envvars.env` to fit your environment, then source the file:

```
source manifests/00-env-vars/envvars.env
```

1. CNI Installation

OVN Kubernetes is used as the primary CNI for the cluster. On worker nodes the primary CNI will be accelerated by offloading work to the DPU. On control plane nodes OVN Kubernetes will run without offloading.

Create the Namespace

```
kubectl create ns ovn-kubernetes
```

Install OVN Kubernetes from the helm chart

Install the OVN Kubernetes CNI components from the helm chart. A number of environment variables must be set before running this command.

```
envsubst < manifests/01-cni-installation/helm-values/ovn-
kubernetes.yml | helm upgrade --install -n ovn-kubernetes ovn-
kubernetes ${OVN_KUBERNETES_REPO_URL}/ovn-kubernetes-chart --
version $TAG --values -
```

OVN-Kubernetes Helm values

```
commonManifests:
  enabled: true
nodeWithoutDPUManifests:
  enabled: true
controlPlaneManifests:
  enabled: true
nodeWithDPUManifests:
  enabled: true
  nodeMgmtPortNetdev: $DPU_P0_VF1
  dpuServiceAccountNamespace: dpf-operator-system
gatewayOpts: --gateway-interface=$DPU_P0
## Note this CIDR is followed by a trailing /24 which informs OVN
Kubernetes on how to split the CIDR per node.
podNetwork: $POD_CIDR/24
serviceNetwork: $SERVICE_CIDR
k8sAPIServer:
https://$TARGETCLUSTER_API_SERVER_HOST:$TARGETCLUSTER_API_SERVER_PORT
```

Verification

These verification commands may need to be run multiple times to ensure the condition is met.

Verify the CNI installation with:

```
## Ensure all nodes in the cluster are ready.
kubectl wait --for=condition=ready nodes --all
## Ensure all pods in the ovn-kubernetes namespace are ready.
kubectl wait --for=condition=ready --namespace ovn-kubernetes pods
--all --timeout=300s
```

2. DPF Operator Installation

Create storage required by the DPF Operator

A number of environment variables must be set before running this command.

```
kubectl create ns dpf-operator-system
cat manifests/02-dpf-operator-installation/*.yaml | envsubst |
kubectl apply -f -
```

This deploys the following objects:

PersistentVolume and PersistentVolumeClaim for the provisioning controller

```
---
apiVersion: v1
kind: PersistentVolume
metadata:
  name: bfb-pv
spec:
  capacity:
    storage: 10Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteMany
  nfs:
    path: /mnt/dpf_share/bfb
    server: $NFS_SERVER_IP
  persistentVolumeReclaimPolicy: Delete
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: bfb-pvc
  namespace: dpf-operator-system
spec:
  accessModes:
  - ReadWriteMany
  resources:
    requests:
      storage: 10Gi
  volumeMode: Filesystem
  storageClassName: ""
```

Additional Dependencies

Before deploying the DPF Operator, ensure that Helm is properly configured according to the Helm prerequisites.

> **Warning**
>
> This is a critical prerequisite step that must be completed for the DPF Operator to function properly.

After applying the additional dependencies you MUST ensure that the KUBERNETES_SERVICE_HOST and KUBERNETES_SERVICE_PORT environment variables are set in the node-feature-discovery-worker DaemonSet.

NFD needs to target the VIP because it needs to be up before cluster services can work.

Example commands to set the environment variables:

```
kubectl -n dpf-operator-system set env daemonset/node-feature-
discovery-worker \
KUBERNETES_SERVICE_HOST=$TARGETCLUSTER_API_SERVER_HOST \
KUBERNETES_SERVICE_PORT=$TARGETCLUSTER_API_SERVER_PORT
```

Deploy the DPF Operator

A number of environment variables must be set before running this command.

HTTP Registry (default)

If the $REGISTRY is an HTTP Registry (default value) use this command:

```
helm repo add --force-update dpf-repository ${REGISTRY}
helm repo update
helm upgrade --install -n dpf-operator-system dpf-operator dpf-
repository/dpf-operator --version=$TAG
```

OCI Registry

For development purposes, if the $REGISTRY is an OCI Registry use this command:

```
helm upgrade --install -n dpf-operator-system dpf-operator
$REGISTRY/dpf-operator --version=$TAG
```

Verification

These verification commands may need to be run multiple times to ensure the condition is met.

Verify the DPF Operator installation with:

```
## Ensure the DPF Operator deployment is available.
kubectl rollout status deployment --namespace dpf-operator-system
dpf-operator-controller-manager
## Ensure all pods in the DPF Operator system are ready.
kubectl wait --for=condition=ready --namespace dpf-operator-system
pods --all
```

3. DPF System Installation

This section involves creating the DPF system components and some basic infrastructure required for a functioning DPF-enabled cluster.

Deploy the DPF System components

A number of environment variables must be set before running this command.

```
kubectl create ns dpu-cplane-tenant1
cat manifests/03-dpf-system-installation/*.yaml | envsubst |
kubectl apply -f -
```

This will create the following objects:

DPFOperatorConfig to install the DPF System components

```
---
apiVersion: operator.dpu.nvidia.com/v1alpha1
kind: DPFOperatorConfig
metadata:
  name: dpfoperatorconfig
  namespace: dpf-operator-system
spec:
  overrides:
    kubernetesAPIServerVIP: $TARGETCLUSTER_API_SERVER_HOST
    kubernetesAPIServerPort: $TARGETCLUSTER_API_SERVER_PORT
  provisioningController:
    bfbPVCName: "bfb-pvc"
    dmsTimeout: 900
  kamajiClusterManager:
    disable: false
```

DPUCluster to serve as Kubernetes control plane for DPU nodes

```
---
apiVersion: provisioning.dpu.nvidia.com/v1alpha1
kind: DPUCluster
metadata:
  name: dpu-cplane-tenant1
  namespace: dpu-cplane-tenant1
spec:
  type: kamaji
  maxNodes: 10
  clusterEndpoint:
    # deploy keepalived instances on the nodes that match the
given nodeSelector.
    keepalived:
      # interface on which keepalived will listen. Should be the oob
interface of the control plane node.
      interface: $DPUCLUSTER_INTERFACE
      # Virtual IP reserved for the DPU Cluster load balancer.
Must not be allocatable by DHCP.
      vip: $DPUCLUSTER_VIP
      # virtualRouterID must be in range [1,255], make sure the
given virtualRouterID does not duplicate with any existing
keepalived process running on the host
      virtualRouterID: 126
      nodeSelector:
        node-role.kubernetes.io/control-plane: ""
```

Verification

These verification commands may need to be run multiple times to ensure the condition is met.

Verify the DPF System with:

```
## Ensure the provisioning and DPUService controller manager
deployments are available.
kubectl rollout status deployment --namespace dpf-operator-system
dpf-provisioning-controller-manager dpuservice-controller-manager
## Ensure all other deployments in the DPF Operator system are
Available.
kubectl rollout status deployment --namespace dpf-operator-system
## Ensure the DPUCluster is ready for nodes to join.
kubectl wait --for=condition=ready --namespace dpu-cplane-tenant1
dpucluster --all
```

4. Install Components to Enable Accelerated CNI Nodes

OVN Kubernetes will accelerate traffic by attaching a VF to each pod using the primary CNI. This VF is used to offload flows to the DPU. This section details the components needed to connect pods to the offloaded OVN Kubernetes CNI.

Install Multus and SRIOV Network Operator using NVIDIA Network Operator

```
helm repo add nvidia https://helm.ngc.nvidia.com/nvidia --force-update
helm upgrade --no-hooks --install --create-namespace --namespace
nvidia-network-operator network-operator nvidia/network-operator
--version 25.7.0 -f ./manifests/04-enable-accelerated-cni/helm-
values/network-operator.yml
```

NVIDIA Network Operator Helm values

```yaml
nfd:
  enabled: false
  deployNodeFeatureRules: false
sriovNetworkOperator:
  enabled: true
sriov-network-operator:
  operator:
    affinity:
      nodeAffinity:
        requiredDuringSchedulingIgnoredDuringExecution:
          nodeSelectorTerms:
            - matchExpressions:
                - key: node-role.kubernetes.io/master
                  operator: Exists
            - matchExpressions:
                - key: node-role.kubernetes.io/control-plane
                  operator: Exists
  crds:
    enabled: true
  sriovOperatorConfig:
    deploy: true
    configDaemonNodeSelector: null
operator:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
          - matchExpressions:
              - key: node-role.kubernetes.io/master
                operator: Exists
          - matchExpressions:
              - key: node-role.kubernetes.io/control-plane
                operator: Exists
```

Install the OVN Kubernetes resource injection webhook

The OVN Kubernetes resource injection webhook injected each pod scheduled to a worker node with a request for a VF and a Network Attachment Definition. This webhook is part of the same helm chart as the other components of the OVN Kubernetes CNI. Here it is installed by adjusting the existing helm installation to add the webhook component to the installation.

```
envsubst < manifests/04-enable-accelerated-cni/helm-values/ovn-
kubernetes.yml | helm upgrade --install -n ovn-kubernetes ovn-
kubernetes-resource-injector ${OVN_KUBERNETES_REPO_URL}/ovn-
kubernetes-chart --version $TAG --values -
```

OVN Kubernetes Resource Injector Helm values

```
ovn-kubernetes-resource-injector:
  ## Enable the ovn-kubernetes-resource-injector
  enabled: true
```

Apply the NICClusterConfiguration and SriovNetworkNodePolicy

```
cat manifests/04-enable-accelerated-cni/*.yaml | envsubst |
kubectl apply -f -
```

This will deploy the following objects:

NICClusterPolicy for the NVIDIA Network Operator

```yaml
---
apiVersion: mellanox.com/v1alpha1
kind: NicClusterPolicy
metadata:
  name: nic-cluster-policy
spec:
  secondaryNetwork:
    multus:
      image: multus-cni
      imagePullSecrets: []
      repository: ghcr.io/k8snetworkplumbingwg
      version: v3.9.3
```

SriovNetworkNodePolicy for the SR-IOV Network Operator

```yaml
---
apiVersion: sriovnetwork.openshift.io/v1
kind: SriovNetworkNodePolicy
metadata:
  name: bf3-p0-vfs
  namespace: nvidia-network-operator
spec:
  nicSelector:
    deviceID: "a2dc"
    vendor: "15b3"
    pfNames:
    - $DPU_P0#2-45
  nodeSelector:
    node-role.kubernetes.io/worker: ""
  numVfs: 46
  resourceName: bf3-p0-vfs
  isRdma: true
  externallyManaged: true
  deviceType: netdevice
  linkType: eth
```

Verification

These verification commands may need to be run multiple times to ensure the condition is met.

Verify that the accelerated CNI is enabled with:

```
## Ensure the nvidia-network-operator pods are ready.
kubectl wait --for=condition=Ready --namespace nvidia-network-
operator pods --all
## Expect the following Daemonsets to be successfully rolled out.
kubectl rollout status daemonset --namespace nvidia-network-
operator kube-multus-ds sriov-network-config-daemon sriov-device-
plugin
## Expect the network injector to be successfully rolled out.
kubectl rollout status deployment --namespace ovn-kubernetes ovn-
kubernetes-resource-injector
```

5. DPU Provisioning and Service Installation

In this step we deploy our DPUs and the services that will run on them.

The user is expected to create a DPUDeployment object that reflects a set of DPUServices that should run on a set of DPUs.

> If you want to learn more about `DPUDeployments`, feel free to check the DPUDeployment documentation.

Create the DPUDeployment, DPUServiceConfig, DPUServiceTemplate and other necessary objects

> ⚠ **Warning**
>
> In case more than 1 DPU exists per node, the relevant selector should be applied in the DPUDeployment to select the appropriate DPU. See DPUDeployment - DPUs Configuration to understand more about the selectors.

A number of environment variables must be set before running this command.

```
cat manifests/05-dpudeployment-installation/*.yaml | envsubst |
kubectl apply -f -
```

This will deploy the following objects:

BFB to download Bluefield Bitstream to a shared volume

```
---
apiVersion: provisioning.dpu.nvidia.com/v1alpha1
kind: BFB
metadata:
  name: bf-bundle-$TAG
  namespace: dpf-operator-system
spec:
  url: $BFB_URL
```

HBN OVN SNAP DPUFlavor to correctly configure the DPUs on provisioning

```
---
apiVersion: provisioning.dpu.nvidia.com/v1alpha1
kind: DPUFlavor
metadata:
  name: hbn-ovnk-snap-nvme-$TAG
  namespace: dpf-operator-system
spec:
  grub:
    kernelParameters:
      - console=hvc0
      - console=ttyAMA0
      - earlycon=pl011,0x13010000
      - fixrttc
      - net.ifnames=0
      - biosdevname=0
      - iommu.passthrough=1
      - cgroup_no_v1=net_prio,net_cls
      - hugepagesz=2048kB
      - hugepages=5120
  nvconfig:
    - device: "*"
      parameters:
        - PF_BAR2_ENABLE=0
        - PER_PF_NUM_SF=1
        - PF_TOTAL_SF=20
        - PF_SF_BAR_SIZE=10
        - NUM_PF_MSIX_VALID=0
        - PF_NUM_PF_MSIX_VALID=1
        - PF_NUM_PF_MSIX=228
        - INTERNAL_CPU_MODEL=1
        - INTERNAL_CPU_OFFLOAD_ENGINE=0
        - SRIOV_EN=1
        - NUM_OF_VFS=46
        - LAG_RESOURCE_ALLOCATION=1
        - NVME_EMULATION_ENABLE=1
```

```
        - NVME_EMULATION_NUM_PF=1
        - NVME_EMULATION_NUM_VF=125
        - NVME_EMULATION_NUM_MSIX=2
        - LINK_TYPE_P1=ETH
        - LINK_TYPE_P2=ETH
  ovs:
    rawConfigScript: |
      _ovs-vsctl() {
        ovs-vsctl --no-wait --timeout 15 "$@"
      }

      _ovs-vsctl set Open_vSwitch . other_config:doca-init=true
      _ovs-vsctl set Open_vSwitch . other_config:dpdk-max-
memzones=50000
      _ovs-vsctl set Open_vSwitch . other_config:hw-offload=true
      _ovs-vsctl set Open_vSwitch . other_config:pmd-quiet-
idle=true
      _ovs-vsctl set Open_vSwitch . other_config:max-idle=20000
      _ovs-vsctl set Open_vSwitch . other_config:max-
revalidator=5000
      _ovs-vsctl set Open_vSwitch . other_config:ctl-pipe-
size=1024
      _ovs-vsctl --if-exists del-br ovsbr1
      _ovs-vsctl --if-exists del-br ovsbr2
      _ovs-vsctl --may-exist add-br br-sfc
      _ovs-vsctl set bridge br-sfc datapath_type=netdev
      _ovs-vsctl set bridge br-sfc fail_mode=secure
      _ovs-vsctl --may-exist add-br br-hbn
      _ovs-vsctl set bridge br-hbn datapath_type=netdev
      _ovs-vsctl set bridge br-hbn fail_mode=secure
      _ovs-vsctl --may-exist add-port br-sfc p0
      _ovs-vsctl set Interface p0 type=dpdk
      _ovs-vsctl set Interface p0 mtu_request=9216
      _ovs-vsctl set Port p0 external_ids:dpf-type=physical
```

```
      _ovs-vsctl set Open_vSwitch . external-ids:ovn-bridge-
datapath-type=netdev
      _ovs-vsctl --may-exist add-br br-ovn
      _ovs-vsctl set bridge br-ovn datapath_type=netdev
      _ovs-vsctl br-set-external-id br-ovn bridge-id br-ovn
      _ovs-vsctl br-set-external-id br-ovn bridge-uplink
puplinkbrovntobrsfc
      _ovs-vsctl --may-exist add-port br-ovn pf0hpf
      _ovs-vsctl set Interface pf0hpf type=dpdk
      _ovs-vsctl set Interface pf0hpf mtu_request=9216

  bfcfgParameters:
    - UPDATE_ATF_UEFI=yes
    - UPDATE_DPU_OS=yes
    - WITH_NIC_FW_UPDATE=yes

  hostNetworkInterfaceConfigs:
    - portNumber: 0
      dhcp: true
      mtu: 1500

  configFiles:
  - path: /etc/mellanox/mlnx-bf.conf
    operation: override
    raw: |
        ALLOW_SHARED_RQ="no"
        IPSEC_FULL_OFFLOAD="no"
        ENABLE_ESWITCH_MULTIPORT="yes"
        RDMA_SET_NETNS_EXCLUSIVE="no"
    permissions: "0644"
  - path: /etc/mellanox/mlnx-ovs.conf
    operation: override
    raw: |
        CREATE_OVS_BRIDGES="no"
        OVS_DOCA="yes"
    permissions: "0644"
```

```
  - path: /etc/mellanox/mlnx-sf.conf
    operation: override
    raw: ""
    permissions: "0644"
```

DPUDeployment to provision DPUs on worker nodes with SNAP Block Storage

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUDeployment
metadata:
  name: hbn-ovnk-snap-nvme
  namespace: dpf-operator-system
spec:
  dpus:
    bfb: bf-bundle-$TAG
    flavor: hbn-ovnk-snap-nvme-$TAG
    dpuSets:
    - nameSuffix: "dpuset1"
      nodeSelector:
        matchLabels:
          feature.node.kubernetes.io/dpu-enabled: "true"
  services:
    ovn:
      serviceTemplate: ovn
      serviceConfiguration: ovn
    hbn:
      serviceTemplate: hbn
      serviceConfiguration: hbn
    dts:
      serviceTemplate: dts
      serviceConfiguration: dts
    blueman:
      serviceTemplate: blueman
      serviceConfiguration: blueman
    snap-csi-plugin:
      serviceTemplate: snap-csi-plugin
      serviceConfiguration: snap-csi-plugin
    snap-host-controller:
      serviceTemplate: snap-host-controller
      serviceConfiguration: snap-host-controller
    snap-node-driver:
```

```
      serviceTemplate: snap-node-driver
      serviceConfiguration: snap-node-driver
    doca-snap:
      serviceTemplate: doca-snap
      serviceConfiguration: doca-snap
    block-storage-dpu-plugin:
      serviceTemplate: block-storage-dpu-plugin
      serviceConfiguration: block-storage-dpu-plugin
    spdk-csi-controller:
      serviceTemplate: spdk-csi-controller
      serviceConfiguration: spdk-csi-controller
    spdk-csi-controller-dpu:
      serviceTemplate: spdk-csi-controller-dpu
      serviceConfiguration: spdk-csi-controller-dpu
  serviceChains:
    switches:
      - ports:
        - serviceInterface:
            matchLabels:
              uplink: p0
        - service:
            name: hbn
            interface: p0_if
      - ports:
        - serviceInterface:
            matchLabels:
              uplink: p1
        - service:
            name: hbn
            interface: p1_if
      - ports:
        - serviceInterface:
            matchLabels:
              port: ovn
        - service:
            name: hbn
```

```
            interface: pf2dpu2_if
      - ports:
        - service:
            name: doca-snap
            interface: app_sf
            ipam:
              matchLabels:
                svc.dpu.nvidia.com/pool: storage-pool
        - service:
            name: hbn
            interface: snap_if
```

OVN DPUServiceConfiguration and DPUServiceTemplate to deploy OVN workloads to the DPUs

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceConfiguration
metadata:
  name: ovn
  namespace: dpf-operator-system
spec:
  deploymentServiceName: "ovn"
  serviceConfiguration:
    helmChart:
      values:
        k8sAPIServer:
https://$TARGETCLUSTER_API_SERVER_HOST:$TARGETCLUSTER_API_SERVER_PORT
        podNetwork: $POD_CIDR/24
        serviceNetwork: $SERVICE_CIDR
        dpuManifests:
          kubernetesSecretName: "ovn-dpu" # user needs to populate
based on DPUServiceCredentialRequest
          vtepCIDR: "10.0.120.0/22" # user needs to populate based on
DPUServiceIPAM
          hostCIDR: $TARGETCLUSTER_NODE_CIDR # user needs to
populate
          ipamPool: "pool1" # user needs to populate based on
DPUServiceIPAM
          ipamPoolType: "cidrpool" # user needs to populate based on
DPUServiceIPAM
          ipamVTEPIPIndex: 0
          ipamPFIPIndex: 1
```

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceTemplate
metadata:
  name: ovn
  namespace: dpf-operator-system
spec:
  deploymentServiceName: "ovn"
  helmChart:
    source:
      repoURL: $OVN_KUBERNETES_REPO_URL
      chart: ovn-kubernetes-chart
      version: $TAG
    values:
      commonManifests:
        enabled: true
      dpuManifests:
        enabled: true
      leaseNamespace: "ovn-kubernetes"
      gatewayOpts: "--gateway-interface=br-ovn"
```

HBN DPUServiceConfiguration and DPUServiceTemplate to deploy HBN workloads to the DPUs

```yaml
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceConfiguration
metadata:
  name: hbn
  namespace: dpf-operator-system
spec:
  deploymentServiceName: "hbn"
  serviceConfiguration:
    serviceDaemonSet:
      annotations:
        k8s.v1.cni.cncf.io/networks: |-
          [
          {"name": "iprequest", "interface": "ip_lo", "cni-args": {"poolNames": ["loopback"], "poolType": "cidrpool"}},
          {"name": "iprequest", "interface": "ip_pf2dpu2", "cni-args": {"poolNames": ["pool1"], "poolType": "cidrpool", "allocateDefaultGateway": true}}
          ]
    helmChart:
      values:
        configuration:
          perDPUValuesYAML: |
            - hostnamePattern: "*"
              values:
                bgp_peer_group: hbn
            - hostnamePattern: "worker1*"
              values:
                bgp_autonomous_system: 65101
            - hostnamePattern: "worker2*"
              values:
                bgp_autonomous_system: 65201
          startupYAMLJ2: |
            - header:
                model: BLUEFIELD
                nvue-api-version: nvue_v1
```

```
            rev-id: 1.0
            version: HBN 2.4.0
        - set:
            evpn:
              enable: on
            nve:
             vxlan:
              enable: on
              source:
                address: {{ ipaddresses.ip_lo.ip }}
            bridge:
              domain:
                br_default:
                  vlan:
                    '10':
                      vni:
                        '10': {}
            interface:
              lo:
                ip:
                  address:
                    {{ ipaddresses.ip_lo.ip }}/32: {}
                type: loopback
              p0_if,p1_if,snap_if:
                type: swp
                link:
                  mtu: 9000
              pf2dpu2_if:
                ip:
                  address:
                    {{ ipaddresses.ip_pf2dpu2.cidr }}: {}
                type: swp
                link:
                  mtu: 9000
              snap_if:
                bridge:
```

```
                      domain:
                        br_default:
                          access: 10
                  vlan10:
                    type: svi
                    vlan: 10
                router:
                  bgp:
                    autonomous-system: {{
config.bgp_autonomous_system }}
                    enable: on
                    graceful-restart:
                     mode: full
                    router-id: {{ ipaddresses.ip_lo.ip }}
                vrf:
                  default:
                    router:
                      bgp:
                        address-family:
                          ipv4-unicast:
                            enable: on
                            redistribute:
                              connected:
                                enable: on
                          ipv6-unicast:
                            enable: on
                            redistribute:
                              connected:
                                enable: on
                          l2vpn-evpn:
                            enable: on
                        enable: on
                        neighbor:
                          p0_if:
                            peer-group: {{ config.bgp_peer_group
}}
```

```
                              type: unnumbered
                            p1_if:
                              peer-group: {{ config.bgp_peer_group
}}
                              type: unnumbered
                          path-selection:
                            multipath:
                              aspath-ignore: on
                          peer-group:
                            {{ config.bgp_peer_group }}:
                              address-family:
                                ipv4-unicast:
                                  enable: on
                                ipv6-unicast:
                                  enable: on
                                l2vpn-evpn:
                                  enable: on
                              remote-as: external
  interfaces:
    ## NOTE: Interfaces inside the HBN pod must have the `_if`
suffix due to a naming convention in HBN.
  - name: p0_if
    network: mybrhbn
  - name: p1_if
    network: mybrhbn
  - name: pf2dpu2_if
    network: mybrhbn
  - name: snap_if
    network: mybrhbn
```

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceTemplate
metadata:
  name: hbn
  namespace: dpf-operator-system
spec:
  deploymentServiceName: "hbn"
  helmChart:
    source:
      repoURL: $HELM_REGISTRY_REPO_URL
      version: 1.0.5
      chart: doca-hbn
    values:
      image:
        repository: $HBN_NGC_IMAGE_URL
        tag: 3.2.1-doca3.2.1
      resources:
        memory: 6Gi
        nvidia.com/bf_sf: 4
```

DOCA Telemetry Service DPUServiceConfiguration and DPUServiceTemplate to deploy DTS to the DPUs

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceConfiguration
metadata:
  name: dts
  namespace: dpf-operator-system
spec:
  deploymentServiceName: "dts"
```

```yaml
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceTemplate
metadata:
  name: dts
  namespace: dpf-operator-system
spec:
  deploymentServiceName: "dts"
  helmChart:
    source:
      repoURL: $HELM_REGISTRY_REPO_URL
      version: 1.23.4
      chart: doca-telemetry
```

Blueman DPUServiceConfiguration and DPUServiceTemplate to deploy Blueman to the DPUs

```yaml
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceConfiguration
metadata:
  name: blueman
  namespace: dpf-operator-system
spec:
  deploymentServiceName: "blueman"
```

```yaml
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceTemplate
metadata:
  name: blueman
  namespace: dpf-operator-system
spec:
  deploymentServiceName: "blueman"
  helmChart:
    source:
      repoURL: $HELM_REGISTRY_REPO_URL
      version: 1.0.8
      chart: doca-blueman
```

DOCA SNAP DPUServiceConfiguration and DPUServiceTemplate for NVMe emulation

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceConfiguration
metadata:
  name: doca-snap
  namespace: dpf-operator-system
spec:
  deploymentServiceName: doca-snap
  serviceConfiguration:
    helmChart:
      values:
        dpu:
          docaSnap:
            enabled: true
            image:
              repository: $SNAP_NGC_IMAGE_URL
              tag: 1.5.0-doca3.2.0
            snapRpcInitConf: |
              nvme_subsystem_create --nqn nqn.2022-
10.io.nvda.nvme:0
              nvme_controller_create --nqn nqn.2022-
10.io.nvda.nvme:0 --ctrl NVMeCtrl1 --pf_id 0 --admin_only
  interfaces:
    - name: app_sf
      network: mybrsfc
```

```yaml
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceTemplate
metadata:
  name: doca-snap
  namespace: dpf-operator-system
spec:
  deploymentServiceName: doca-snap
  helmChart:
    source:
      repoURL: $REGISTRY
      version: $TAG
      chart: dpf-storage
    values:
      serviceDaemonSet:
        resources:
          memory: "2Gi"
          hugepages-2Mi: "4Gi"
          cpu: "8"
          nvidia.com/bf_sf: 1
  resourceRequirements:
    memory: "2Gi"
    hugepages-2Mi: "4Gi"
    cpu: "8"
    nvidia.com/bf_sf: 1
```

SNAP CSI Plugin DPUServiceConfiguration and DPUServiceTemplate

```yaml
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceConfiguration
metadata:
  name: snap-csi-plugin
  namespace: dpf-operator-system
spec:
  deploymentServiceName: snap-csi-plugin
  upgradePolicy:
    applyNodeEffect: false
  serviceConfiguration:
    deployInCluster: true
    helmChart:
      values:
        host:
          snapCsiPlugin:
            enabled: true
            controller:
              affinity:
                nodeAffinity:
                  requiredDuringSchedulingIgnoredDuringExecution:
                    nodeSelectorTerms:
                      - matchExpressions:
                          - key: "node-role.kubernetes.io/master"
                            operator: Exists
                      - matchExpressions:
                          - key: "node-role.kubernetes.io/control-plane"
                            operator: Exists
            node:
              hostNetwork: true
```

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceTemplate
metadata:
  name: snap-csi-plugin
  namespace: dpf-operator-system
spec:
  deploymentServiceName: snap-csi-plugin
  helmChart:
    source:
      repoURL: $REGISTRY
      version: $TAG
      chart: dpf-storage
```

SNAP Host Controller DPUServiceConfiguration and DPUServiceTemplate

```yaml
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceConfiguration
metadata:
  name: snap-host-controller
  namespace: dpf-operator-system
spec:
  deploymentServiceName: snap-host-controller
  upgradePolicy:
    applyNodeEffect: false
  serviceConfiguration:
    deployInCluster: true
    helmChart:
      values:
        host:
          snapHostController:
            enabled: true
            config:
              targetNamespace: dpf-operator-system
            affinity:
              nodeAffinity:
                requiredDuringSchedulingIgnoredDuringExecution:
                  nodeSelectorTerms:
                  - matchExpressions:
                      - key: "node-role.kubernetes.io/master"
                        operator: Exists
                  - matchExpressions:
                      - key: "node-role.kubernetes.io/control-plane"
                        operator: Exists
```

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceTemplate
metadata:
  name: snap-host-controller
  namespace: dpf-operator-system
spec:
  deploymentServiceName: snap-host-controller
  helmChart:
    source:
      repoURL: $REGISTRY
      version: $TAG
      chart: dpf-storage
```

SNAP Node Driver DPUServiceConfiguration and DPUServiceTemplate

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceConfiguration
metadata:
  name: snap-node-driver
  namespace: dpf-operator-system
spec:
  deploymentServiceName: snap-node-driver
  serviceConfiguration:
    helmChart:
      values:
        dpu:
          deployCrds: true
          snapNodeDriver:
            enabled: true
```

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceTemplate
metadata:
  name: snap-node-driver
  namespace: dpf-operator-system
spec:
  deploymentServiceName: snap-node-driver
  helmChart:
    source:
      repoURL: $REGISTRY
      version: $TAG
      chart: dpf-storage
```

Block Storage DPU Plugin DPUServiceConfiguration and DPUServiceTemplate

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceConfiguration
metadata:
  name: block-storage-dpu-plugin
  namespace: dpf-operator-system
spec:
  deploymentServiceName: block-storage-dpu-plugin
  serviceConfiguration:
    helmChart:
      values:
        dpu:
          blockStorageVendorDpuPlugin:
            enabled: true
```

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceTemplate
metadata:
  name: block-storage-dpu-plugin
  namespace: dpf-operator-system
spec:
  deploymentServiceName: block-storage-dpu-plugin
  helmChart:
    source:
      repoURL: $REGISTRY
      version: $TAG
      chart: dpf-storage
```

SPDK CSI Controller DPUServiceConfiguration and DPUServiceTemplate (Host Cluster)

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceConfiguration
metadata:
  name: spdk-csi-controller
  namespace: dpf-operator-system
spec:
  deploymentServiceName: spdk-csi-controller
  upgradePolicy:
    applyNodeEffect: false
  serviceConfiguration:
    deployInCluster: true
    helmChart:
      values:
        host:
          enabled: true
          config:
            targets:
              nodes:
                # name of the target
                - name: spdk-target
                  # management address
                  rpcURL: http://10.0.110.25:8000
                  # type of the target, e.g. nvme-tcp, nvme-rdma
                  targetType: nvme-rdma
                  # target service IP
                  targetAddr: 10.0.124.1
            # required parameter, name of the secret that
contains connection
            # details to access the DPU cluster.
            # this secret should be created by the
DPUServiceCredentialRequest API.
```

```
            dpuClusterSecret: spdk-csi-controller-dpu-cluster-
credentials
```

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceTemplate
metadata:
  name: spdk-csi-controller
  namespace: dpf-operator-system
spec:
  deploymentServiceName: spdk-csi-controller
  helmChart:
    source:
      repoURL: oci://ghcr.io/mellanox/dpf-storage-vendors-charts
      version: v0.3.0
      chart: spdk-csi-controller
```

SPDK CSI Controller DPU DPUServiceConfiguration and DPUServiceTemplate (DPU Cluster)

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceConfiguration
metadata:
  name: spdk-csi-controller-dpu
  namespace: dpf-operator-system
spec:
  deploymentServiceName: spdk-csi-controller-dpu
  upgradePolicy:
    applyNodeEffect: false
  serviceConfiguration:
    helmChart:
      values:
        dpu:
          enabled: true
          storageClass:
            # the name of the storage class that will be created for
spdk-csi,
            # this StorageClass name should be used in the
StorageVendor settings
            name: spdkcsi-sc
            # name of the secret that contains credentials for the
remote SPDK target,
            # content of the secret is injected during
CreateVolume request
            secretName: spdkcsi-secret
            # namespace of the secret with credentials for the
remote SPDK target
            secretNamespace: dpf-operator-system
          rbacRoles:
            spdkCsiController:
              # the name of the service account for spdk-csi-
controller
              # this value must be aligned with the value from the
DPUServiceCredentialRequest
```

```
              serviceAccount: spdk-csi-controller-sa
```

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceTemplate
metadata:
  name: spdk-csi-controller-dpu
  namespace: dpf-operator-system
spec:
  deploymentServiceName: spdk-csi-controller-dpu
  helmChart:
    source:
      repoURL: oci://ghcr.io/mellanox/dpf-storage-vendors-charts
      version: v0.3.0
      chart: spdk-csi-controller
```

DPUServiceIPAM for SNAP storage networking

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceIPAM
metadata:
  name: storage-pool
  namespace: dpf-operator-system
spec:
  metadata:
    labels:
      svc.dpu.nvidia.com/pool: storage-pool
  ipv4Subnet:
    subnet: "10.0.124.0/24"
    gateway: "10.0.124.1"
    perNodeIPCount: 4
```

Secret for SPDK CSI credentials

```
---
apiVersion: v1
kind: Secret
metadata:
  name: spdkcsi-secret
  namespace: dpf-operator-system
  labels:
    # this label enables replication of the secret from the host to
the dpu cluster
    dpu.nvidia.com/image-pull-secret: ""
stringData:
  # name field in the "rpcTokens" list should match name of the
  # spdk target from
DPUService.helmChart.values.host.config.targets.nodes
  secret.json: |-
    {
      "rpcTokens": [
        {
          "name": "spdk-target",
          "username": "exampleuser",
          "password": "examplepassword"
        }
      ]
    }
```

SPDK CSI Controller DPUServiceCredentialRequest

```yaml
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceCredentialRequest
metadata:
  name: spdk-csi-controller-credentials
  namespace: dpf-operator-system
spec:
  duration: 10m
  serviceAccount:
    name: spdk-csi-controller-sa
    namespace: dpf-operator-system
  targetCluster:
    name: dpu-cplane-tenant1
    namespace: dpu-cplane-tenant1
  type: tokenFile
  secret:
    name: spdk-csi-controller-dpu-cluster-credentials
    namespace: dpf-operator-system
```

OVN DPUServiceCredentialRequest to allow cross cluster communication

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceCredentialRequest
metadata:
  name: ovn-dpu
  namespace: dpf-operator-system
spec:
  serviceAccount:
    name: ovn-dpu
    namespace: dpf-operator-system
  duration: 24h
  type: tokenFile
  secret:
    name: ovn-dpu
    namespace: dpf-operator-system
  metadata:
    labels:
      dpu.nvidia.com/image-pull-secret: ""
```

DPUServiceInterfaces for physical ports on the DPU

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceInterface
metadata:
  name: p0
  namespace: dpf-operator-system
spec:
  template:
    spec:
      template:
        metadata:
          labels:
            uplink: "p0"
        spec:
          interfaceType: physical
          physical:
            interfaceName: p0
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceInterface
metadata:
  name: p1
  namespace: dpf-operator-system
spec:
  template:
    spec:
      template:
        metadata:
          labels:
            uplink: "p1"
        spec:
          interfaceType: physical
          physical:
```

```
            interfaceName: p1
```

OVN DPUServiceInterface to define the ports attached to OVN workloads on the DPU

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceInterface
metadata:
  name: ovn
  namespace: dpf-operator-system
spec:
  template:
    spec:
      template:
        metadata:
          labels:
            port: ovn
        spec:
          interfaceType: ovn
```

DPUServiceIPAM to set up IP Address Management on the DPUCluster

```yaml
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceIPAM
metadata:
  name: pool1
  namespace: dpf-operator-system
spec:
  ipv4Network:
    network: "10.0.120.0/22"
    gatewayIndex: 3
    prefixSize: 29
```

DPUServiceIPAM for the loopback interface in HBN

```yaml
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceIPAM
metadata:
  name: loopback
  namespace: dpf-operator-system
spec:
  ipv4Network:
    network: "11.0.0.0/24"
    prefixSize: 32
```

Verification

These verification commands may need to be run multiple times to ensure the condition is met.

Note that the DPUService name will have a random suffix. For example, `hbn-vs6mj`. Use the correct name for the verification.

Verify the DPU and Service installation with:

```
## Ensure the DPUServices are created and have been reconciled.
kubectl wait --for=condition=ApplicationsReconciled --namespace
dpf-operator-system dpuservices -l svc.dpu.nvidia.com/owned-by-
dpudeployment=dpf-operator-system_hbn-ovnk-snap-nvme
## Ensure the DPUServiceIPAMs have been reconciled
kubectl wait --for=condition=DPUIPAMObjectReconciled --namespace
dpf-operator-system dpuserviceipam --all
## Ensure the DPUServiceInterfaces have been reconciled
kubectl wait --for=condition=ServiceInterfaceSetReconciled --
namespace dpf-operator-system dpuserviceinterface --all
## Ensure the DPUServiceChains have been reconciled
kubectl wait --for=condition=ServiceChainSetReconciled --namespace
dpf-operator-system dpuservicechain --all
```

6. Test Traffic

Add worker nodes to the cluster

At this point workers should be added to the cluster. Each worker node should be configured in line with the prerequisites. As workers are added to the cluster DPUs will be provisioned and DPUServices will begin to be spun up.

You can verify the status of the DPUDeployment and its components with the following command:

```
$ kubectl -n dpf-operator-system exec deploy/dpf-operator-
controller-manager -- /dpfctl describe dpudeployments
```

Deploy test pods

```
kubectl apply -f manifests/06-test-traffic
```

HBN and OVN functionality can be tested by pinging between the pods and services deployed in the default namespace.

7. Apply Storage Configuration

```
kubectl apply -f manifests/07-storage-configuration
```

This will create the following objects:

DPUStorageVendor for SPDK CSI

```
---
apiVersion: storage.dpu.nvidia.com/v1alpha1
kind: DPUStorageVendor
metadata:
  name: spdk-csi
  namespace: dpf-operator-system
spec:
  storageClassName: spdkcsi-sc
  pluginName: nvidia-block
```

DPUStoragePolicy for block storage

```
---
apiVersion: storage.dpu.nvidia.com/v1alpha1
kind: DPUStoragePolicy
metadata:
  name: policy-block
  namespace: dpf-operator-system
spec:
  dpuStorageVendors:
    - spdk-csi
  selectionAlgorithm: "NumberVolumes"
  parameters: {}
```

Wait for the objects to be ready:

```
kubectl wait --for=condition=Ready --namespace dpf-operator-system
dpustoragevendors --all
kubectl wait --for=condition=Ready --namespace dpf-operator-system
dpustoragepolicies --all
```

8. Test Storage Workload

This section walks you through deploying a workload that utilizes storage resources provided by the SNAP service.

```
kubectl apply -f manifests/08-test-storage
```

This will create the following objects:

StorageClass for SNAP NVMe VF

```
---
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: snap-nvme-vf
provisioner: csi.snap.nvidia.com
parameters:
  policy: "policy-block"
  functionType: "vf"
  hotplugFunction: "false"
```

StatefulSet with NVMe VF block storage

```yaml
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: storage-test-pod-nvme-vf
spec:
  serviceName: "storage-test-pod-nvme-vf"
  podManagementPolicy: "Parallel"
  replicas: 1
  selector:
    matchLabels:
      app: storage-test-pod-nvme-vf
  template:
    metadata:
      labels:
        app: storage-test-pod-nvme-vf
    spec:
      containers:
      - name: test
        image: registry.k8s.io/nginx-slim:0.21
        volumeDevices:
          - name: vol1
            devicePath: /dev/xvda
  volumeClaimTemplates:
  - metadata:
      name: vol1
    spec:
      accessModes: [ "ReadWriteOnce" ]
      volumeMode: Block
      storageClassName: snap-nvme-vf
      resources:
        requests:
          storage: 1Gi
```

Verification

Wait for the pod to be in running state:

```
kubectl wait --for=condition=ready pod -l app=storage-test-pod-nvme-vf --timeout=300s
```

Verify the block device is available in the pod:

```
kubectl exec -it storage-test-pod-nvme-vf-0 -- ls -l /dev/xvda
```

You can test read/write operations on the block device:

```
## Write test data to the block device
kubectl exec -it storage-test-pod-nvme-vf-0 -- sh -c "echo 'test data' | dd of=/dev/xvda bs=512 count=1"

## Read back the test data
kubectl exec -it storage-test-pod-nvme-vf-0 -- dd if=/dev/xvda bs=512 count=1 2>/dev/null
```

Uninstall

This section describes how to clean up the cluster after the DPF setup has been completed. It is important to follow the steps in the correct order to ensure that all components are removed cleanly and that the cluster remains functional.

Delete the storage test resources

```
kubectl delete -f manifests/08-test-storage --wait --ignore-not-
found=true
# delete all PVCs created by StatefulSet
kubectl delete pvc --selector=app=storage-test-pod-nvme-vf --wait
-n default
kubectl delete -n dpf-operator-system dpuvolumeattachment --all -
-wait
kubectl delete -n dpf-operator-system dpuvolume --all --wait
# delete storage configuration
kubectl delete -f manifests/07-storage-configuration --wait --
ignore-not-found=true
```

Delete the network test pods

```
kubectl delete -f manifests/06-test-traffic --wait --ignore-not-
found=true
```

Delete DPF CNI acceleration components

```
kubectl delete -f manifests/04-enable-accelerated-cni --wait --
ignore-not-found=true
helm uninstall -n nvidia-network-operator network-operator --wait

## Note: Uninstalling OVN Kubernetes as primary CNI is not
supported but this command must be run to remove the webhook and
restore a functioning cluster.
helm uninstall -n ovn-kubernetes ovn-kubernetes-resource-injector
--wait
```

Delete the DPF Operator system and DPF Operator

First we have to delete some DPUServiceInterfaces. This is necessary because of a known issue during uninstallation.

```
kubectl delete -n dpf-operator-system dpuserviceinterface p0 p1
ovn --wait
```

Then we can delete the config and system namespace.

```
kubectl delete -n dpf-operator-system dpfoperatorconfig
dpfoperatorconfig --wait
helm uninstall -n dpf-operator-system dpf-operator --wait
```

Delete DPF Operator PVC

```
kubectl -n dpf-operator-system delete pvc bfb-pvc
kubectl delete pv bfb-pv
```

Note: there can be a race condition with deleting the underlying Kamaji cluster which runs the DPU cluster control plane in this guide. If that happens it may be necessary to remove finalizers manually from `DPUCluster` and `Datastore` objects.

Limitations of DPF Setup

Host network pod services

The Kubelet process on the Kubernetes nodes use the OOB interface IP address to register in Kubernetes. This means that the nodes have the OOB IP addresses as node IP addresses. This means that pods using host networking have the OOB IP address of the hosts as pod IP address. However, that interface is not accelerated. This means that any component using the addresses of the pods using host networking will not benefit from hardware acceleration and high-speed ports.

For example, this means that when creating a Kubernetes NodePort service selecting pods using host networking, even if the user uses the high-speed IP of the host, the traffic will not be accelerated. In order to solve this, it is possible to create dedicated endpointSlices that contain the host high-speed port IP addresses instead of OOB port IP addresses. This way, the entire path to the pods will be accelerated and benefit from high performances, if the user uses the high speed IP address of the host with the nodePort port. This requires the workload running on the pod with host networking to also listen on the high-speed port IP address.

SNAP Storage with Host Based Networking

Note

Follow this guide from the source GitHub repo at github.com/NVIDIA/doca-platform and moving to the `docs/public/user-guides/host-trusted/use-cases/hbn-snap/README.md` for better formatting of the code.

In this configuration DOCA SNAP is installed as a DPUService and combined with NVIDIA Host Based Networking (HBN).

This guide includes examples for both SNAP Block (NVMe) and SNAP VirtioFS storage scenarios.

Prerequisites

This guide should be run by cloning the repo from github.com/NVIDIA/doca-platform and moving to the `docs/public/user-guides/host-trusted/use-cases/hbn-snap` directory.

The system is set up as described in the system prerequisites. The HBN with SNAP storage use case has the additional requirements:

SNAP Block (NVMe) Prerequisites

- A remote SPDK target should be set up to provide persistent storage for SNAP Block Storage
- The SPDK target should be reachable from the DPUs
- The management interface of the SPDK target should be reachable from the control plane nodes
- Make sure to check Host OS Configuration Section in SNAP service documentation to validate the host OS configuration on the worker nodes

SNAP VirtioFS Prerequisites

- An external NFS server is required to provide persistent storage for SNAP VirtioFS
- The NFS server must be reachable by both the SNAP DPU service and the nvidia-fs DPU plugin
- The NFS service must also be accessible from the DPF control plane nodes to ensure proper operation
- Make sure to check Host OS Configuration Section in SNAP VirtioFS service documentation to validate the host OS configuration on the worker nodes

Software Prerequisites

This guide uses the following tools which must be installed on the machine where the commands contained in this guide run.

- kubectl

- helm
- envsubst

Kubernetes Prerequisites

- control plane setup is complete before starting this guide
- CNI installed before starting this guide
- worker nodes are not added until indicated by this guide
- High-speed ports are used for secondary workload network and not for primary CNI

Virtual Functions

A number of virtual functions (VFs) will be created on hosts when provisioning DPUs. Certain of these VFs are marked for specific usage:

- The first VF (vf0) is used by provisioning components.
- The remaining VFs are allocated by SR-IOV Device Plugin.

Installation Guide

0. Required Variables

The following variables are required by this guide. A sensible default is provided where it makes sense, but many will be specific to the target infrastructure.

Commands in this guide are run in the same directory that contains this readme.

Environment variables file

```
## Virtual IP used by the load balancer for the DPU Cluster. Must
be a reserved IP from the management subnet and not allocated by
DHCP.
export DPUCLUSTER_VIP=

## Interface on which the DPUCluster load balancer will listen.
Should be the management interface of the control plane node.
export DPUCLUSTER_INTERFACE=

## IP address of the NFS server used for storing the BFB image.
## NOTE: This environment variable does NOT control the address
of the NFS server used as a remote target by SNAP VirtioFS.
export NFS_SERVER_IP=

## The repository URL for the NVIDIA Helm chart registry.
## Usually this is the NVIDIA Helm NGC registry. For development
purposes, this can be set to a different repository.
export HELM_REGISTRY_REPO_URL=https://helm.ngc.nvidia.com/nvidia/doca

## The repository URL for the HBN container image.
## Usually this is the NVIDIA NGC registry. For development
purposes, this can be set to a different repository.
export HBN_NGC_IMAGE_URL=nvcr.io/nvidia/doca/doca_hbn

## The repository URL for the SNAP VFS container image.
## Usually this is the NVIDIA NGC registry. For development
purposes, this can be set to a different repository.
export SNAP_NGC_IMAGE_URL=nvcr.io/nvidia/doca/doca_vfs

## The DPF REGISTRY is the Helm repository URL where the DPF
Operator Chart resides.
## Usually this is the NVIDIA Helm NGC registry. For development
purposes, this can be set to a different repository.
export REGISTRY=https://helm.ngc.nvidia.com/nvidia/doca
```

```
## The DPF TAG is the version of the DPF components which will be deployed in this guide.
export TAG=v25.10.0

## URL to the BFB used in the `bfb.yaml` and linked by the DPUSet.
export BFB_URL="https://content.mellanox.com/BlueField/BFBs/Ubuntu24.04/bf-bundle-3.2.1-34_25.11_ubuntu-24.04_64k_prod.bfb"
```

Modify the variables in `manifests/00-env-vars/envvars.env` to fit your environment, then source the file:

```
source manifests/00-env-vars/envvars.env
```

1. DPF Operator Installation

Create storage required by the DPF Operator

A number of environment variables must be set before running this command.

```
kubectl create ns dpf-operator-system
cat manifests/01-dpf-operator-installation/*.yaml | envsubst |
kubectl apply -f -
```

This deploys the following objects:

PersistentVolume and PersistentVolumeClaim for the provisioning controller

```
---
apiVersion: v1
kind: PersistentVolume
metadata:
  name: bfb-pv
spec:
  capacity:
    storage: 10Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteMany
  nfs:
    path: /mnt/dpf_share/bfb
    server: $NFS_SERVER_IP
  persistentVolumeReclaimPolicy: Delete
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: bfb-pvc
  namespace: dpf-operator-system
spec:
  accessModes:
  - ReadWriteMany
  resources:
    requests:
      storage: 10Gi
  volumeMode: Filesystem
  storageClassName: ""
```

Local Path Provisioner Helm values

```
tolerations:
  - operator: Exists
    effect: NoSchedule
    key: node-role.kubernetes.io/control-plane
  - operator: Exists
    effect: NoSchedule
    key: node-role.kubernetes.io/master
```

Additional Dependencies

Before deploying the DPF Operator, ensure that Helm is properly configured according to the Helm prerequisites.

> ⚠ **Warning**
>
> This is a critical prerequisite step that must be completed for the DPF Operator to function properly.

Deploy the DPF Operator

A number of environment variables must be set before running this command.

HTTP Registry (default)

If the $REGISTRY is an HTTP Registry (default value) use this command:

```
helm repo add --force-update dpf-repository ${REGISTRY}
helm repo update
helm upgrade --install -n dpf-operator-system dpf-operator dpf-
repository/dpf-operator --version=$TAG
```

OCI Registry

For development purposes, if the $REGISTRY is an OCI Registry use this command:

```
helm upgrade --install -n dpf-operator-system dpf-operator
$REGISTRY/dpf-operator --version=$TAG
```

Verification

These verification commands may need to be run multiple times to ensure the condition is met.

Verify the DPF Operator installation with:

```
## Ensure the DPF Operator deployment is available.
kubectl rollout status deployment --namespace dpf-operator-system
dpf-operator-controller-manager
## Ensure all pods in the DPF Operator system are ready.
kubectl wait --for=condition=ready --namespace dpf-operator-system
pods --all
```

2. DPF System Installation

This section involves creating the DPF system components and some basic infrastructure required for a functioning DPF-enabled cluster.

Deploy the DPF System components

A number of environment variables must be set before running this command.

```
kubectl create ns dpu-cplane-tenant1
cat manifests/02-dpf-system-installation/*.yaml | envsubst |
kubectl apply -f -
```

This will create the following objects:

DPFOperatorConfig to install the DPF System components

```
---
apiVersion: operator.dpu.nvidia.com/v1alpha1
kind: DPFOperatorConfig
metadata:
  name: dpfoperatorconfig
  namespace: dpf-operator-system
spec:
  provisioningController:
    bfbPVCName: "bfb-pvc"
    dmsTimeout: 900
  kamajiClusterManager:
    disable: false
```

DPUCluster to serve as Kubernetes control plane for DPU nodes

```
---
apiVersion: provisioning.dpu.nvidia.com/v1alpha1
kind: DPUCluster
metadata:
  name: dpu-cplane-tenant1
  namespace: dpu-cplane-tenant1
spec:
  type: kamaji
  maxNodes: 10
  clusterEndpoint:
    # deploy keepalived instances on the nodes that match the
given nodeSelector.
    keepalived:
      # interface on which keepalived will listen. Should be the oob
interface of the control plane node.
      interface: $DPUCLUSTER_INTERFACE
      # Virtual IP reserved for the DPU Cluster load balancer.
Must not be allocatable by DHCP.
      vip: $DPUCLUSTER_VIP
      # virtualRouterID must be in range [1,255], make sure the
given virtualRouterID does not duplicate with any existing
keepalived process running on the host
      virtualRouterID: 126
      nodeSelector:
        node-role.kubernetes.io/control-plane: ""
```

Verification

These verification commands may need to be run multiple times to ensure the condition is met.

Verify the DPF System with:

```
## Ensure the provisioning and DPUService controller manager
deployments are available.
kubectl rollout status deployment --namespace dpf-operator-system
dpf-provisioning-controller-manager dpuservice-controller-manager
## Ensure all other deployments in the DPF Operator system are
Available.
kubectl rollout status deployment --namespace dpf-operator-system
## Ensure the DPUCluster is ready for nodes to join.
kubectl wait --for=condition=ready --namespace dpu-cplane-tenant1
dpucluster --all
```

3. Install Prerequisites for Accelerated Network

Install Multus using NVIDIA Network Operator

```
helm repo add nvidia https://helm.ngc.nvidia.com/nvidia --force-update
helm upgrade --no-hooks --install --create-namespace --namespace
nvidia-network-operator network-operator nvidia/network-operator
--version 25.7.0 -f ./manifests/03-enable-accelerated-
interfaces/helm-values/network-operator.yml
```

NVIDIA Network Operator Helm values

```yaml
nfd:
  enabled: false
  deployNodeFeatureRules: false
sriovNetworkOperator:
  enabled: true
sriov-network-operator:
  operator:
    affinity:
      nodeAffinity:
        requiredDuringSchedulingIgnoredDuringExecution:
          nodeSelectorTerms:
            - matchExpressions:
                - key: node-role.kubernetes.io/master
                  operator: Exists
            - matchExpressions:
                - key: node-role.kubernetes.io/control-plane
                  operator: Exists
  crds:
    enabled: true
  sriovOperatorConfig:
    deploy: true
    configDaemonNodeSelector: null
operator:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
          - matchExpressions:
              - key: node-role.kubernetes.io/master
                operator: Exists
          - matchExpressions:
              - key: node-role.kubernetes.io/control-plane
                operator: Exists
```

Apply the NicClusterPolicy

```
cat manifests/03-enable-accelerated-interfaces/*.yaml | envsubst
| kubectl apply -f -
```

This will deploy the following objects:

NICClusterPolicy for the NVIDIA Network Operator

```
---
apiVersion: mellanox.com/v1alpha1
kind: NicClusterPolicy
metadata:
  name: nic-cluster-policy
spec:
  secondaryNetwork:
    multus:
      image: multus-cni
      imagePullSecrets: []
      repository: ghcr.io/k8snetworkplumbingwg
      version: v3.9.3
```

Verification

These verification commands may need to be run multiple times to ensure the condition is met.

Verify the DPF System with:

```
## Ensure the provisioning and DPUService controller manager
deployments are available.
kubectl wait --for=condition=Ready --namespace nvidia-network-
operator pods --all
## Expect the following Daemonsets to be successfully rolled out.
kubectl rollout status daemonset --namespace nvidia-network-
operator kube-multus-ds
```

4. DPU Provisioning and Service Installation

In this section, you'll provision your DPUs and deploy the required services. You'll need to create a `DPUDeployment` object that defines which `DPUServices` should be installed on each selected DPU. This provides a flexible way to specify and manage the services that run on your DPUs.

> If you want to learn more about `DPUDeployments`, check the DPUDeployment documentation.

This guide includes examples for both SNAP Block (NVMe) and SNAP VirtioFS Storage. Please refer to the relevant sections below and follow the instructions to deploy the desired storage type.

SNAP Block (NVMe)

Create the DPUDeployment, DPUServiceConfig, DPUServiceTemplate and other necessary objects

A number of environment variables must be set before running this command.

```
cat manifests/04.1-dpudeployment-installation-nvme/*.yaml |
envsubst | kubectl apply -f -
```

This will deploy the following objects:

BFB to download Bluefield Bitstream to a shared volume

```
---
apiVersion: provisioning.dpu.nvidia.com/v1alpha1
kind: BFB
metadata:
  name: bf-bundle-$TAG
  namespace: dpf-operator-system
spec:
  url: $BFB_URL
```

HBN + SNAP NVMe DPUFlavor to configure DPUs on provisioning

```
---
apiVersion: provisioning.dpu.nvidia.com/v1alpha1
kind: DPUFlavor
metadata:
  name: hbn-snap-nvme-$TAG
  namespace: dpf-operator-system
spec:
  bfcfgParameters:
  - UPDATE_ATF_UEFI=yes
  - UPDATE_DPU_OS=yes
  - WITH_NIC_FW_UPDATE=yes
  configFiles:
  - operation: override
    path: /etc/mellanox/mlnx-bf.conf
    permissions: "0644"
    raw: |
      ALLOW_SHARED_RQ="no"
      IPSEC_FULL_OFFLOAD="no"
      ENABLE_ESWITCH_MULTIPORT="yes"
      RDMA_SET_NETNS_EXCLUSIVE="no"
  - operation: override
    path: /etc/mellanox/mlnx-ovs.conf
    permissions: "0644"
    raw: |
      CREATE_OVS_BRIDGES="no"
      OVS_DOCA="yes"
  - operation: override
    path: /etc/mellanox/mlnx-sf.conf
    permissions: "0644"
    raw: ""
  grub:
    kernelParameters:
    - console=hvc0
    - console=ttyAMA0
    - earlycon=pl011,0x13010000
```

```
    - fixrttc
    - net.ifnames=0
    - biosdevname=0
    - iommu.passthrough=1
    - cgroup_no_v1=net_prio,net_cls
    - hugepagesz=2048kB
    - hugepages=5120
  nvconfig:
  - device: '*'
    parameters:
    - PF_BAR2_ENABLE=0
    - PER_PF_NUM_SF=1
    - PF_TOTAL_SF=20
    - PF_SF_BAR_SIZE=10
    - NUM_PF_MSIX_VALID=0
    - PF_NUM_PF_MSIX_VALID=1
    - PF_NUM_PF_MSIX=228
    - INTERNAL_CPU_MODEL=1
    - INTERNAL_CPU_OFFLOAD_ENGINE=0
    - SRIOV_EN=1
    - NUM_OF_VFS=46
    - LAG_RESOURCE_ALLOCATION=1
    - PCI_SWITCH_EMULATION_ENABLE=1
    - PCI_SWITCH_EMULATION_NUM_PORT=32
    - NVME_EMULATION_ENABLE=1
    - NVME_EMULATION_NUM_PF=0
    - LINK_TYPE_P1=ETH
    - LINK_TYPE_P2=ETH
  ovs:
    rawConfigScript: |
      _ovs-vsctl() {
        ovs-vsctl --no-wait --timeout 15 "$@"
      }

      _ovs-vsctl set Open_vSwitch . other_config:doca-init=true
```

```
      _ovs-vsctl set Open_vSwitch . other_config:dpdk-max-
memzones=50000
      _ovs-vsctl set Open_vSwitch . other_config:hw-offload=true
      _ovs-vsctl set Open_vSwitch . other_config:pmd-quiet-
idle=true
      _ovs-vsctl set Open_vSwitch . other_config:max-idle=20000
      _ovs-vsctl set Open_vSwitch . other_config:max-
revalidator=5000
      _ovs-vsctl --if-exists del-br ovsbr1
      _ovs-vsctl --if-exists del-br ovsbr2
      _ovs-vsctl --may-exist add-br br-sfc
      _ovs-vsctl set bridge br-sfc datapath_type=netdev
      _ovs-vsctl set bridge br-sfc fail_mode=secure
      _ovs-vsctl --may-exist add-port br-sfc p0
      _ovs-vsctl set Interface p0 type=dpdk
      _ovs-vsctl set Interface p0 mtu_request=9216
      _ovs-vsctl set Port p0 external_ids:dpf-type=physical
      _ovs-vsctl --may-exist add-port br-sfc p1
      _ovs-vsctl set Interface p1 type=dpdk
      _ovs-vsctl set Interface p1 mtu_request=9216
      _ovs-vsctl set Port p1 external_ids:dpf-type=physical
      _ovs-vsctl --may-exist add-br br-hbn
      _ovs-vsctl set bridge br-hbn datapath_type=netdev
      _ovs-vsctl set bridge br-hbn fail_mode=secure
```

DPUServiceConfiguration and DPUServiceTemplate for DOCA HBN

```yaml
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceConfiguration
metadata:
  name: doca-hbn
  namespace: dpf-operator-system
spec:
  deploymentServiceName: "doca-hbn"
  serviceConfiguration:
    serviceDaemonSet:
      annotations:
        k8s.v1.cni.cncf.io/networks: |-
          [
          {"name": "iprequest", "interface": "ip_lo", "cni-args": {"poolNames": ["loopback"], "poolType": "cidrpool"}},
          {"name": "iprequest", "interface": "ip_pf0vf10", "cni-args": {"poolNames": ["pool1"], "poolType": "cidrpool", "allocateDefaultGateway": true}},
          {"name": "iprequest", "interface": "ip_pf1vf10", "cni-args": {"poolNames": ["pool2"], "poolType": "cidrpool", "allocateDefaultGateway": true}}
          ]
    helmChart:
      values:
        configuration:
          perDPUValuesYAML: |
            - hostnamePattern: "*"
              values:
                bgp_peer_group: hbn
                vrf1: RED
                vrf2: BLUE
                l3vni1: 100001
                l3vni2: 100002
            - hostnamePattern: "worker1*"
              values:
                bgp_autonomous_system: 65101
            - hostnamePattern: "worker2*"
```

```
            values:
              bgp_autonomous_system: 65201
        startupYAMLJ2: |
          - header:
              model: BLUEFIELD
              nvue-api-version: nvue_v1
              rev-id: 1.0
              version: HBN 3.0.0
          - set:
              evpn:
                enable: on
                route-advertise: {}
              bridge:
                domain:
                  br_default:
                    vlan:
                      '10':
                        vni:
                          '10': {}
              interface:
                lo:
                  ip:
                    address:
                      {{ ipaddresses.ip_lo.ip }}/32: {}
                  type: loopback
                p0_if,p1_if,pf0vf10_if,pf1vf10_if,snap_if:
                  type: swp
                  link:
                    mtu: 9000
                pf0vf10_if:
                  ip:
                    address:
                      {{ ipaddresses.ip_pf0vf10.cidr }}: {}
                    vrf: {{ config.vrf1 }}
                pf1vf10_if:
                  ip:
```

```
            address:
              {{ ipaddresses.ip_pf1vf10.cidr }}: {}
            vrf: {{ config.vrf2 }}
          snap_if:
            bridge:
              domain:
                br_default:
                  access: 10
          vlan10:
            type: svi
            vlan: 10
        nve:
          vxlan:
            arp-nd-suppress: on
            enable: on
            source:
              address: {{ ipaddresses.ip_lo.ip }}
        router:
          bgp:
            enable: on
            graceful-restart:
              mode: full
        vrf:
          default:
            router:
              bgp:
                address-family:
                  ipv4-unicast:
                    enable: on
                    redistribute:
                      connected:
                        enable: on
                    multipaths:
                      ebgp: 16
                  l2vpn-evpn:
                    enable: on
```

```
                        autonomous-system: {{
config.bgp_autonomous_system }}
                        enable: on
                        neighbor:
                          p0_if:
                            peer-group: {{ config.bgp_peer_group
}}
                            type: unnumbered
                            address-family:
                              l2vpn-evpn:
                                enable: on
                                add-path-tx: off
                          p1_if:
                            peer-group: {{ config.bgp_peer_group
}}
                            type: unnumbered
                            address-family:
                              l2vpn-evpn:
                                enable: on
                                add-path-tx: off
                        path-selection:
                          multipath:
                            aspath-ignore: on
                        peer-group:
                          {{ config.bgp_peer_group }}:
                            address-family:
                              ipv4-unicast:
                                enable: on
                              l2vpn-evpn:
                                enable: on
                            remote-as: external
                        router-id: {{ ipaddresses.ip_lo.ip }}
                    {{ config.vrf1 }}:
                      evpn:
                        enable: on
                        vni:
```

```
                        {{ config.l3vni1 }}: {}
                    router:
                      bgp:
                        address-family:
                          ipv4-unicast:
                            enable: on
                            redistribute:
                              connected:
                                enable: on
                            route-export:
                              to-evpn:
                                enable: on
                        autonomous-system: {{
config.bgp_autonomous_system }}
                        enable: on
                  {{ config.vrf2 }}:
                    evpn:
                      enable: on
                      vni:
                        {{ config.l3vni2 }}: {}
                    router:
                      bgp:
                        address-family:
                          ipv4-unicast:
                            enable: on
                            redistribute:
                              connected:
                                enable: on
                            route-export:
                              to-evpn:
                                enable: on
                        autonomous-system: {{
config.bgp_autonomous_system }}
                        enable: on

  interfaces:
```

```
  - name: p0_if
    network: mybrhbn
  - name: p1_if
    network: mybrhbn
  - name: pf0vf10_if
    network: mybrhbn
  - name: pf1vf10_if
    network: mybrhbn
  - name: snap_if
    network: mybrhbn
```

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceTemplate
metadata:
  name: doca-hbn
  namespace: dpf-operator-system
spec:
  deploymentServiceName: "doca-hbn"
  helmChart:
    source:
      repoURL: $HELM_REGISTRY_REPO_URL
      version: 1.0.5
      chart: doca-hbn
    values:
      image:
        repository: $HBN_NGC_IMAGE_URL
        tag: 3.2.1-doca3.2.1
      resources:
        memory: 6Gi
        nvidia.com/bf_sf: 5
```

DPUServiceInterfaces for physical ports and VFs on the DPU

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceInterface
metadata:
  name: p0
  namespace: dpf-operator-system
spec:
  template:
    spec:
      template:
        metadata:
          labels:
            uplink: "p0"
        spec:
          interfaceType: physical
          physical:
            interfaceName: p0
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceInterface
metadata:
  name: p1
  namespace: dpf-operator-system
spec:
  template:
    spec:
      template:
        metadata:
          labels:
            uplink: "p1"
        spec:
          interfaceType: physical
          physical:
            interfaceName: p1
---
```

```yaml
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceInterface
metadata:
  name: pf0vf10-rep
  namespace: dpf-operator-system
spec:
  template:
    spec:
      template:
        metadata:
          labels:
            vf: "pf0vf10"
        spec:
          interfaceType: vf
          vf:
            parentInterfaceRef: p0
            pfID: 0
            vfID: 10
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceInterface
metadata:
  name: pf1vf10-rep
  namespace: dpf-operator-system
spec:
  template:
    spec:
      template:
        metadata:
          labels:
            vf: "pf1vf10"
        spec:
          interfaceType: vf
          vf:
            parentInterfaceRef: p1
            pfID: 1
```

```
            vfID: 10
```

DPUServiceIPAMs

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceIPAM
metadata:
  name: pool1
  namespace: dpf-operator-system
spec:
  ipv4Network:
    network: "10.0.121.0/24"
    gatewayIndex: 2
    prefixSize: 29
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceIPAM
metadata:
  name: pool2
  namespace: dpf-operator-system
spec:
  ipv4Network:
    network: "10.0.122.0/24"
    gatewayIndex: 2
    prefixSize: 29
```

```yaml
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceIPAM
metadata:
  name: loopback
  namespace: dpf-operator-system
spec:
  ipv4Network:
    network: "11.0.0.0/24"
    prefixSize: 32
```

```yaml
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceIPAM
metadata:
  name: storage-pool
  namespace: dpf-operator-system
spec:
  metadata:
    labels:
      svc.dpu.nvidia.com/pool: storage-pool
  ipv4Subnet:
    subnet: "10.0.124.0/24"
    gateway: "10.0.124.1"
    perNodeIPCount: 8
```

DPUServiceConfiguration and DPUServiceTemplate for DOCA SNAP (NVMe)

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceConfiguration
metadata:
  name: doca-snap
  namespace: dpf-operator-system
spec:
  deploymentServiceName: doca-snap
  serviceConfiguration:
    helmChart:
      values:
        dpu:
          docaSnap:
            enabled: true
            image:
              repository: $SNAP_NGC_IMAGE_URL
              tag: 1.5.0-doca3.2.0
            snapRpcInitConf: |
              nvme_subsystem_create --nqn nqn.2022-
10.io.nvda.nvme:0
  interfaces:
  - name: app_sf
    network: mybrsfc
```

```yaml
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceTemplate
metadata:
  name: doca-snap
  namespace: dpf-operator-system
spec:
  deploymentServiceName: doca-snap
  helmChart:
    source:
      repoURL: $REGISTRY
      version: $TAG
      chart: dpf-storage
    values:
      serviceDaemonSet:
        resources:
          memory: "2Gi"
          hugepages-2Mi: "4Gi"
          cpu: "8"
          nvidia.com/bf_sf: 1
  resourceRequirements:
    memory: "2Gi"
    hugepages-2Mi: "4Gi"
    cpu: "8"
    nvidia.com/bf_sf: 1
```

DPUServiceConfiguration and DPUServiceTemplate for SNAP Host Controller

```yaml
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceConfiguration
metadata:
  name: snap-host-controller
  namespace: dpf-operator-system
spec:
  deploymentServiceName: snap-host-controller
  upgradePolicy:
    applyNodeEffect: false
  serviceConfiguration:
    deployInCluster: true
    helmChart:
      values:
        host:
          snapHostController:
            enabled: true
            config:
              targetNamespace: dpf-operator-system
            affinity:
              nodeAffinity:
                requiredDuringSchedulingIgnoredDuringExecution:
                  nodeSelectorTerms:
                  - matchExpressions:
                      - key: "node-role.kubernetes.io/master"
                        operator: Exists
                  - matchExpressions:
                      - key: "node-role.kubernetes.io/control-plane"
                        operator: Exists
```

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceTemplate
metadata:
  name: snap-host-controller
  namespace: dpf-operator-system
spec:
  deploymentServiceName: snap-host-controller
  helmChart:
    source:
      repoURL: $REGISTRY
      version: $TAG
      chart: dpf-storage
```

DPUServiceConfiguration and DPUServiceTemplate for SNAP Node Driver

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceConfiguration
metadata:
  name: snap-node-driver
  namespace: dpf-operator-system
spec:
  deploymentServiceName: snap-node-driver
  serviceConfiguration:
    helmChart:
      values:
        dpu:
          deployCrds: true
          snapNodeDriver:
            enabled: true
```

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceTemplate
metadata:
  name: snap-node-driver
  namespace: dpf-operator-system
spec:
  deploymentServiceName: snap-node-driver
  helmChart:
    source:
      repoURL: $REGISTRY
      version: $TAG
      chart: dpf-storage
```

DPUServiceConfiguration and DPUServiceTemplate for SNAP CSI Plugin (NVMe)

```yaml
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceConfiguration
metadata:
  name: snap-csi-plugin
  namespace: dpf-operator-system
spec:
  deploymentServiceName: snap-csi-plugin
  upgradePolicy:
    applyNodeEffect: false
  serviceConfiguration:
    deployInCluster: true
    helmChart:
      values:
        host:
          snapCsiPlugin:
            enabled: true
            emulationMode: "nvme"
            controller:
              affinity:
                nodeAffinity:
                  requiredDuringSchedulingIgnoredDuringExecution:
                    nodeSelectorTerms:
                      - matchExpressions:
                          - key: "node-role.kubernetes.io/master"
                            operator: Exists
                      - matchExpressions:
                          - key: "node-role.kubernetes.io/control-plane"
                            operator: Exists
```

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceTemplate
metadata:
  name: snap-csi-plugin
  namespace: dpf-operator-system
spec:
  deploymentServiceName: snap-csi-plugin
  helmChart:
    source:
      repoURL: $REGISTRY
      version: $TAG
      chart: dpf-storage
```

DPUServiceConfiguration and DPUServiceTemplate for Block Storage DPU Plugin

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceConfiguration
metadata:
  name: block-storage-dpu-plugin
  namespace: dpf-operator-system
spec:
  deploymentServiceName: block-storage-dpu-plugin
  serviceConfiguration:
    helmChart:
      values:
        dpu:
          blockStorageVendorDpuPlugin:
            enabled: true
```

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceTemplate
metadata:
  name: block-storage-dpu-plugin
  namespace: dpf-operator-system
spec:
  deploymentServiceName: block-storage-dpu-plugin
  helmChart:
    source:
      repoURL: $REGISTRY
      version: $TAG
      chart: dpf-storage
```

DPUServiceConfiguration, DPUServiceTemplate and DPUServiceCredentialRequest for SPDK CSI Controller

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceConfiguration
metadata:
  name: spdk-csi-controller
  namespace: dpf-operator-system
spec:
  deploymentServiceName: spdk-csi-controller
  upgradePolicy:
    applyNodeEffect: false
  serviceConfiguration:
    deployInCluster: true
    helmChart:
      values:
        host:
          enabled: true
          config:
            targets:
              nodes:
                # name of the target
                - name: spdk-target
                  # management address
                  rpcURL: http://10.0.110.25:8000
                  # type of the target, e.g. nvme-tcp, nvme-rdma
                  targetType: nvme-rdma
                  # target service IP
                  targetAddr: 10.0.124.1
            # required parameter, name of the secret that
contains connection
            # details to access the DPU cluster.
            # this secret should be created by the
DPUServiceCredentialRequest API.
```

```
            dpuClusterSecret: spdk-csi-controller-dpu-cluster-
credentials
```

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceTemplate
metadata:
  name: spdk-csi-controller
  namespace: dpf-operator-system
spec:
  deploymentServiceName: spdk-csi-controller
  helmChart:
    source:
      repoURL: oci://ghcr.io/mellanox/dpf-storage-vendors-charts
      version: v0.3.0
      chart: spdk-csi-controller
```

```yaml
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceCredentialRequest
metadata:
  name: spdk-csi-controller-credentials
  namespace: dpf-operator-system
spec:
  duration: 10m
  serviceAccount:
    name: spdk-csi-controller-sa
    namespace: dpf-operator-system
  targetCluster:
    name: dpu-cplane-tenant1
    namespace: dpu-cplane-tenant1
  type: tokenFile
  secret:
    name: spdk-csi-controller-dpu-cluster-credentials
    namespace: dpf-operator-system
```

Secret for SPDK CSI Controller

```
---
apiVersion: v1
kind: Secret
metadata:
  name: spdkcsi-secret
  namespace: dpf-operator-system
  labels:
    # this label enables replication of the secret from the host to
the dpu cluster
    dpu.nvidia.com/image-pull-secret: ""
stringData:
  # name field in the "rpcTokens" list should match name of the
  # spdk target from
DPUService.helmChart.values.host.config.targets.nodes
  secret.json: |-
    {
      "rpcTokens": [
        {
          "name": "spdk-target",
          "username": "exampleuser",
          "password": "examplepassword"
        }
      ]
    }
```

DPUServiceConfiguration and DPUServiceTemplate for SPDK CSI Controller on DPU

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceConfiguration
metadata:
  name: spdk-csi-controller-dpu
  namespace: dpf-operator-system
spec:
  deploymentServiceName: spdk-csi-controller-dpu
  upgradePolicy:
    applyNodeEffect: false
  serviceConfiguration:
    helmChart:
      values:
        dpu:
          enabled: true
          storageClass:
            # the name of the storage class that will be created for
spdk-csi,
            # this StorageClass name should be used in the
StorageVendor settings
            name: spdkcsi-sc
            # name of the secret that contains credentials for the
remote SPDK target,
            # content of the secret is injected during
CreateVolume request
            secretName: spdkcsi-secret
            # namespace of the secret with credentials for the
remote SPDK target
            secretNamespace: dpf-operator-system
          rbacRoles:
            spdkCsiController:
              # the name of the service account for spdk-csi-
controller
              # this value must be aligned with the value from the
DPUServiceCredentialRequest
```

```
                serviceAccount: spdk-csi-controller-sa
```

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceTemplate
metadata:
  name: spdk-csi-controller-dpu
  namespace: dpf-operator-system
spec:
  deploymentServiceName: spdk-csi-controller-dpu
  helmChart:
    source:
      repoURL: oci://ghcr.io/mellanox/dpf-storage-vendors-charts
      version: v0.3.0
      chart: spdk-csi-controller
```

DPUDeployment to provision DPUs on worker nodes

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUDeployment
metadata:
  name: hbn-snap
  namespace: dpf-operator-system
spec:
  dpus:
    bfb: bf-bundle-$TAG
    flavor: hbn-snap-nvme-$TAG
    dpuSets:
    - nameSuffix: "dpuset1"
      dpuAnnotations:
        storage.nvidia.com/preferred-dpu: "true"
      nodeSelector:
        matchLabels:
          feature.node.kubernetes.io/dpu-enabled: "true"
  services:
    doca-hbn:
      serviceTemplate: doca-hbn
      serviceConfiguration: doca-hbn
    snap-csi-plugin:
      serviceTemplate: snap-csi-plugin
      serviceConfiguration: snap-csi-plugin
    snap-host-controller:
      serviceTemplate: snap-host-controller
      serviceConfiguration: snap-host-controller
    snap-node-driver:
      serviceTemplate: snap-node-driver
      serviceConfiguration: snap-node-driver
    doca-snap:
      serviceTemplate: doca-snap
      serviceConfiguration: doca-snap
    block-storage-dpu-plugin:
      serviceTemplate: block-storage-dpu-plugin
```

```
      serviceConfiguration: block-storage-dpu-plugin
    spdk-csi-controller:
      serviceTemplate: spdk-csi-controller
      serviceConfiguration: spdk-csi-controller
    spdk-csi-controller-dpu:
      serviceTemplate: spdk-csi-controller-dpu
      serviceConfiguration: spdk-csi-controller-dpu
  serviceChains:
    switches:
      - ports:
        - serviceInterface:
            matchLabels:
              uplink: p0
        - service:
            name: doca-hbn
            interface: p0_if
      - ports:
        - serviceInterface:
            matchLabels:
              uplink: p1
        - service:
            name: doca-hbn
            interface: p1_if
      - ports:
        - serviceInterface:
            matchLabels:
              vf: pf0vf10
        - service:
            name: doca-hbn
            interface: pf0vf10_if
      - ports:
        - serviceInterface:
            matchLabels:
              vf: pf1vf10
        - service:
            name: doca-hbn
```

```
            interface: pf1vf10_if
      - ports:
        - service:
            name: doca-snap
            interface: app_sf
            ipam:
              matchLabels:
                svc.dpu.nvidia.com/pool: storage-pool
        - service:
            name: doca-hbn
            interface: snap_if
```

Verification

These verification commands may need to be run multiple times to ensure the condition is met.

```
## Ensure the DPUServices are created and have been reconciled.
kubectl wait --for=condition=ApplicationsReconciled --namespace
dpf-operator-system dpuservices -l svc.dpu.nvidia.com/owned-by-
dpudeployment=dpf-operator-system_hbn-snap
## Ensure the DPUServiceIPAMs have been reconciled
kubectl wait --for=condition=DPUIPAMObjectReconciled --namespace
dpf-operator-system dpuserviceipam --all
## Ensure the DPUServiceInterfaces have been reconciled
kubectl wait --for=condition=ServiceInterfaceSetReconciled --
namespace dpf-operator-system dpuserviceinterface --all
## Ensure the DPUServiceChains have been reconciled
kubectl wait --for=condition=ServiceChainSetReconciled --namespace
dpf-operator-system dpuservicechain --all
```

SNAP VirtioFS

Create the DPUDeployment, DPUServiceConfig, DPUServiceTemplate and other necessary objects

A number of environment variables must be set before running this command.

```
cat manifests/04.2-dpudeployment-installation-virtiofs/*.yaml |
envsubst | kubectl apply -f -
```

This will deploy the following objects:

BFB to download BlueField Bitstream to a shared volume

```
---
apiVersion: provisioning.dpu.nvidia.com/v1alpha1
kind: BFB
metadata:
  name: bf-bundle-$TAG
  namespace: dpf-operator-system
spec:
  url: $BFB_URL
```

SNAP VirtioFS with HBN DPUFlavor to configure DPUs on provisioning

```
---
apiVersion: provisioning.dpu.nvidia.com/v1alpha1
kind: DPUFlavor
metadata:
  name: hbn-snap-virtiofs-$TAG
  namespace: dpf-operator-system
spec:
  bfcfgParameters:
  - UPDATE_ATF_UEFI=yes
  - UPDATE_DPU_OS=yes
  - WITH_NIC_FW_UPDATE=yes
  configFiles:
  - operation: override
    path: /etc/mellanox/mlnx-bf.conf
    permissions: "0644"
    raw: |
      ALLOW_SHARED_RQ="no"
      IPSEC_FULL_OFFLOAD="no"
      ENABLE_ESWITCH_MULTIPORT="yes"
      RDMA_SET_NETNS_EXCLUSIVE="no"
  - operation: override
    path: /etc/mellanox/mlnx-ovs.conf
    permissions: "0644"
    raw: |
      CREATE_OVS_BRIDGES="no"
      OVS_DOCA="yes"
  - operation: override
    path: /etc/mellanox/mlnx-sf.conf
    permissions: "0644"
    raw: ""
  grub:
    kernelParameters:
    - console=hvc0
    - console=ttyAMA0
    - earlycon=pl011,0x13010000
```

```
    - fixrttc
    - net.ifnames=0
    - biosdevname=0
    - iommu.passthrough=1
    - cgroup_no_v1=net_prio,net_cls
    - hugepagesz=2048kB
    - hugepages=5120
  nvconfig:
  - device: '*'
    parameters:
    - PF_BAR2_ENABLE=0
    - PER_PF_NUM_SF=1
    - PF_TOTAL_SF=20
    - PF_SF_BAR_SIZE=10
    - NUM_PF_MSIX_VALID=0
    - PF_NUM_PF_MSIX_VALID=1
    - PF_NUM_PF_MSIX=228
    - INTERNAL_CPU_MODEL=1
    - INTERNAL_CPU_OFFLOAD_ENGINE=0
    - SRIOV_EN=1
    - NUM_OF_VFS=46
    - LAG_RESOURCE_ALLOCATION=1
    - PCI_SWITCH_EMULATION_ENABLE=1
    - PCI_SWITCH_EMULATION_NUM_PORT=32
    - VIRTIO_FS_EMULATION_ENABLE=1
    - VIRTIO_FS_EMULATION_NUM_PF=0
    - LINK_TYPE_P1=ETH
    - LINK_TYPE_P2=ETH
  ovs:
    rawConfigScript: |
      _ovs-vsctl() {
        ovs-vsctl --no-wait --timeout 15 "$@"
      }

      _ovs-vsctl set Open_vSwitch . other_config:doca-init=true
```

```
      _ovs-vsctl set Open_vSwitch . other_config:dpdk-max-
memzones=50000
      _ovs-vsctl set Open_vSwitch . other_config:hw-offload=true
      _ovs-vsctl set Open_vSwitch . other_config:pmd-quiet-
idle=true
      _ovs-vsctl set Open_vSwitch . other_config:max-idle=20000
      _ovs-vsctl set Open_vSwitch . other_config:max-
revalidator=5000
      _ovs-vsctl --if-exists del-br ovsbr1
      _ovs-vsctl --if-exists del-br ovsbr2
      _ovs-vsctl --may-exist add-br br-sfc
      _ovs-vsctl set bridge br-sfc datapath_type=netdev
      _ovs-vsctl set bridge br-sfc fail_mode=secure
      _ovs-vsctl --may-exist add-port br-sfc p0
      _ovs-vsctl set Interface p0 type=dpdk
      _ovs-vsctl set Interface p0 mtu_request=9216
      _ovs-vsctl set Port p0 external_ids:dpf-type=physical
      _ovs-vsctl --may-exist add-port br-sfc p1
      _ovs-vsctl set Interface p1 type=dpdk
      _ovs-vsctl set Interface p1 mtu_request=9216
      _ovs-vsctl set Port p1 external_ids:dpf-type=physical
      _ovs-vsctl --may-exist add-br br-hbn
      _ovs-vsctl set bridge br-hbn datapath_type=netdev
      _ovs-vsctl set bridge br-hbn fail_mode=secure
```

DPUServiceConfiguration and DPUServiceTemplate for DOCA HBN

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceConfiguration
metadata:
  name: doca-hbn
  namespace: dpf-operator-system
spec:
  deploymentServiceName: "doca-hbn"
  serviceConfiguration:
    serviceDaemonSet:
      annotations:
        k8s.v1.cni.cncf.io/networks: |-
          [
          {"name": "iprequest", "interface": "ip_lo", "cni-args": {"poolNames": ["loopback"], "poolType": "cidrpool"}},
          {"name": "iprequest", "interface": "ip_pf0vf10", "cni-args": {"poolNames": ["pool1"], "poolType": "cidrpool", "allocateDefaultGateway": true}},
          {"name": "iprequest", "interface": "ip_pf1vf10", "cni-args": {"poolNames": ["pool2"], "poolType": "cidrpool", "allocateDefaultGateway": true}}
          ]
    helmChart:
      values:
        configuration:
          perDPUValuesYAML: |
            - hostnamePattern: "*"
              values:
                bgp_peer_group: hbn
                vrf1: RED
                vrf2: BLUE
                l3vni1: 100001
                l3vni2: 100002
            - hostnamePattern: "worker1*"
              values:
                bgp_autonomous_system: 65101
            - hostnamePattern: "worker2*"
```

```
            values:
              bgp_autonomous_system: 65201
        startupYAMLJ2: |
          - header:
              model: BLUEFIELD
              nvue-api-version: nvue_v1
              rev-id: 1.0
              version: HBN 3.0.0
          - set:
              evpn:
                enable: on
                route-advertise: {}
              bridge:
                domain:
                  br_default:
                    vlan:
                      '10':
                        vni:
                          '10': {}
              interface:
                lo:
                  ip:
                    address:
                      {{ ipaddresses.ip_lo.ip }}/32: {}
                  type: loopback
                p0_if,p1_if,pf0vf10_if,pf1vf10_if,snap_if:
                  type: swp
                  link:
                    mtu: 9000
                pf0vf10_if:
                  ip:
                    address:
                      {{ ipaddresses.ip_pf0vf10.cidr }}: {}
                    vrf: {{ config.vrf1 }}
                pf1vf10_if:
                  ip:
```

```
            address:
              {{ ipaddresses.ip_pf1vf10.cidr }}: {}
            vrf: {{ config.vrf2 }}
          snap_if:
            bridge:
              domain:
                br_default:
                  access: 10
          vlan10:
            type: svi
            vlan: 10
        nve:
          vxlan:
            arp-nd-suppress: on
            enable: on
            source:
              address: {{ ipaddresses.ip_lo.ip }}
        router:
          bgp:
            enable: on
            graceful-restart:
              mode: full
        vrf:
          default:
            router:
              bgp:
                address-family:
                  ipv4-unicast:
                    enable: on
                    redistribute:
                      connected:
                        enable: on
                    multipaths:
                      ebgp: 16
                  l2vpn-evpn:
                    enable: on
```

```
                        autonomous-system: {{
config.bgp_autonomous_system }}
                        enable: on
                        neighbor:
                          p0_if:
                            peer-group: {{ config.bgp_peer_group
}}
                            type: unnumbered
                            address-family:
                              l2vpn-evpn:
                                enable: on
                                add-path-tx: off
                          p1_if:
                            peer-group: {{ config.bgp_peer_group
}}
                            type: unnumbered
                            address-family:
                              l2vpn-evpn:
                                enable: on
                                add-path-tx: off
                        path-selection:
                          multipath:
                            aspath-ignore: on
                        peer-group:
                          {{ config.bgp_peer_group }}:
                            address-family:
                              ipv4-unicast:
                                enable: on
                              l2vpn-evpn:
                                enable: on
                            remote-as: external
                        router-id: {{ ipaddresses.ip_lo.ip }}
                    {{ config.vrf1 }}:
                      evpn:
                        enable: on
                        vni:
```

```
                        {{ config.l3vni1 }}: {}
                    router:
                      bgp:
                        address-family:
                          ipv4-unicast:
                            enable: on
                            redistribute:
                              connected:
                                enable: on
                            route-export:
                              to-evpn:
                                enable: on
                        autonomous-system: {{
config.bgp_autonomous_system }}
                        enable: on
                  {{ config.vrf2 }}:
                    evpn:
                      enable: on
                      vni:
                        {{ config.l3vni2 }}: {}
                    router:
                      bgp:
                        address-family:
                          ipv4-unicast:
                            enable: on
                            redistribute:
                              connected:
                                enable: on
                            route-export:
                              to-evpn:
                                enable: on
                        autonomous-system: {{
config.bgp_autonomous_system }}
                        enable: on

  interfaces:
```

```
  - name: p0_if
    network: mybrhbn
  - name: p1_if
    network: mybrhbn
  - name: pf0vf10_if
    network: mybrhbn
  - name: pf1vf10_if
    network: mybrhbn
  - name: snap_if
    network: mybrhbn
```

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceTemplate
metadata:
  name: doca-hbn
  namespace: dpf-operator-system
spec:
  deploymentServiceName: "doca-hbn"
  helmChart:
    source:
      repoURL: $HELM_REGISTRY_REPO_URL
      version: 1.0.5
      chart: doca-hbn
    values:
      image:
        repository: $HBN_NGC_IMAGE_URL
        tag: 3.2.1-doca3.2.1
      resources:
        memory: 6Gi
        nvidia.com/bf_sf: 5
```

DPUServiceConfiguration and DPUServiceTemplate for DOCA SNAP

```yaml
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceConfiguration
metadata:
  name: doca-snap
  namespace: dpf-operator-system
spec:
  deploymentServiceName: doca-snap
  serviceConfiguration:
    helmChart:
      values:
        dpu:
          docaSnap:
            enabled: true
            env:
              XLIO_ENABLED: "0"
            image:
              repository: $SNAP_NGC_IMAGE_URL
              tag: 1.5.0-doca3.2.0
  interfaces:
  - name: app_sf
    network: mybrsfc
```

```yaml
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceTemplate
metadata:
  name: doca-snap
  namespace: dpf-operator-system
spec:
  deploymentServiceName: doca-snap
  helmChart:
    source:
      repoURL: $REGISTRY
      version: $TAG
      chart: dpf-storage
    values:
      serviceDaemonSet:
        resources:
          memory: "2Gi"
          hugepages-2Mi: "4Gi"
          cpu: "8"
          nvidia.com/bf_sf: 1
  resourceRequirements:
    memory: "2Gi"
    hugepages-2Mi: "4Gi"
    cpu: "8"
    nvidia.com/bf_sf: 1
```

DPUServiceConfiguration and DPUServiceTemplate for SNAP Host Controller

```yaml
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceConfiguration
metadata:
  name: snap-host-controller
  namespace: dpf-operator-system
spec:
  deploymentServiceName: snap-host-controller
  upgradePolicy:
    applyNodeEffect: false
  serviceConfiguration:
    deployInCluster: true
    helmChart:
      values:
        host:
          snapHostController:
            enabled: true
            config:
              targetNamespace: dpf-operator-system
            affinity:
              nodeAffinity:
                requiredDuringSchedulingIgnoredDuringExecution:
                  nodeSelectorTerms:
                  - matchExpressions:
                      - key: "node-role.kubernetes.io/master"
                        operator: Exists
                  - matchExpressions:
                      - key: "node-role.kubernetes.io/control-plane"
                        operator: Exists
```

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceTemplate
metadata:
  name: snap-host-controller
  namespace: dpf-operator-system
spec:
  deploymentServiceName: snap-host-controller
  helmChart:
    source:
      repoURL: $REGISTRY
      version: $TAG
      chart: dpf-storage
```

DPUServiceConfiguration and DPUServiceTemplate for SNAP Node Driver

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceConfiguration
metadata:
  name: snap-node-driver
  namespace: dpf-operator-system
spec:
  deploymentServiceName: snap-node-driver
  serviceConfiguration:
    helmChart:
      values:
        dpu:
          deployCrds: true
          snapNodeDriver:
            enabled: true
```

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceTemplate
metadata:
  name: snap-node-driver
  namespace: dpf-operator-system
spec:
  deploymentServiceName: snap-node-driver
  helmChart:
    source:
      repoURL: $REGISTRY
      version: $TAG
      chart: dpf-storage
```

DPUServiceConfiguration and DPUServiceTemplate for SNAP CSI Plugin

```yaml
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceConfiguration
metadata:
  name: snap-csi-plugin
  namespace: dpf-operator-system
spec:
  deploymentServiceName: snap-csi-plugin
  upgradePolicy:
    applyNodeEffect: false
  serviceConfiguration:
    deployInCluster: true
    helmChart:
      values:
        host:
          snapCsiPlugin:
            enabled: true
            emulationMode: "virtiofs"
            controller:
              affinity:
                nodeAffinity:
                  requiredDuringSchedulingIgnoredDuringExecution:
                    nodeSelectorTerms:
                      - matchExpressions:
                          - key: "node-role.kubernetes.io/master"
                            operator: Exists
                      - matchExpressions:
                          - key: "node-role.kubernetes.io/control-plane"
                            operator: Exists
```

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceTemplate
metadata:
  name: snap-csi-plugin
  namespace: dpf-operator-system
spec:
  deploymentServiceName: snap-csi-plugin
  helmChart:
    source:
      repoURL: $REGISTRY
      version: $TAG
      chart: dpf-storage
```

DPUServiceConfiguration and DPUServiceTemplate for FS Storage DPU Plugin

```yaml
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceConfiguration
metadata:
  name: fs-storage-dpu-plugin
  namespace: dpf-operator-system
spec:
  deploymentServiceName: fs-storage-dpu-plugin
  serviceConfiguration:
    helmChart:
      values:
        dpu:
          fsStorageVendorDpuPlugin:
            enabled: true
  interfaces:
    - name: app_sf
      network: mybrsfc
```

```yaml
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceTemplate
metadata:
  name: fs-storage-dpu-plugin
  namespace: dpf-operator-system
spec:
  deploymentServiceName: fs-storage-dpu-plugin
  helmChart:
    source:
      repoURL: $REGISTRY
      version: $TAG
      chart: dpf-storage
    values:
      serviceDaemonSet:
        resources:
          nvidia.com/bf_sf: 1
  resourceRequirements:
    nvidia.com/bf_sf: 1
```

DPUServiceConfiguration, DPUServiceTemplate and DPUServiceCredentialRequest for NFS CSI Controller (host)

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceConfiguration
metadata:
  name: nfs-csi-controller
  namespace: dpf-operator-system
spec:
  deploymentServiceName: nfs-csi-controller
  upgradePolicy:
    applyNodeEffect: false
  serviceConfiguration:
    deployInCluster: true
    helmChart:
      values:
        host:
          enabled: true
          config:
            # required parameter, name of the secret that
contains connection
            # details to access the DPU cluster.
            # this secret should be created by the
DPUServiceCredentialRequest API.
            dpuClusterSecret: nfs-csi-controller-dpu-cluster-
credentials
```

```yaml
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceTemplate
metadata:
  name: nfs-csi-controller
  namespace: dpf-operator-system
spec:
  deploymentServiceName: nfs-csi-controller
  helmChart:
    source:
      repoURL: oci://ghcr.io/mellanox/dpf-storage-vendors-charts
      version: v0.2.0
      chart: nfs-csi-controller
```

```yaml
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceCredentialRequest
metadata:
  name: nfs-csi-controller-credentials
  namespace: dpf-operator-system
spec:
  duration: 24h
  serviceAccount:
    name: nfs-csi-controller-sa
    namespace: dpf-operator-system
  targetCluster:
    name: dpu-cplane-tenant1
    namespace: dpu-cplane-tenant1
  type: tokenFile
  secret:
    name: nfs-csi-controller-dpu-cluster-credentials
    namespace: dpf-operator-system
```

DPUServiceConfiguration and DPUServiceTemplate for NFS CSI Controller (DPU)

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceConfiguration
metadata:
  name: nfs-csi-controller-dpu
  namespace: dpf-operator-system
spec:
  deploymentServiceName: nfs-csi-controller-dpu
  upgradePolicy:
    applyNodeEffect: false
  serviceConfiguration:
    helmChart:
      values:
        dpu:
          enabled: true
          storageClasses:
            # List of storage classes to be created for nfs-csi
            # These StorageClass names should be used in the
StorageVendor settings
            - name: nfs-csi
              parameters:
                server: 10.0.124.1
                share: /srv/nfs/share
          rbacRoles:
            nfsCsiController:
              # the name of the service account for nfs-csi-
controller
              # this value must be aligned with the value from the
DPUServiceCredentialRequest
              serviceAccount: nfs-csi-controller-sa
```

```yaml
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceTemplate
metadata:
  name: nfs-csi-controller-dpu
  namespace: dpf-operator-system
spec:
  deploymentServiceName: nfs-csi-controller-dpu
  helmChart:
    source:
      repoURL: oci://ghcr.io/mellanox/dpf-storage-vendors-charts
      version: v0.2.0
      chart: nfs-csi-controller
```

DPUServiceIPAMs

```yaml
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceIPAM
metadata:
  name: pool1
  namespace: dpf-operator-system
spec:
  ipv4Network:
    network: "10.0.121.0/24"
    gatewayIndex: 2
    prefixSize: 29
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceIPAM
metadata:
  name: pool2
  namespace: dpf-operator-system
spec:
  ipv4Network:
    network: "10.0.122.0/24"
    gatewayIndex: 2
    prefixSize: 29
```

```yaml
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceIPAM
metadata:
  name: loopback
  namespace: dpf-operator-system
spec:
  ipv4Network:
    network: "11.0.0.0/24"
    prefixSize: 32
```

```yaml
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceIPAM
metadata:
  name: storage-pool
  namespace: dpf-operator-system
spec:
  metadata:
    labels:
      svc.dpu.nvidia.com/pool: storage-pool
  ipv4Subnet:
    subnet: "10.0.124.0/24"
    gateway: "10.0.124.1"
    perNodeIPCount: 8
```

DPUServiceInterfaces for physical ports and VFs on the DPU

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceInterface
metadata:
  name: p0
  namespace: dpf-operator-system
spec:
  template:
    spec:
      template:
        metadata:
          labels:
            uplink: "p0"
        spec:
          interfaceType: physical
          physical:
            interfaceName: p0
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceInterface
metadata:
  name: p1
  namespace: dpf-operator-system
spec:
  template:
    spec:
      template:
        metadata:
          labels:
            uplink: "p1"
        spec:
          interfaceType: physical
          physical:
            interfaceName: p1
---
```

```yaml
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceInterface
metadata:
  name: pf0vf10-rep
  namespace: dpf-operator-system
spec:
  template:
    spec:
      template:
        metadata:
          labels:
            vf: "pf0vf10"
        spec:
          interfaceType: vf
          vf:
            parentInterfaceRef: p0
            pfID: 0
            vfID: 10
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceInterface
metadata:
  name: pf1vf10-rep
  namespace: dpf-operator-system
spec:
  template:
    spec:
      template:
        metadata:
          labels:
            vf: "pf1vf10"
        spec:
          interfaceType: vf
          vf:
            parentInterfaceRef: p1
            pfID: 1
```

```
            vfID: 10
```

DPUDeployment to provision DPUs on worker nodes

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUDeployment
metadata:
  name: hbn-snap
  namespace: dpf-operator-system
spec:
  dpus:
    bfb: bf-bundle-$TAG
    flavor: hbn-snap-virtiofs-$TAG
    dpuSets:
    - nameSuffix: "dpuset1"
      dpuAnnotations:
        storage.nvidia.com/preferred-dpu: "true"
      nodeSelector:
        matchLabels:
          feature.node.kubernetes.io/dpu-enabled: "true"
  services:
    doca-hbn:
      serviceTemplate: doca-hbn
      serviceConfiguration: doca-hbn
    snap-csi-plugin:
      serviceTemplate: snap-csi-plugin
      serviceConfiguration: snap-csi-plugin
    snap-host-controller:
      serviceTemplate: snap-host-controller
      serviceConfiguration: snap-host-controller
    snap-node-driver:
      serviceTemplate: snap-node-driver
      serviceConfiguration: snap-node-driver
    doca-snap:
      serviceTemplate: doca-snap
      serviceConfiguration: doca-snap
    fs-storage-dpu-plugin:
      serviceTemplate: fs-storage-dpu-plugin
```

```
      serviceConfiguration: fs-storage-dpu-plugin
    nfs-csi-controller:
      serviceTemplate: nfs-csi-controller
      serviceConfiguration: nfs-csi-controller
    nfs-csi-controller-dpu:
      serviceTemplate: nfs-csi-controller-dpu
      serviceConfiguration: nfs-csi-controller-dpu
  serviceChains:
    switches:
      - ports:
        - serviceInterface:
            matchLabels:
              uplink: p0
        - service:
            name: doca-hbn
            interface: p0_if
      - ports:
        - serviceInterface:
            matchLabels:
              uplink: p1
        - service:
            name: doca-hbn
            interface: p1_if
      - ports:
        - serviceInterface:
            matchLabels:
              vf: pf0vf10
        - service:
            name: doca-hbn
            interface: pf0vf10_if
      - ports:
        - serviceInterface:
            matchLabels:
              vf: pf1vf10
        - service:
            name: doca-hbn
```

```
            interface: pf1vf10_if
      - ports:
        - service:
            name: doca-snap
            interface: app_sf
            ipam:
              matchLabels:
                svc.dpu.nvidia.com/pool: storage-pool
        - service:
            name: fs-storage-dpu-plugin
            interface: app_sf
            ipam:
              matchLabels:
                svc.dpu.nvidia.com/pool: storage-pool
        - service:
            name: doca-hbn
            interface: snap_if
```

Verification

These verification commands may need to be run multiple times to ensure the condition is met.

Note that the DPUService name will have a random suffix. For example, `doca-hbn-l2xsl`.

Verify the DPU and Service installation with:

```
## Ensure the DPUServices are created and have been reconciled.
kubectl wait --for=condition=ApplicationsReconciled --namespace
dpf-operator-system dpuservices -l svc.dpu.nvidia.com/owned-by-
dpudeployment=dpf-operator-system_hbn-snap
## Ensure the DPUServiceIPAMs have been reconciled
kubectl wait --for=condition=DPUIPAMObjectReconciled --namespace
dpf-operator-system dpuserviceipam --all
## Ensure the DPUServiceInterfaces have been reconciled
kubectl wait --for=condition=ServiceInterfaceSetReconciled --
namespace dpf-operator-system dpuserviceinterface --all
## Ensure the DPUServiceChains have been reconciled
kubectl wait --for=condition=ServiceChainSetReconciled --namespace
dpf-operator-system dpuservicechain --all
```

5. Add Worker Nodes and Apply Network Configuration

At this point workers should be added to the cluster. Each worker node should be configured in line with the prerequisites. As workers are added to the cluster DPUs will be provisioned and DPUServices will begin to be spun up.

You can verify the status of the DPUDeployment and its components with the following command:

```
$ kubectl -n dpf-operator-system exec deploy/dpf-operator-
controller-manager -- /dpfctl describe dpudeployments
```

Apply network configuration

This document contains network configuration that uses the P0 and P1 network PFs and VF 10 from both PFs. These interfaces are referenced in `SriovNetworkNodePolicy` and `NetworkAttachmentDefinition` objects, which are created in the next step. The

PCI addresses and names of the network PFs and VFs on the host side are likely to change after DPU provisioning, because the DPUFlavor includes the `PCI_SWITCH_EMULATION_ENABLE` firmware setting.

Before applying the network configuration, you need to identify the new names of the network PFs and VFs on the host side and set the following environment variables:

```
# contains the name of the network PF 0 on the host side, e.g.
enp8s0f0np0
export DPU_P0_PF_NAME=<REPLACE_WITH_INTERFACE_NAME>
# contains the name of the network PF 1 on the host side, e.g.
enp8s0f1np1
export DPU_P1_PF_NAME=<REPLACE_WITH_INTERFACE_NAME>
# contains the name of the network VF 10 on P0 on the host side,
e.g. enp8s0f0v10
export DPU_P0_VF10_NAME=<REPLACE_WITH_INTERFACE_NAME>
# contains the name of the network VF 10 on P1 on the host side,
e.g. enp8s0f1v10
export DPU_P1_VF10_NAME=<REPLACE_WITH_INTERFACE_NAME>
```

```
cat manifests/05-network-configuration/*.yaml | envsubst |
kubectl apply -f -
```

This will create the following objects:

SriovNetworkNodePolicy to configure VFs on P0 and P1

```yaml
---
apiVersion: sriovnetwork.openshift.io/v1
kind: SriovNetworkNodePolicy
metadata:
  name: bf3-p0-vfs
  namespace: nvidia-network-operator
spec:
  nicSelector:
    deviceID: "a2dc"
    vendor: "15b3"
    pfNames:
    - $DPU_P0_PF_NAME#2-45
  nodeSelector:
    node-role.kubernetes.io/worker: ""
  numVfs: 46
  resourceName: bf3-p0-vfs
  isRdma: true
  externallyManaged: true
  deviceType: netdevice
  linkType: eth
---
apiVersion: sriovnetwork.openshift.io/v1
kind: SriovNetworkNodePolicy
metadata:
  name: bf3-p1-vfs
  namespace: nvidia-network-operator
spec:
  nicSelector:
    deviceID: "a2dc"
    vendor: "15b3"
    pfNames:
    - $DPU_P1_PF_NAME#2-45
  nodeSelector:
    node-role.kubernetes.io/worker: ""
  numVfs: 46
```

```
    resourceName: bf3-p1-vfs
    isRdma: true
    externallyManaged: true
    deviceType: netdevice
    linkType: eth
```

NetworkAttachmentDefinition for host-device VFs

```
apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: hostdev-pf0vf10-worker1
spec:
  config: '{
    "cniVersion": "0.3.1",
    "name": "hostpf0vf10",
    "type": "host-device",
    "device": "$DPU_P0_VF10_NAME",
    "ipam": {
        "type": "static",
        "addresses": [
          {
            "address": "10.0.121.1/29"
          }
        ],
        "routes": [
          {
            "dst": "10.0.121.8/29",
            "gw": "10.0.121.2"
          }
        ]
    }
  }'
---
apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: hostdev-pf1vf10-worker1
spec:
  config: '{
    "cniVersion": "0.3.1",
    "name": "hostpf1vf10",
    "type": "host-device",
```

```
      "device": "$DPU_P1_VF10_NAME",
      "ipam": {
            "type": "static",
            "addresses": [
               {
                  "address": "10.0.122.1/29"
               }
            ],
            "routes": [
               {
                  "dst": "10.0.122.8/29",
                  "gw": "10.0.122.2"
               }
            ]
      }
   }'
---
apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
   name: hostdev-pf0vf10-worker2
spec:
   config: '{
      "cniVersion": "0.3.1",
      "name": "hostpf0vf10",
      "type": "host-device",
      "device": "$DPU_P0_VF10_NAME",
      "ipam": {
            "type": "static",
            "addresses": [
               {
                  "address": "10.0.121.9/29"
               }
            ],
            "routes": [
               {
```

```
                "dst": "10.0.121.0/29",
                "gw": "10.0.121.10"
            }
        ]
    }
  }'
---
apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: hostdev-pf1vf10-worker2
spec:
  config: '{
    "cniVersion": "0.3.1",
    "name": "hostpf1vf10",
    "type": "host-device",
    "device": "$DPU_P1_VF10_NAME",
    "ipam": {
        "type": "static",
        "addresses": [
            {
                "address": "10.0.122.9/29"
            }
        ],
        "routes": [
            {
                "dst": "10.0.122.0/29",
                "gw": "10.0.122.10"
            }
        ]
    }
  }'
```

Verify network

```
kubectl apply -f manifests/06-network-test
```

HBN functionality can be tested by pinging between the pods and services deployed in the default namespace.

6. Storage Configuration

SNAP Block (NVMe)

Apply storage configuration

```
cat manifests/07.1-storage-configuration-nvme/*.yaml | envsubst |
kubectl apply -f -
```

This will create the following objects:

DPUStorageVendor for SPDK CSI

```
---
apiVersion: storage.dpu.nvidia.com/v1alpha1
kind: DPUStorageVendor
metadata:
  name: spdk-csi
  namespace: dpf-operator-system
spec:
  storageClassName: spdkcsi-sc
  pluginName: nvidia-block
```

DPUStoragePolicy for block storage

```
---
apiVersion: storage.dpu.nvidia.com/v1alpha1
kind: DPUStoragePolicy
metadata:
  name: policy-block
  namespace: dpf-operator-system
spec:
  dpuStorageVendors:
    - spdk-csi
  selectionAlgorithm: "NumberVolumes"
  parameters: {}
```

Validate the DPUStorageVendor and DPUStoragePolicy objects are ready:

```
kubectl wait --for=condition=Ready --namespace dpf-operator-system
dpustoragevendors --all
kubectl wait --for=condition=Ready --namespace dpf-operator-system
dpustoragepolicies --all
```

Run Workloads to test NVMe block storage

Deploy storage test pods that request a block volume from SNAP NVMe.

```
kubectl apply -f manifests/08.1-storage-test-nvme
```

Storage functionality can be tested by verifying the block device is attached inside the pod and performing I/O.

SNAP VirtioFS

```
cat manifests/07.2-storage-configuration-virtiofs/*.yaml |
envsubst | kubectl apply -f -
```

This will create the following objects:

DPUStorageVendor for NFS CSI

```
---
apiVersion: storage.dpu.nvidia.com/v1alpha1
kind: DPUStorageVendor
metadata:
  name: nfs-csi
  namespace: dpf-operator-system
spec:
  storageClassName: nfs-csi
  pluginName: nvidia-fs
```

DPUStoragePolicy for filesystem policy

```
---
apiVersion: storage.dpu.nvidia.com/v1alpha1
kind: DPUStoragePolicy
metadata:
  name: policy-fs
  namespace: dpf-operator-system
spec:
  dpuStorageVendors:
    - nfs-csi
  selectionAlgorithm: "NumberVolumes"
  parameters: {}
```

Validate the DPUStorageVendor and DPUStoragePolicy objects are ready:

```
kubectl wait --for=condition=Ready --namespace dpf-operator-system
dpustoragevendors --all
kubectl wait --for=condition=Ready --namespace dpf-operator-system
dpustoragepolicies --all
```

Run Workloads to test VirtioFS storage

Deploy storage test pods that mount a storage volume provided by SNAP VirtioFS.

```
kubectl apply -f manifests/08.2-storage-test-virtiofs
```

Storage functionality can be tested by writing and reading data to the mounted volume.

Uninstall

This section describes how to clean up the DPF components installed in this guide. It is recommended to run this section only after the DPF Operator and DPUCluster are no longer needed.

Delete test workloads

```
kubectl delete -f manifests/06-network-test --wait --ignore-not-found=true
kubectl delete -f manifests/08.2-storage-test-virtiofs --wait --ignore-not-found=true
kubectl delete -f manifests/08.1-storage-test-nvme --wait --ignore-not-found=true
# delete all PVCs created by StatefulSets
kubectl delete pvc --selector=app=storage-test-pod-nvme-hotplug-pf --wait -n default --ignore-not-found=true
kubectl delete pvc --selector=app=storage-test-pod-virtiofs-hotplug-pf --wait -n default --ignore-not-found=true
kubectl delete -n dpf-operator-system dpuvolumeattachment --all --wait
kubectl delete -n dpf-operator-system dpuvolume --all --wait
# delete storage configuration
kubectl delete -f manifests/07.1-storage-configuration-nvme --wait --ignore-not-found=true
kubectl delete -f manifests/07.2-storage-configuration-virtiofs --wait --ignore-not-found=true
```

Delete DPF CNI acceleration components

```
kubectl delete -f manifests/05-network-configuration --wait --
ignore-not-found=true
kubectl delete -f manifests/03-enable-accelerated-interfaces --
wait --ignore-not-found=true
helm uninstall -n nvidia-network-operator network-operator --wait
```

Delete the DPF Operator system and DPF Operator

```
kubectl delete -n dpf-operator-system dpfoperatorconfig
dpfoperatorconfig --wait
helm uninstall -n dpf-operator-system dpf-operator --wait
```

Delete DPF Operator PVC

```
kubectl -n dpf-operator-system delete pvc bfb-pvc
kubectl delete pv bfb-pv
```

Note: there can be a race condition with deleting the underlying Kamaji cluster which runs the DPU cluster control plane in this guide. If that happens it may be necessary to remove finalizers manually from `DPUCluster` and `Datastore` objects.

Zero Trust Deployment

Overview

The **Zero Trust** deployment scenario represents DPF for zero trust environments, where the DPU is treated as a **host accelerator card** and is **managed by DPF via BMC and**

Redfish. In this deployment model, host machines remain outside the DPF management cluster, while the DPU serves as an integrated acceleration resource that enhances the host's capabilities.

This architecture provides secure isolation between the management plane and host systems. DPU services are managed through Kubernetes APIs within the DPF cluster, while host workloads operate independently on their respective machines.

Redfish is used to manage the DPU hardware, including firmware updates and configuration changes.

Use Cases

The Zero Trust model supports various use cases, each optimized for specific networking and acceleration requirements.

Each use case may require tailored configurations on the DPU for optimal performance. Below are the validated user guides with corresponding deployment guides.

HBN

- Host Based Networking

DPU Services	Comments
DOCA Host-Based Networking (HBN)	Accelerates underlay BGP routing with ECMP and EVPN-based overlays

VPC

- Virtual Private Cloud

DPU Services	Comments
VPC OVN-Controller	OVN-Controller for VPC management and networking
VPC OVN-Node	OVN-Node for VPC networking and connectivity

HBN and SNAP Storage

- HBN and SNAP Storage

DPU Services	Comments
DOCA Host-Based Networking (HBN)	Accelerates underlay BGP routing with ECMP and EVPN-based overlays
DOCA SNAP	Provides SNAP storage services for the DPUs

Passthrough

- Passthrough

No DPUServices are necessary for this use case.

Prerequisites

Before deploying DPF in zero trust mode, ensure you have the necessary hardware and software prerequisites in place. This includes compatible DPU hardware, host operating system configurations, and network settings. For detailed requirements, refer to the prerequisites guide.

Deployment Steps

To deploy DPF in zero trust mode:

1. **Setup Prerequisites**: Ensure all hardware, software, and network requirements are met
2. **Choose Use Case**: Select the appropriate deployment scenario based on your requirements
3. **Follow Guide**: Use the specific use case documentation for step-by-step deployment
4. **Verify Deployment**: Validate the installation and perform basic functionality tests

Support and Troubleshooting

For assistance with zero trust deployments:

- Review the specific use case documentation
- Check the DPF troubleshooting guide
- Consult the release notes for known issues
- Engage with the NVIDIA support team for further assistance

Next Steps

Once you've familiarized yourself with the zero trust concept, proceed to:

1. Review the prerequisites.
2. Select an appropriate use case.
3. Begin your DPF zero trust deployment.

DPF Prerequisites for Zero Trust

Before deploying DPF, ensure the following prerequisites are met:

- DPF System Prerequisites for Zero Trust
- DPF Operator Helm Prerequisites

DPF System Prerequisites for Zero Trust

DPF makes a number of assumptions about the hardware, software and networking of the machines it runs on. Some of the specific user guides add their own requirements.

Hardware Setup

There are high availability control plane machines running DPF and workload machines.

Control Plane Machines

Each control plane machine:

- May be virtualized
- x86_64 architecture
- 16 GB RAM
- 8 CPUs
- DPUs are not installed

Workload Machines

Each workload machine has the following characteristics:

- Bare metal - no virtualization
- Any number of DPUs

DPUs

- Bluefield 3
- 32 GB memory
- Flashed with NVIDIA BFB with DOCA version 2.5 or higher
- out-of-band management port must be connected to the management network

System Software Setup

Control Plane Machines

- NFS client packages - i.e. `nfs-common`
- NFS server available with `/mnt/dpf_share` readable and writable by any user

Kubernetes

- Kubernetes 1.32 - 1.34
- Control plane nodes have the labels `"node-role.kubernetes.io/control-plane" : ""`

Network Setup

- All nodes must have internet access to be able to pull images - included the DPUs
- Virtual IP from the management subnet reserved for internal DPF usage
- The DPU out-of-band physical interface must be connected with the DPF control planes
- The control plane nodes hosting the DPU control plane pods must be located on the same L2 broadcast domain
- The out-of-band management fabric on which control plane nodes are connected should allow MultiCast traffic (used for VRRP)

Zero Trust Use Cases

Zero Trust Use Cases

The **Zero Trust** scenario supports the following use cases, each tailored to specific networking and acceleration needs. Each use case may require specific configurations on the DPU for optimal performance.

- Host Based Networking
- Host Based Networking and SNAP Storage
- Virtual Private Cloud
- Passthrough

DPU Passthrough in DPF Zero Trust

> **Note**
>
> Follow this guide from the source GitHub repo at github.com/NVIDIA/doca-platform and moving to the `docs/public/user-guides/zero-trust/use-cases/passthrough/README.md` for better formatting of the code.

This configuration provides instructions for deploying the NVIDIA DOCA Platform Framework (DPF) on high-performance, bare-metal infrastructure in Zero Trust mode, utilizing DPU BMC and Redfish. It focuses on provisioning NVIDIA® BlueField®-3 DPUs using DPF and enabling them to act as passthrough devices.

Prerequisites

This guide should be run by cloning the repo from github.com/NVIDIA/doca-platform and moving to the `docs/public/user-guides/zero-trust/use-cases/passthrough` directory.

The system is set up as described in the prerequisites.

Software Prerequisites

The following tools must be installed on the machine where the commands contained in this guide run:

- kubectl
- helm
- envsubst

Installation Guide

0. Required Variables

The following variables are required by this guide. A sensible default is provided where it makes sense, but many will be specific to the target infrastructure.

Commands in this guide are run in the same directory that contains this readme.

Environment variables file

```
## IP Address for the Kubernetes API server of the target cluster
on which DPF is installed.
## This should never include a scheme or a port.
## e.g. 10.10.10.10
export TARGETCLUSTER_API_SERVER_HOST=

## Virtual IP used by the load balancer for the DPU Cluster. Must
be a reserved IP from the management subnet and not
## allocated by DHCP.
export DPUCLUSTER_VIP=

## Interface on which the DPUCluster load balancer will listen.
Should be the management interface of the control plane
## node.
export DPUCLUSTER_INTERFACE=

## IP address to the NFS server used as storage for the BFB.
export NFS_SERVER_IP=

## The DPF REGISTRY is the Helm repository URL where the DPF
Operator Chart resides.
## Usually this is the NVIDIA Helm NGC registry. For development
purposes, this can be set to a different repository.
export REGISTRY=https://helm.ngc.nvidia.com/nvidia/doca

## The DPF TAG is the version of the DPF components which will be
deployed in this guide.
export TAG=v25.10.0

## URL to the BFB used in the `bfb.yaml` and linked by the
DPUSet.
export BFB_URL="https://content.mellanox.com/BlueField/BFBs/Ubuntu24.04/bf-bundle-3.2.1-
34_25.11_ubuntu-24.04_64k_prod.bfb"

## IP_RANGE_START and IP_RANGE_END
```

```
## These define the IP range for DPU discovery via Redfish/BMC
interfaces
## Example: If your DPUs have BMC IPs in range 192.168.1.100-110
## export IP_RANGE_START=192.168.1.100
## export IP_RANGE_END=192.168.1.110
export IP_RANGE_START=

export IP_RANGE_END=

# The password used for DPU BMC root login, must be the same for
all DPUs
export BMC_ROOT_PASSWORD=
```

Modify the variables in `manifests/00-env-vars/envvars.env` to fit your environment, then source the file:

```
source manifests/00-env-vars/envvars.env
```

1. DPF Operator Installation

Create storage required by the DPF Operator

A number of environment variables must be set before running this command.

```
kubectl create ns dpf-operator-system
cat manifests/01-dpf-operator-installation/*.yaml | envsubst |
kubectl apply -f -
```

This deploys the following objects:

PersistentVolume and PersistentVolumeClaim for the provisioning controller

```
---
apiVersion: v1
kind: PersistentVolume
metadata:
  name: bfb-pv
spec:
  capacity:
    storage: 10Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteMany
  nfs:
    path: /mnt/dpf_share/bfb
    server: $NFS_SERVER_IP
  persistentVolumeReclaimPolicy: Delete
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: bfb-pvc
  namespace: dpf-operator-system
spec:
  accessModes:
  - ReadWriteMany
  resources:
    requests:
      storage: 10Gi
  volumeMode: Filesystem
  storageClassName: ""
```

Create DPU BMC shared password secret

In Zero Trust mode, provisioning DPUs requires authentication with Redfish. In order to do that, you must set the same root password to access the BMC for all DPUs DPF is going

to manage.

For more information on how to set the BMC root password refer to BlueField DPU Administrator Quick Start Guide

The password is provided to DPF by creating the following secret:

```
kubectl create secret generic -n dpf-operator-system bmc-shared-
password --from-literal=password=$BMC_ROOT_PASSWORD
```

Additional Dependencies

Before deploying the DPF Operator, ensure that Helm is properly configured according to the Helm prerequisites.

> ⚠ **Warning**
>
> This is a critical prerequisite step that must be completed for the DPF Operator to function properly.

Deploy the DPF Operator

A number of environment variables must be set before running this command.

HTTP Registry (default)

If the $REGISTRY is an HTTP Registry (default value) use this command:

```
helm repo add --force-update dpf-repository ${REGISTRY}
helm repo update
helm upgrade --install -n dpf-operator-system dpf-operator dpf-
repository/dpf-operator --version=$TAG
```

OCI Registry

For development purposes, if the $REGISTRY is an OCI Registry use this command:

```
helm upgrade --install -n dpf-operator-system dpf-operator
$REGISTRY/dpf-operator --version=$TAG
```

Verification

These verification commands may need to be run multiple times to ensure the condition is met.

Verify the DPF Operator installation with:

```
## Ensure the DPF Operator deployment is available.
kubectl rollout status deployment --namespace dpf-operator-system
dpf-operator-controller-manager
## Ensure all pods in the DPF Operator system are ready.
kubectl wait --for=condition=ready --namespace dpf-operator-system
pods --all
```

2. DPF System Installation

This section involves creating the DPF system components and some basic infrastructure required for a functioning DPF-enabled cluster.

Deploy the DPF System components

A number of environment variables must be set before running this command.

```
kubectl create ns dpu-cplane-tenant1
cat manifests/02-dpf-system-installation/*.yaml | envsubst |
kubectl apply -f -
```

This will create the following objects:

DPFOperatorConfig to install the DPF System components

```
---
apiVersion: operator.dpu.nvidia.com/v1alpha1
kind: DPFOperatorConfig
metadata:
  name: dpfoperatorconfig
  namespace: dpf-operator-system
spec:
  dpuDetector:
    disable: true
  provisioningController:
    bfbPVCName: "bfb-pvc"
    dmsTimeout: 900
    installInterface:
      installViaRedfish:
        # Set this to the IP of one of your control plane nodes +
8080 port
        bfbRegistryAddress: "$TARGETCLUSTER_API_SERVER_HOST:8080"
        skipDPUNodeDiscovery: false
  kamajiClusterManager:
    disable: false
```

DPUCluster to serve as Kubernetes control plane for DPU nodes

```
---
apiVersion: provisioning.dpu.nvidia.com/v1alpha1
kind: DPUCluster
metadata:
  name: dpu-cplane-tenant1
  namespace: dpu-cplane-tenant1
spec:
  type: kamaji
  maxNodes: 10
  clusterEndpoint:
    # deploy keepalived instances on the nodes that match the
given nodeSelector.
    keepalived:
      # interface on which keepalived will listen. Should be the oob
interface of the control plane node.
      interface: $DPUCLUSTER_INTERFACE
      # Virtual IP reserved for the DPU Cluster load balancer.
Must not be allocatable by DHCP.
      vip: $DPUCLUSTER_VIP
      # virtualRouterID must be in range [1,255], make sure the
given virtualRouterID does not duplicate with any existing
keepalived process running on the host
      virtualRouterID: 126
      nodeSelector:
        node-role.kubernetes.io/control-plane: ""
```

DPUDiscovery to discover DPUDevices or DPUNodes

```
---
apiVersion: provisioning.dpu.nvidia.com/v1alpha1
kind: DPUDiscovery
metadata:
  name: dpu-discovery
  namespace: dpf-operator-system
spec:
  ipRangeSpec:
    ipRange:
      startIP: $IP_RANGE_START
      endIP: $IP_RANGE_END
```

Verification

These verification commands may need to be run multiple times to ensure the condition is met.

Verify the DPF System with:

```
## Ensure the provisioning and DPUService controller manager
deployments are available.
kubectl rollout status deployment --namespace dpf-operator-system
dpf-provisioning-controller-manager dpuservice-controller-manager
## Ensure all other deployments in the DPF Operator system are
Available.
kubectl rollout status deployment --namespace dpf-operator-system
## Ensure bfb registry daemonset is available
kubectl rollout status daemonset --namespace dpf-operator-system
bfb-registry
## Ensure the DPUCluster is ready for nodes to join.
kubectl wait --for=condition=ready --namespace dpu-cplane-tenant1
dpucluster --all
```

3. DPU Provisioning and Interface Plumbing

> ⚠ **Warning**
>
> In case more than 1 DPU exists per node, the relevant selector should be applied in the DPUSet to select the appropriate DPU. See DPUSet - DPU Selection to understand more about the selectors.

In this step we provision our DPUs and we do the nessecary interface plumbing to enable the DPU to act as a passthrough device.

The user is expected to create a DPUSet object to provision the DPUs and a DPUServiceChain to enable the nessecary connectivity between the host and DPU interfaces.

> Check the DPUSet documentation and DPUServiceChain documentation for more information about these objects.

Create the BFB, DPUSet and DPUServiceChain

A number of environment variables must be set before running this command.

```
cat manifests/03-dpf-object-installation/*.yaml | envsubst |
kubectl apply -f -
```

This will deploy the following objects:

BFB to download the BFB to a shared volume

```
---
apiVersion: provisioning.dpu.nvidia.com/v1alpha1
kind: BFB
metadata:
  name: bf-bundle-$TAG
  namespace: dpf-operator-system
spec:
  url: $BFB_URL
```

DPUFlavor used for provisioning the DPUs

```
---
apiVersion: provisioning.dpu.nvidia.com/v1alpha1
kind: DPUFlavor
metadata:
  name: passthrough-$TAG
  namespace: dpf-operator-system
spec:
  dpuMode: zero-trust
  grub:
    kernelParameters:
    - console=hvc0
    - console=ttyAMA0
    - earlycon=pl011,0x13010000
    - fixrttc
    - net.ifnames=0
    - biosdevname=0
    - iommu.passthrough=1
    - cgroup_no_v1=net_prio,net_cls
    - hugepagesz=2048kB
    - hugepages=3072
  nvconfig:
  - device: "*"
    parameters:
    - PF_BAR2_ENABLE=0
    - PER_PF_NUM_SF=1
    - PF_TOTAL_SF=20
    - PF_SF_BAR_SIZE=10
    - NUM_PF_MSIX_VALID=0
    - PF_NUM_PF_MSIX_VALID=1
    - PF_NUM_PF_MSIX=228
    - INTERNAL_CPU_MODEL=1
    - INTERNAL_CPU_OFFLOAD_ENGINE=0
    - SRIOV_EN=1
    - NUM_OF_VFS=46
    - LAG_RESOURCE_ALLOCATION=1
```

```
    - LINK_TYPE_P1=ETH
    - LINK_TYPE_P2=ETH
  ovs:
    rawConfigScript: |
      _ovs-vsctl() {
        ovs-vsctl --no-wait --timeout 15 "$@"
      }

      _ovs-vsctl set Open_vSwitch . other_config:doca-init=true
      _ovs-vsctl set Open_vSwitch . other_config:dpdk-max-
memzones=50000
      _ovs-vsctl set Open_vSwitch . other_config:hw-offload=true
      _ovs-vsctl set Open_vSwitch . other_config:pmd-quiet-
idle=true
      _ovs-vsctl set Open_vSwitch . other_config:max-idle=20000
      _ovs-vsctl set Open_vSwitch . other_config:max-
revalidator=5000
      _ovs-vsctl set Open_vSwitch . other_config:ctl-pipe-
size=1024
      _ovs-vsctl --if-exists del-br ovsbr1
      _ovs-vsctl --if-exists del-br ovsbr2
      _ovs-vsctl --may-exist add-br br-sfc
      _ovs-vsctl set bridge br-sfc datapath_type=netdev
      _ovs-vsctl set bridge br-sfc fail_mode=secure
      _ovs-vsctl --may-exist add-port br-sfc p0
      _ovs-vsctl set Interface p0 type=dpdk
      _ovs-vsctl set Interface p0 mtu_request=9216
      _ovs-vsctl set Port p0 external_ids:dpf-type=physical

  bfcfgParameters:
  - UPDATE_ATF_UEFI=yes
  - UPDATE_DPU_OS=yes
  - WITH_NIC_FW_UPDATE=yes

  configFiles:
  - path: /etc/mellanox/mlnx-bf.conf
```

```
    operation: override
    raw: |
        ALLOW_SHARED_RQ="no"
        IPSEC_FULL_OFFLOAD="no"
        ENABLE_ESWITCH_MULTIPORT="yes"
    permissions: "0644"
  - path: /etc/mellanox/mlnx-ovs.conf
    operation: override
    raw: |
        CREATE_OVS_BRIDGES="no"
        OVS_DOCA="yes"
    permissions: "0644"
  - path: /etc/mellanox/mlnx-sf.conf
    operation: override
    raw: ""
    permissions: "0644"
```

DPUSet to provision DPUs on worker nodes

```yaml
---
apiVersion: provisioning.dpu.nvidia.com/v1alpha1
kind: DPUSet
metadata:
  name: passthrough
  namespace: dpf-operator-system
spec:
  dpuNodeSelector:
    matchLabels:
      feature.node.kubernetes.io/dpu-enabled: "true"
  dpuTemplate:
    spec:
      dpuFlavor: passthrough-$TAG
      bfb:
        name: bf-bundle-$TAG
      nodeEffect:
        noEffect: true
```

DPUServiceInterfaces used by the DPUServiceChain

```yaml
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceInterface
metadata:
  name: p0
  namespace: dpf-operator-system
spec:
  template:
    spec:
      template:
        metadata:
          labels:
            interface: "p0"
        spec:
          interfaceType: physical
          physical:
            interfaceName: p0
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceInterface
metadata:
  name: p1
  namespace: dpf-operator-system
spec:
  template:
    spec:
      template:
        metadata:
          labels:
            interface: "p1"
        spec:
          interfaceType: physical
          physical:
            interfaceName: p1
---
```

```yaml
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceInterface
metadata:
  name: pf0hpf
  namespace: dpf-operator-system
spec:
  template:
    spec:
      template:
        metadata:
          labels:
            interface: "pf0hpf"
        spec:
          interfaceType: pf
          pf:
            pfID: 0
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceInterface
metadata:
  name: pf1hpf
  namespace: dpf-operator-system
spec:
  template:
    spec:
      template:
        metadata:
          labels:
            interface: "pf1hpf"
        spec:
          interfaceType: pf
          pf:
            pfID: 1
```

DPUServiceChain to make the device act as passthrough device

```yaml
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceChain
metadata:
  name: passthrough
  namespace: dpf-operator-system
spec:
  template:
    spec:
      template:
        spec:
          switches:
            - ports:
              - serviceInterface:
                  matchLabels:
                    interface: p0
              - serviceInterface:
                  matchLabels:
                    interface: pf0hpf
            - ports:
              - serviceInterface:
                  matchLabels:
                    interface: p1
              - serviceInterface:
                  matchLabels:
                    interface: pf1hpf
```

Verification

The following verification commands may need to be run multiple times to ensure the condition is met.

```
## Ensure the DPUServiceChain is ready
kubectl wait --for=condition=ready --namespace dpf-operator-system
dpuservicechain passthrough
## Ensure the DPUServiceInterfaces are ready
kubectl wait --for=condition=ready --namespace dpf-operator-system
dpuserviceinterface p0 p1 pf0hpf pf1hpf
## Ensure the BFB is ready
kubectl wait --for="jsonpath={.status.phase}=Ready" --namespace dpf-operator-
system bfb bf-bundle
## Ensure the DPUs have the condition Initialized (this may take
time)
kubectl wait --for=condition=Initialized --namespace dpf-operator-
system dpu --all
```

or with `dpfctl`:

```
$ kubectl -n dpf-operator-system exec deploy/dpf-operator-
controller-manager -- /dpfctl describe dpusets
NAME                                                NAMESPACE
STATUS       REASON         SINCE  MESSAGE
DPFOperatorConfig/dpfoperatorconfig                 dpf-operator-
system  Ready: True  Success        24m
  DPUServiceChains
    DPUServiceChain/passthrough                     dpf-operator-
system  Ready: True  Success        7s
  DPUServiceInterfaces
    4 DPUServiceInterfaces...                       dpf-operator-
system  Ready: True  Success        78m    See p0, p1, pf0hpf,
pf1hpf
  DPUSets
    DPUSet/passthrough                              dpf-operator-
system
      BFB/bf-bundle                                 dpf-operator-
system  Ready: True  Ready          78m    File: bf-bundle-3.2.1-
34_25.11_ubuntu-24.04_64k_prod.bfb, DOCA: 3.2.1
      DPUs
        DPU/dpu-node-mt2402xz0f6v-mt2402xz0f6v  dpf-operator-
system
                      Ready
False        OS Installing  1s
        DPU/dpu-node-mt2404xz0c98-mt2404xz0c98  dpf-operator-
system
                      Ready
False        OS Installing  1s
```

Making the DPUs Ready

In order to make the DPUs ready, we will need to manually power cycle the host. This operation should be done in the most graceful manner by gracefully shutting down the Host and DPU, powering off the server and then powering it on to avoid corruption. This

should happen when the object gives us the signal. The described flow can be automated by the admin depending on the infrastructure.

The following verification command may need to be run multiple times to ensure the condition is met.

```
## Ensure the DPUs have the condition
WaitingForManualPowerCycleOrReboot (this may take time)
kubectl wait --for=condition=WaitingForManualPowerCycleOrReboot --
namespace dpf-operator-system dpu --all
```

or with `dpfctl`:

```
$ kubectl -n dpf-operator-system exec deploy/dpf-operator-
controller-manager -- /dpfctl describe dpusets
NAME                                              NAMESPACE
STATUS       REASON                               SINCE  MESSAGE
DPFOperatorConfig/dpfoperatorconfig               dpf-operator-
system
              Ready
False        Pending                               36m    The
following conditions are not ready:

* SystemComponentsReady
              SystemComponentsReady
False        Error                                 35m    System
components must be ready for DPF Operator to continue:

* nvidia-k8s-ipam: DPUService dpf-operator-system/nvidia-k8s-ipam
is not ready
  DPUServiceChains
    DPUServiceChain/passthrough                   dpf-operator-
system  Ready: True  Success                            34m
  DPUServiceInterfaces
    4 DPUServiceInterfaces...                     dpf-operator-
system  Ready: True  Success                            34m
See p0, p1, pf0hpf, pf1hpf
  DPUSets
    DPUSet/passthrough                            dpf-operator-
system
      BFB/bf-bundle                               dpf-operator-
system  Ready: True  Ready                              34m
File: bf-bundle-3.2.1-34_25.11_ubuntu-24.04_64k_prod.bfb, DOCA:
3.2.1
      DPUs
        DPU/dpu-node-mt2402xz0f6v-mt2402xz0f6v  dpf-operator-
system
```

```
                      Rebooted
False        WaitingForManualPowerCycleOrReboot  12m
                      Ready
False        Rebooting                           12m
        DPU/dpu-node-mt2404xz0c98-mt2404xz0c98  dpf-operator-
system
                      Rebooted
False        WaitingForManualPowerCycleOrReboot  10m
                      Ready
False        Rebooting                           10m
```

At this point, we have to power cycle the hosts. Once **all the hosts are back online**, we have to remove an annotation from the DPUNodes. The user can choose to remove this annotation node by node but to make it simpler in this guide, we do that all at once.

```
kubectl annotate dpunodes -n dpf-operator-system --all
provisioning.dpu.nvidia.com/dpunode-external-reboot-required-
```

After this is done, we should expect that all DPUs become Ready:

```
kubectl wait --for="jsonpath={.status.phase}=Ready" --namespace dpf-operator-
system dpu --all
```

or with dpfctl:

```
$ kubectl -n dpf-operator-system exec deploy/dpf-operator-
controller-manager -- /dpfctl describe dpusets
NAME                                  NAMESPACE            STATUS
REASON    SINCE  MESSAGE
DPFOperatorConfig/dpfoperatorconfig  dpf-operator-system  Ready:
True  Success   42s
  DPUServiceChains
    DPUServiceChain/passthrough      dpf-operator-system  Ready:
True  Success   54m
  DPUServiceInterfaces
    4 DPUServiceInterfaces...        dpf-operator-system  Ready:
True  Success   54m    See p0, p1, pf0hpf, pf1hpf
  DPUSets
    DPUSet/passthrough               dpf-operator-system
      BFB/bf-bundle                  dpf-operator-system  Ready:
True  Ready     54m    File: bf-bundle-3.2.1-34_25.11_ubuntu-
24.04_64k_prod.bfb, DOCA: 3.2.1
      DPUs
        2 DPUs...                    dpf-operator-system  Ready:
True  DPUReady  2m33s  See dpu-node-mt2402xz0f6v-mt2402xz0f6v,
dpu-node-mt2404xz0c98-mt2404xz0c98
```

Test Traffic

After the DPUs are provisioned and the rest of the objects are Ready, we can test traffic by assigning an IP on one of the PFs on the host for each DPU, and run a simple ping. This assumes that the high speed ports of the DPUs are connected and the DPUs can reach each other. Assuming the pf is named `ens5f0np0` then:

Host 1:

```
ip link set dev ens5f0np0 up
ip addr add 192.168.1.1/24 dev ens5f0np0
```

Host 2:

```
ip link set dev ens5f0np0 up
ip addr add 192.168.1.2/24 dev ens5f0np0
```

From Host 1:

```
$ ping 192.168.1.2 -c3
PING 192.168.1.2 (192.168.1.2) 56(84) bytes of data.
64 bytes from 192.168.1.2: icmp_seq=1 ttl=64 time=0.387 ms
64 bytes from 192.168.1.2: icmp_seq=2 ttl=64 time=0.344 ms
64 bytes from 192.168.1.2: icmp_seq=3 ttl=64 time=0.396 ms

--- 192.168.1.2 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2053ms
rtt min/avg/max/mdev = 0.344/0.375/0.396/0.022 ms
```

Uninstall

This section covers only the DPF related components and not the prerequisites as these must be managed by the admin.

Delete the DPF Operator system and DPF Operator

```
kubectl delete -n dpf-operator-system dpfoperatorconfig
dpfoperatorconfig --wait
helm uninstall -n dpf-operator-system dpf-operator --wait
```

Delete DPF Operator PVC

```
kubectl -n dpf-operator-system delete pvc bfb-pvc
kubectl delete pv bfb-pv
```

Note: there can be a race condition with deleting the underlying Kamaji cluster which runs the DPU cluster control plane in this guide. If that happens it may be necessary to remove finalizers manually from `DPUCluster` and `Datastore` objects.

HBN and SNAP Storage in DPF Zero Trust

> **Note**
>
> Follow this guide from the source GitHub repo at github.com/NVIDIA/doca-platform and moving to the `docs/public/user-guides/zero-trust/use-cases/hbn-snap/README.md` for better formatting of the code.

This configuration provides instructions for deploying the NVIDIA DOCA Platform Framework (DPF) on high-performance, bare-metal infrastructure in Zero Trust mode, utilizing DPU BMC and Redfish. It focuses on provisioning NVIDIA® BlueField®-3 DPUs

using DPF, installing the HBN DPUService on those DPUs and enabling SNAP Storage on those DPUs.

Prerequisites

This guide should be run by cloning the repo from github.com/NVIDIA/doca-platform and moving to the `docs/public/user-guides/zero-trust/use-cases/hbn-snap` directory.

The system is set up as described in the prerequisites.

In addition, for this use case, the Top of Rack switch(ToR) should be configured to support unnumbered BGP towards the two ports of the DPU, where HBN will act as peer, and advertise routes over BGP to allow for ECMP from the DPU. The storage traffic between the DPU and the remote storage system goes through a VXLAN interface. Additional information about the required switch configuration can be found in the Technology Preview for DPF Zero Trust (DPF-ZT) with SNAP DPU Service in virtio-fs mode.

This guide includes examples for both SNAP Block (NVMe) and SNAP VirtioFS Storage. Depending on the storage type you want to deploy, you need to ensure that the following prerequisites are met:

SNAP Block (NVMe) Prerequisites

- An remote SPDK target should be set up to provide persistent storage for SNAP Block Storage.
- The SPDK target should be reachable from the DPUs.
- The management interface of the SPDK target should be reachable from the control plane nodes.
- Make sure to check Host OS Configuration Section in SNAP service documentation to validate the host OS configuration on the worker nodes.

SNAP VirtioFS Prerequisites

- An external NFS server is required to provide persistent storage for SNAP VirtioFS.
- The NFS server must be reachable by both the SNAP DPU service and the nvidia-fs DPU plugin.

- The NFS service must also be accessible from the DPF control plane nodes to ensure proper operation.
- Make sure to check Host OS Configuration Section in SNAP VirtioFS service documentation to validate the host OS configuration on the worker nodes.

Software Prerequisites

The following tools must be installed on the machine where the commands contained in this guide run:

- kubectl
- helm
- envsubst

Installation Guide

This guide assumes that the setup includes only 2 workers with DPUs. If your setup has more than 2 workers, then you will need to set additional variables to enable the rest of the DPUs.

0. Required Variables

The following variables are required by this guide. A sensible default is provided where it makes sense, but many will be specific to the target infrastructure.

Commands in this guide are run in the same directory that contains this readme.

Environment variables file

```
## IP Address for the Kubernetes API server of the target cluster
on which DPF is installed.
## This should never include a scheme or a port.
## e.g. 10.10.10.10
export TARGETCLUSTER_API_SERVER_HOST=

## Virtual IP used by the load balancer for the DPU Cluster. Must
be a reserved IP from the management subnet and not
## allocated by DHCP.
export DPUCLUSTER_VIP=

## Interface on which the DPUCluster load balancer will listen.
Should be the management interface of the control plane node.
export DPUCLUSTER_INTERFACE=

## IP address to the NFS server used as storage for the BFB.
export NFS_SERVER_IP=

## The repository URL for the NVIDIA Helm chart registry.
## Usually this is the NVIDIA Helm NGC registry. For development
purposes, this can be set to a different repository.
export HELM_REGISTRY_REPO_URL=https://helm.ngc.nvidia.com/nvidia/doca

## The repository URL for the HBN container image.
## Usually this is the NVIDIA NGC registry. For development
purposes, this can be set to a different repository.
export HBN_NGC_IMAGE_URL=nvcr.io/nvidia/doca/doca_hbn

## The repository URL for the SNAP VFS container image.
## Usually this is the NVIDIA NGC registry. For development
purposes, this can be set to a different repository.
export SNAP_NGC_IMAGE_URL=nvcr.io/nvidia/doca/doca_vfs

## The DPF REGISTRY is the Helm repository URL where the DPF
Operator Chart resides.
```

```
## Usually this is the NVIDIA Helm NGC registry. For development
purposes, this can be set to a different repository.
export REGISTRY=https://helm.ngc.nvidia.com/nvidia/doca

## The DPF TAG is the version of the DPF components which will be
deployed in this guide.
export TAG=v25.10.0

## URL to the BFB used in the `bfb.yaml` and linked by the
DPUSet.
export BFB_URL="https://content.mellanox.com/BlueField/BFBs/Ubuntu24.04/bf-bundle-3.2.1-
34_25.11_ubuntu-24.04_64k_prod.bfb"

## IP_RANGE_START and IP_RANGE_END
## These define the IP range for DPU discovery via Redfish/BMC
interfaces
## Example: If your DPUs have BMC IPs in range 192.168.1.100-110
## export IP_RANGE_START=192.168.1.100
## export IP_RANGE_END=192.168.1.110
export IP_RANGE_START=

export IP_RANGE_END=

# The password used for DPU BMC root login, must be the same for
all DPUs
export BMC_ROOT_PASSWORD=

## Serial number of DPUs. If you have more than 2 DPUs, you will
need to parameterize the system accordingly and expose
## additional variables.
## All serial numbers must be in lowercase.
export DPU1_SERIAL=

export DPU2_SERIAL=
```

Modify the variables in `manifests/00-env-vars/envvars.env` to fit your environment, then source the file:

```
source manifests/00-env-vars/envvars.env
```

1. DPF Operator Installation

Create storage required by the DPF Operator

A number of environment variables must be set before running this command.

```
kubectl create ns dpf-operator-system
cat manifests/01-dpf-operator-installation/*.yaml | envsubst |
kubectl apply -f -
```

This deploys the following objects:

PersistentVolume and PersistentVolumeClaim for the provisioning controller

```
---
apiVersion: v1
kind: PersistentVolume
metadata:
  name: bfb-pv
spec:
  capacity:
    storage: 10Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteMany
  nfs:
    path: /mnt/dpf_share/bfb
    server: $NFS_SERVER_IP
  persistentVolumeReclaimPolicy: Delete
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: bfb-pvc
  namespace: dpf-operator-system
spec:
  accessModes:
  - ReadWriteMany
  resources:
    requests:
      storage: 10Gi
  volumeMode: Filesystem
  storageClassName: ""
```

Create DPU BMC shared password secret

In Zero Trust mode, provisioning DPUs requires authentication with Redfish. In order to do that, you must set the same root password to access the BMC for all DPUs DPF is going

to manage.

For more information on how to set the BMC root password refer to BlueField DPU Administrator Quick Start Guide

The password is provided to DPF by creating the following secret:

```
kubectl create secret generic -n dpf-operator-system bmc-shared-
password --from-literal=password=$BMC_ROOT_PASSWORD
```

Additional Dependencies

Before deploying the DPF Operator, ensure that Helm is properly configured according to the Helm prerequisites.

> ⚠ **Warning**
>
> This is a critical prerequisite step that must be completed for the DPF Operator to function properly.

Deploy the DPF Operator

A number of environment variables must be set before running this command.

HTTP Registry (default)

If the $REGISTRY is an HTTP Registry (default value) use this command:

```
helm repo add --force-update dpf-repository ${REGISTRY}
helm repo update
helm upgrade --install -n dpf-operator-system dpf-operator dpf-
repository/dpf-operator --version=$TAG
```

OCI Registry

For development purposes, if the $REGISTRY is an OCI Registry use this command:

```
helm upgrade --install -n dpf-operator-system dpf-operator
$REGISTRY/dpf-operator --version=$TAG
```

Verification

These verification commands may need to be run multiple times to ensure the condition is met.

Verify the DPF Operator installation with:

```
## Ensure the DPF Operator deployment is available.
kubectl rollout status deployment --namespace dpf-operator-system
dpf-operator-controller-manager
## Ensure all pods in the DPF Operator system are ready.
kubectl wait --for=condition=ready --namespace dpf-operator-system
pods --all
```

2. DPF System Installation

This section involves creating the DPF system components and some basic infrastructure required for a functioning DPF-enabled cluster.

Deploy the DPF System components

A number of environment variables must be set before running this command.

```
kubectl create ns dpu-cplane-tenant1
cat manifests/02-dpf-system-installation/*.yaml | envsubst |
kubectl apply -f -
```

This will create the following objects:

DPFOperatorConfig to install the DPF System components

```
---
apiVersion: operator.dpu.nvidia.com/v1alpha1
kind: DPFOperatorConfig
metadata:
  name: dpfoperatorconfig
  namespace: dpf-operator-system
spec:
  dpuDetector:
    disable: true
  provisioningController:
    bfbPVCName: "bfb-pvc"
    dmsTimeout: 900
    installInterface:
      installViaRedfish:
        # Set this to the IP of one of your control plane nodes +
8080 port
        bfbRegistryAddress: "$TARGETCLUSTER_API_SERVER_HOST:8080"
        skipDPUNodeDiscovery: false
  kamajiClusterManager:
    disable: false
```

DPUCluster to serve as Kubernetes control plane for DPU nodes

```
---
apiVersion: provisioning.dpu.nvidia.com/v1alpha1
kind: DPUCluster
metadata:
  name: dpu-cplane-tenant1
  namespace: dpu-cplane-tenant1
spec:
  type: kamaji
  maxNodes: 10
  clusterEndpoint:
    # deploy keepalived instances on the nodes that match the
given nodeSelector.
    keepalived:
      # interface on which keepalived will listen. Should be the oob
interface of the control plane node.
      interface: $DPUCLUSTER_INTERFACE
      # Virtual IP reserved for the DPU Cluster load balancer.
Must not be allocatable by DHCP.
      vip: $DPUCLUSTER_VIP
      # virtualRouterID must be in range [1,255], make sure the
given virtualRouterID does not duplicate with any existing
keepalived process running on the host
      virtualRouterID: 126
      nodeSelector:
        node-role.kubernetes.io/control-plane: ""
```

DPUDiscovery to discover DPUDevices or DPUNodes

```
---
apiVersion: provisioning.dpu.nvidia.com/v1alpha1
kind: DPUDiscovery
metadata:
  name: dpu-discovery
  namespace: dpf-operator-system
spec:
  ipRangeSpec:
    ipRange:
      startIP: $IP_RANGE_START
      endIP: $IP_RANGE_END
```

Verification

These verification commands may need to be run multiple times to ensure the condition is met.

Verify the DPF System with:

```
## Ensure the provisioning and DPUService controller manager
deployments are available.
kubectl rollout status deployment --namespace dpf-operator-system
dpf-provisioning-controller-manager dpuservice-controller-manager
## Ensure all other deployments in the DPF Operator system are
Available.
kubectl rollout status deployment --namespace dpf-operator-system
## Ensure bfb registry daemonset is available
kubectl rollout status daemonset --namespace dpf-operator-system
bfb-registry
## Ensure the DPUCluster is ready for nodes to join.
kubectl wait --for=condition=ready --namespace dpu-cplane-tenant1
dpucluster --all
```

3. DPU Provisioning and Service Installation

In this section, you'll provision your DPUs and deploy the required services. You'll need to create a `DPUDeployment` object that defines which `DPUServices` should be installed on each selected DPU. This provides a flexible way to specify and manage the services that run on your DPUs.

> If you want to learn more about `DPUDeployments`, check the DPUDeployment documentation.

This guide includes examples for both SNAP Block (NVMe) and SNAP VirtioFS Storage. Please refer to the relevant sections below and follow the instructions to deploy the desired storage type.

SNAP Block (NVMe)

Create the DPUDeployment, DPUServiceConfig, DPUServiceTemplate and other necessary objects

> ⚠ **Warning**
>
> In case more than 1 DPU exists per node, the relevant selector should be applied in the DPUDeployment to select the appropriate DPU. See DPUDeployment - DPUs Configuration to understand more about the selectors.

A number of environment variables must be set before running this command.

```
cat manifests/03.1-dpudeployment-installation-nvme/*.yaml |
envsubst | kubectl apply -f -
```

This will deploy the following objects:

BFB to download Bluefield Bitstream to a shared volume

```
---
apiVersion: provisioning.dpu.nvidia.com/v1alpha1
kind: BFB
metadata:
  name: bf-bundle-$TAG
  namespace: dpf-operator-system
spec:
  url: $BFB_URL
```

HBN DPUFlavor to correctly configure the DPUs on provisioning

```
---
apiVersion: provisioning.dpu.nvidia.com/v1alpha1
kind: DPUFlavor
metadata:
  name: hbn-snap-nvme-$TAG
  namespace: dpf-operator-system
spec:
  bfcfgParameters:
  - UPDATE_ATF_UEFI=yes
  - UPDATE_DPU_OS=yes
  - WITH_NIC_FW_UPDATE=yes
  configFiles:
  - operation: override
    path: /etc/mellanox/mlnx-bf.conf
    permissions: "0644"
    raw: |
      ALLOW_SHARED_RQ="no"
      IPSEC_FULL_OFFLOAD="no"
      ENABLE_ESWITCH_MULTIPORT="yes"
      RDMA_SET_NETNS_EXCLUSIVE="no"
  - operation: override
    path: /etc/mellanox/mlnx-ovs.conf
    permissions: "0644"
    raw: |
      CREATE_OVS_BRIDGES="no"
      OVS_DOCA="yes"
  - operation: override
    path: /etc/mellanox/mlnx-sf.conf
    permissions: "0644"
    raw: ""
  grub:
    kernelParameters:
    - console=hvc0
    - console=ttyAMA0
    - earlycon=pl011,0x13010000
```

```
    - fixrttc
    - net.ifnames=0
    - biosdevname=0
    - iommu.passthrough=1
    - cgroup_no_v1=net_prio,net_cls
    - hugepagesz=2048kB
    - hugepages=5120
  nvconfig:
  - device: '*'
    parameters:
    - PF_BAR2_ENABLE=0
    - PER_PF_NUM_SF=1
    - PF_TOTAL_SF=20
    - PF_SF_BAR_SIZE=10
    - NUM_PF_MSIX_VALID=0
    - PF_NUM_PF_MSIX_VALID=1
    - PF_NUM_PF_MSIX=228
    - INTERNAL_CPU_MODEL=1
    - INTERNAL_CPU_OFFLOAD_ENGINE=0
    - SRIOV_EN=1
    - NUM_OF_VFS=46
    - LAG_RESOURCE_ALLOCATION=1
    - NVME_EMULATION_ENABLE=1
    - NVME_EMULATION_NUM_PF=1
    - PCI_SWITCH_EMULATION_ENABLE=1
    - PCI_SWITCH_EMULATION_NUM_PORT=32
    - LINK_TYPE_P1=ETH
    - LINK_TYPE_P2=ETH
  ovs:
    rawConfigScript: |
      _ovs-vsctl() {
        ovs-vsctl --no-wait --timeout 15 "$@"
      }

      _ovs-vsctl set Open_vSwitch . other_config:doca-init=true
```

```
      _ovs-vsctl set Open_vSwitch . other_config:dpdk-max-
memzones=50000
      _ovs-vsctl set Open_vSwitch . other_config:hw-offload=true
      _ovs-vsctl set Open_vSwitch . other_config:pmd-quiet-
idle=true
      _ovs-vsctl set Open_vSwitch . other_config:max-idle=20000
      _ovs-vsctl set Open_vSwitch . other_config:max-
revalidator=5000
      _ovs-vsctl --if-exists del-br ovsbr1
      _ovs-vsctl --if-exists del-br ovsbr2
      _ovs-vsctl --may-exist add-br br-sfc
      _ovs-vsctl set bridge br-sfc datapath_type=netdev
      _ovs-vsctl set bridge br-sfc fail_mode=secure
      _ovs-vsctl --may-exist add-port br-sfc p0
      _ovs-vsctl set Interface p0 type=dpdk
      _ovs-vsctl set Interface p0 mtu_request=9216
      _ovs-vsctl set Port p0 external_ids:dpf-type=physical
      _ovs-vsctl --may-exist add-port br-sfc p1
      _ovs-vsctl set Interface p1 type=dpdk
      _ovs-vsctl set Interface p1 mtu_request=9216
      _ovs-vsctl set Port p1 external_ids:dpf-type=physical
      _ovs-vsctl --may-exist add-br br-hbn
      _ovs-vsctl set bridge br-hbn datapath_type=netdev
      _ovs-vsctl set bridge br-hbn fail_mode=secure
```

DPUDeployment to provision DPUs on worker nodes

```yaml
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUDeployment
metadata:
  name: hbn-snap
  namespace: dpf-operator-system
spec:
  dpus:
    bfb: bf-bundle-$TAG
    flavor: hbn-snap-nvme-$TAG
    nodeEffect:
      noEffect: true
    dpuSets:
    - nameSuffix: "dpuset1"
      dpuAnnotations:
        storage.nvidia.com/preferred-dpu: "true"
      nodeSelector:
        matchLabels:
          feature.node.kubernetes.io/dpu-enabled: "true"
  services:
    doca-hbn:
      serviceTemplate: doca-hbn
      serviceConfiguration: doca-hbn
    snap-host-controller:
      serviceTemplate: snap-host-controller
      serviceConfiguration: snap-host-controller
    snap-node-driver:
      serviceTemplate: snap-node-driver
      serviceConfiguration: snap-node-driver
    doca-snap:
      serviceTemplate: doca-snap
      serviceConfiguration: doca-snap
    block-storage-dpu-plugin:
      serviceTemplate: block-storage-dpu-plugin
      serviceConfiguration: block-storage-dpu-plugin
```

```
    spdk-csi-controller:
      serviceTemplate: spdk-csi-controller
      serviceConfiguration: spdk-csi-controller
    spdk-csi-controller-dpu:
      serviceTemplate: spdk-csi-controller-dpu
      serviceConfiguration: spdk-csi-controller-dpu
  serviceChains:
    switches:
      - ports:
        - serviceInterface:
            matchLabels:
              interface: p0
        - service:
            name: doca-hbn
            interface: p0_if
      - ports:
        - serviceInterface:
            matchLabels:
              interface: p1
        - service:
            name: doca-hbn
            interface: p1_if
      - ports:
        - serviceInterface:
            matchLabels:
              interface: pf0hpf
        - service:
            name: doca-hbn
            interface: pf0hpf_if
      - ports:
        - serviceInterface:
            matchLabels:
              interface: pf1hpf
        - service:
            name: doca-hbn
            interface: pf1hpf_if
```

```
      - ports:
        - service:
            name: doca-snap
            interface: app_sf
            ipam:
              matchLabels:
                svc.dpu.nvidia.com/pool: storage-pool
        - service:
            name: doca-hbn
            interface: snap_if
```

DPUServiceConfig and DPUServiceTemplate to deploy HBN workloads to the DPUs

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceConfiguration
metadata:
  name: doca-hbn
  namespace: dpf-operator-system
spec:
  deploymentServiceName: "doca-hbn"
  serviceConfiguration:
    serviceDaemonSet:
      annotations:
        k8s.v1.cni.cncf.io/networks: |-
          [
          {"name": "iprequest", "interface": "ip_lo", "cni-args": {"poolNames": ["loopback"], "poolType": "cidrpool"}},
          {"name": "iprequest", "interface": "ip_pf0hpf", "cni-args": {"poolNames": ["pool1"], "poolType": "cidrpool", "allocateDefaultGateway": true}},
          {"name": "iprequest", "interface": "ip_pf1hpf", "cni-args": {"poolNames": ["pool2"], "poolType": "cidrpool", "allocateDefaultGateway": true}}
          ]
    helmChart:
      values:
        configuration:
          perDPUValuesYAML: |
            - hostnamePattern: "*"
              values:
                bgp_peer_group: hbn
                vrf1: RED
                vrf2: BLUE
                l3vni1: 100001
                l3vni2: 100002
            - hostnamePattern: "dpu-node-${DPU1_SERIAL}*"
              values:
                bgp_autonomous_system: 65101
            - hostnamePattern: "dpu-node-${DPU2_SERIAL}*"
```

```
            values:
              bgp_autonomous_system: 65201
        startupYAMLJ2: |
          - header:
              model: bluefield
              nvue-api-version: nvue_v1
              rev-id: 1.0
              version: HBN 3.0.0
          - set:
              evpn:
                enable: on
                route-advertise: {}
              bridge:
                domain:
                  br_default:
                    vlan:
                      '10':
                        vni:
                          '10': {}
              interface:
                lo:
                  ip:
                    address:
                      {{ ipaddresses.ip_lo.ip }}/32: {}
                  type: loopback
                p0_if,p1_if,pf0hpf_if,pf1hpf_if:
                  type: swp
                  link:
                    mtu: 9000
                pf0hpf_if:
                  ip:
                    address:
                      {{ ipaddresses.ip_pf0hpf.cidr }}: {}
                    vrf: {{ config.vrf1 }}
                pf1hpf_if:
                  ip:
```

```
            address:
              {{ ipaddresses.ip_pf1hpf.cidr }}: {}
            vrf: {{ config.vrf2 }}
          snap_if:
            bridge:
              domain:
                br_default:
                  access: 10
          vlan10:
            type: svi
            vlan: 10
        nve:
          vxlan:
            arp-nd-suppress: on
            enable: on
            source:
              address: {{ ipaddresses.ip_lo.ip }}
        router:
          bgp:
            enable: on
            graceful-restart:
              mode: full
        vrf:
          default:
            router:
              bgp:
                address-family:
                  ipv4-unicast:
                    enable: on
                    redistribute:
                      connected:
                        enable: on
                    multipaths:
                      ebgp: 16
                  l2vpn-evpn:
                    enable: on
```

```
                        autonomous-system: {{
config.bgp_autonomous_system }}
                        enable: on
                        neighbor:
                          p0_if:
                            peer-group: {{ config.bgp_peer_group
}}
                            type: unnumbered
                            address-family:
                              l2vpn-evpn:
                                enable: on
                                add-path-tx: off
                          p1_if:
                            peer-group: {{ config.bgp_peer_group
}}
                            type: unnumbered
                            address-family:
                              l2vpn-evpn:
                                enable: on
                                add-path-tx: off
                        path-selection:
                          multipath:
                            aspath-ignore: on
                        peer-group:
                          {{ config.bgp_peer_group }}:
                            address-family:
                              ipv4-unicast:
                                enable: on
                              l2vpn-evpn:
                                enable: on
                            remote-as: external
                        router-id: {{ ipaddresses.ip_lo.ip }}
                    {{ config.vrf1 }}:
                      evpn:
                        enable: on
                        vni:
```

```
                        {{ config.l3vni1 }}: {}
                    router:
                      bgp:
                        address-family:
                          ipv4-unicast:
                            enable: on
                            redistribute:
                              connected:
                                enable: on
                            route-export:
                              to-evpn:
                                enable: on
                        autonomous-system: {{
config.bgp_autonomous_system }}
                        enable: on
                  {{ config.vrf2 }}:
                    evpn:
                      enable: on
                      vni:
                        {{ config.l3vni2 }}: {}
                    router:
                      bgp:
                        address-family:
                          ipv4-unicast:
                            enable: on
                            redistribute:
                              connected:
                                enable: on
                            route-export:
                              to-evpn:
                                enable: on
                        autonomous-system: {{
config.bgp_autonomous_system }}
                        enable: on

  interfaces:
```

```
  - name: p0_if
    network: mybrhbn
  - name: p1_if
    network: mybrhbn
  - name: pf0hpf_if
    network: mybrhbn
  - name: pf1hpf_if
    network: mybrhbn
  - name: snap_if
    network: mybrhbn
```

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceTemplate
metadata:
  name: doca-hbn
  namespace: dpf-operator-system
spec:
  deploymentServiceName: "doca-hbn"
  helmChart:
    source:
      repoURL: $HELM_REGISTRY_REPO_URL
      version: 1.0.5
      chart: doca-hbn
    values:
      image:
        repository: $HBN_NGC_IMAGE_URL
        tag: 3.2.1-doca3.2.1
      resources:
        memory: 6Gi
        nvidia.com/bf_sf: 5
```

DPUServiceInterfaces for physical ports on the DPU

```yaml
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceInterface
metadata:
  name: p0
  namespace: dpf-operator-system
spec:
  template:
    spec:
      template:
        metadata:
          labels:
            interface: "p0"
        spec:
          interfaceType: physical
          physical:
            interfaceName: p0
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceInterface
metadata:
  name: p1
  namespace: dpf-operator-system
spec:
  template:
    spec:
      template:
        metadata:
          labels:
            interface: "p1"
        spec:
          interfaceType: physical
          physical:
            interfaceName: p1
---
```

```yaml
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceInterface
metadata:
  name: pf0hpf
  namespace: dpf-operator-system
spec:
  template:
    spec:
      template:
        metadata:
          labels:
            interface: "pf0hpf"
        spec:
          interfaceType: pf
          pf:
            pfID: 0
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceInterface
metadata:
  name: pf1hpf
  namespace: dpf-operator-system
spec:
  template:
    spec:
      template:
        metadata:
          labels:
            interface: "pf1hpf"
        spec:
          interfaceType: pf
          pf:
            pfID: 1
```

DPUServiceIPAM to set up IP Address Management on the DPUCluster

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceIPAM
metadata:
  name: pool1
  namespace: dpf-operator-system
spec:
  ipv4Network:
    network: "10.0.121.0/24"
    gatewayIndex: 2
    prefixSize: 29
    # These preallocations are not necessary. We specify them so
that the validation commands are straightforward.
    allocations:
      dpu-node-${DPU1_SERIAL}-${DPU1_SERIAL}: 10.0.121.0/29
      dpu-node-${DPU2_SERIAL}-${DPU2_SERIAL}: 10.0.121.8/29
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceIPAM
metadata:
  name: pool2
  namespace: dpf-operator-system
spec:
  ipv4Network:
    network: "10.0.122.0/24"
    gatewayIndex: 2
    prefixSize: 29
```

DPUServiceIPAM for the loopback interface in HBN

```yaml
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceIPAM
metadata:
  name: loopback
  namespace: dpf-operator-system
spec:
  ipv4Network:
    network: "11.0.0.0/24"
    prefixSize: 32
```

DPUServiceIPAM for storage network

```yaml
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceIPAM
metadata:
  name: storage-pool
  namespace: dpf-operator-system
spec:
  metadata:
    labels:
      svc.dpu.nvidia.com/pool: storage-pool
  ipv4Subnet:
    subnet: "10.0.124.0/24"
    gateway: "10.0.124.1"
    perNodeIPCount: 8
```

DPUServiceConfiguration and DPUServiceTemplate for DOCA SNAP

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceConfiguration
metadata:
  name: doca-snap
  namespace: dpf-operator-system
spec:
  deploymentServiceName: doca-snap
  serviceConfiguration:
    helmChart:
      values:
        dpu:
          docaSnap:
            enabled: true
            image:
              repository: $SNAP_NGC_IMAGE_URL
              tag: 1.5.0-doca3.2.0
            snapRpcInitConf: |
              nvme_subsystem_create --nqn nqn.2022-
10.io.nvda.nvme:0
  interfaces:
  - name: app_sf
    network: mybrsfc
```

```yaml
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceTemplate
metadata:
  name: doca-snap
  namespace: dpf-operator-system
spec:
  deploymentServiceName: doca-snap
  helmChart:
    source:
      repoURL: $REGISTRY
      version: $TAG
      chart: dpf-storage
    values:
      serviceDaemonSet:
        resources:
          memory: "2Gi"
          hugepages-2Mi: "4Gi"
          cpu: "8"
          nvidia.com/bf_sf: 1
  resourceRequirements:
    memory: "2Gi"
    hugepages-2Mi: "4Gi"
    cpu: "8"
    nvidia.com/bf_sf: 1
```

DPUServiceConfiguration and DPUServiceTemplate for Block Storage DPU Plugin

```yaml
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceConfiguration
metadata:
  name: block-storage-dpu-plugin
  namespace: dpf-operator-system
spec:
  deploymentServiceName: block-storage-dpu-plugin
  serviceConfiguration:
    helmChart:
      values:
        dpu:
          blockStorageVendorDpuPlugin:
            enabled: true
```

```yaml
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceTemplate
metadata:
  name: block-storage-dpu-plugin
  namespace: dpf-operator-system
spec:
  deploymentServiceName: block-storage-dpu-plugin
  helmChart:
    source:
      repoURL: $REGISTRY
      version: $TAG
      chart: dpf-storage
```

DPUServiceConfiguration and DPUServiceTemplate for SNAP Host Controller

```yaml
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceConfiguration
metadata:
  name: snap-host-controller
  namespace: dpf-operator-system
spec:
  deploymentServiceName: snap-host-controller
  upgradePolicy:
    applyNodeEffect: false
  serviceConfiguration:
    deployInCluster: true
    helmChart:
      values:
        host:
          snapHostController:
            enabled: true
            config:
              targetNamespace: dpf-operator-system
            affinity:
              nodeAffinity:
                requiredDuringSchedulingIgnoredDuringExecution:
                  nodeSelectorTerms:
                  - matchExpressions:
                      - key: "node-role.kubernetes.io/master"
                        operator: Exists
                  - matchExpressions:
                      - key: "node-role.kubernetes.io/control-plane"
                        operator: Exists
```

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceTemplate
metadata:
  name: snap-host-controller
  namespace: dpf-operator-system
spec:
  deploymentServiceName: snap-host-controller
  helmChart:
    source:
      repoURL: $REGISTRY
      version: $TAG
      chart: dpf-storage
```

DPUServiceConfiguration and DPUServiceTemplate for SNAP Node Driver

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceConfiguration
metadata:
  name: snap-node-driver
  namespace: dpf-operator-system
spec:
  deploymentServiceName: snap-node-driver
  serviceConfiguration:
    helmChart:
      values:
        dpu:
          deployCrds: true
          snapNodeDriver:
            enabled: true
```

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceTemplate
metadata:
  name: snap-node-driver
  namespace: dpf-operator-system
spec:
  deploymentServiceName: snap-node-driver
  helmChart:
    source:
      repoURL: $REGISTRY
      version: $TAG
      chart: dpf-storage
```

DPUServiceConfiguration and DPUServiceTemplate for SPDK CSI Controller

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceConfiguration
metadata:
  name: spdk-csi-controller
  namespace: dpf-operator-system
spec:
  deploymentServiceName: spdk-csi-controller
  upgradePolicy:
    applyNodeEffect: false
  serviceConfiguration:
    deployInCluster: true
    helmChart:
      values:
        host:
          enabled: true
          config:
            targets:
              nodes:
                # name of the target
                - name: spdk-target
                  # management address
                  rpcURL: http://10.0.110.25:8000
                  # type of the target, e.g. nvme-tcp, nvme-rdma
                  targetType: nvme-rdma
                  # target service IP
                  targetAddr: 10.0.124.1
            # required parameter, name of the secret that
contains connection
            # details to access the DPU cluster.
            # this secret should be created by the
DPUServiceCredentialRequest API.
```

```
            dpuClusterSecret: spdk-csi-controller-dpu-cluster-
credentials
```

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceTemplate
metadata:
  name: spdk-csi-controller
  namespace: dpf-operator-system
spec:
  deploymentServiceName: spdk-csi-controller
  helmChart:
    source:
      repoURL: oci://ghcr.io/mellanox/dpf-storage-vendors-charts
      version: v0.3.0
      chart: spdk-csi-controller
```

DPUServiceConfiguration and DPUServiceTemplate for SPDK CSI Controller on DPU

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceConfiguration
metadata:
  name: spdk-csi-controller-dpu
  namespace: dpf-operator-system
spec:
  deploymentServiceName: spdk-csi-controller-dpu
  upgradePolicy:
    applyNodeEffect: false
  serviceConfiguration:
    helmChart:
      values:
        dpu:
          enabled: true
          storageClass:
            # the name of the storage class that will be created for
spdk-csi,
            # this StorageClass name should be used in the
StorageVendor settings
            name: spdkcsi-sc
            # name of the secret that contains credentials for the
remote SPDK target,
            # content of the secret is injected during
CreateVolume request
            secretName: spdkcsi-secret
            # namespace of the secret with credentials for the
remote SPDK target
            secretNamespace: dpf-operator-system
          rbacRoles:
            spdkCsiController:
              # the name of the service account for spdk-csi-
controller
              # this value must be aligned with the value from the
DPUServiceCredentialRequest
```

```
              serviceAccount: spdk-csi-controller-sa
```

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceTemplate
metadata:
  name: spdk-csi-controller-dpu
  namespace: dpf-operator-system
spec:
  deploymentServiceName: spdk-csi-controller-dpu
  helmChart:
    source:
      repoURL: oci://ghcr.io/mellanox/dpf-storage-vendors-charts
      version: v0.3.0
      chart: spdk-csi-controller
```

DPUServiceCredentialRequest for SPDK CSI Controller

```yaml
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceCredentialRequest
metadata:
  name: spdk-csi-controller-credentials
  namespace: dpf-operator-system
spec:
  duration: 10m
  serviceAccount:
    name: spdk-csi-controller-sa
    namespace: dpf-operator-system
  targetCluster:
    name: dpu-cplane-tenant1
    namespace: dpu-cplane-tenant1
  type: tokenFile
  secret:
    name: spdk-csi-controller-dpu-cluster-credentials
    namespace: dpf-operator-system
```

Secret for SPDK CSI

```
---
apiVersion: v1
kind: Secret
metadata:
  name: spdkcsi-secret
  namespace: dpf-operator-system
  labels:
    # this label enables replication of the secret from the host to
the dpu cluster
    dpu.nvidia.com/image-pull-secret: ""
stringData:
  # name field in the "rpcTokens" list should match name of the
  # spdk target from
DPUService.helmChart.values.host.config.targets.nodes
  secret.json: |-
    {
      "rpcTokens": [
        {
          "name": "spdk-target",
          "username": "exampleuser",
          "password": "examplepassword"
        }
      ]
    }
```

Verification

These verification commands may need to be run multiple times to ensure the condition is met.

Note that the DPUService name will have a random suffix. For example, `doca-hbn-l2xsl`.

Verify the DPU and Service installation with:

```
## Ensure the DPUServices are created and have been reconciled.
kubectl wait --for=condition=ApplicationsReconciled --namespace
dpf-operator-system dpuservices -l svc.dpu.nvidia.com/owned-by-
dpudeployment=dpf-operator-system_hbn-snap
## Ensure the DPUServiceIPAMs have been reconciled
kubectl wait --for=condition=DPUIPAMObjectReconciled --namespace
dpf-operator-system dpuserviceipam --all
## Ensure the DPUServiceInterfaces have been reconciled
kubectl wait --for=condition=ServiceInterfaceSetReconciled --
namespace dpf-operator-system dpuserviceinterface --all
## Ensure the DPUServiceChains have been reconciled
kubectl wait --for=condition=ServiceChainSetReconciled --namespace
dpf-operator-system dpuservicechain --all
## Ensure the DPUs have the condition Initialized (this may take
time)
kubectl wait --for=condition=Initialized --namespace dpf-operator-
system dpu --all
```

or with `dpfctl`:

```
$ kubectl -n dpf-operator-system exec deploy/dpf-operator-
controller-manager -- /dpfctl describe dpudeployments
```

Apply Storage Configuration

A number of environment variables must be set before running this command.

```
cat manifests/04.1-storage-configuration-nvme/*.yaml | envsubst |
kubectl apply -f -
```

This will create the following objects:

DPUStorageVendor for SPDK CSI

```
---
apiVersion: storage.dpu.nvidia.com/v1alpha1
kind: DPUStorageVendor
metadata:
  name: spdk-csi
  namespace: dpf-operator-system
spec:
  storageClassName: spdkcsi-sc
  pluginName: nvidia-block
```

DPUStoragePolicy for block storage

```
---
apiVersion: storage.dpu.nvidia.com/v1alpha1
kind: DPUStoragePolicy
metadata:
  name: policy-block
  namespace: dpf-operator-system
spec:
  dpuStorageVendors:
    - spdk-csi
  selectionAlgorithm: "NumberVolumes"
  parameters: {}
```

Check that the objects are ready:

```
kubectl wait --for=condition=Ready --namespace dpf-operator-system
dpustoragevendors --all
kubectl wait --for=condition=Ready --namespace dpf-operator-system
dpustoragepolicies --all
```

Create static volume attachments

A number of environment variables must be set before running this command.

```
cat manifests/05.1-storage-test-nvme/*.yaml | envsubst | kubectl
apply -f -
```

This will create the following objects:

DPUVolumes for block storage

```
---
apiVersion: storage.dpu.nvidia.com/v1alpha1
kind: DPUVolume
metadata:
  name: test-volume-static-pf-${DPU1_SERIAL}
  namespace: dpf-operator-system
spec:
  dpuStoragePolicyName: policy-block
  resources:
    requests:
      storage: 30Gi
  accessModes:
  - ReadWriteOnce
  volumeMode: Block
---
apiVersion: storage.dpu.nvidia.com/v1alpha1
kind: DPUVolume
metadata:
  name: test-volume-static-pf-${DPU2_SERIAL}
  namespace: dpf-operator-system
spec:
  dpuStoragePolicyName: policy-block
  resources:
    requests:
      storage: 30Gi
  accessModes:
  - ReadWriteOnce
  volumeMode: Block
```

DPUVolumeAttachments for block storage with static PFs

```
---
apiVersion: storage.dpu.nvidia.com/v1alpha1
kind: DPUVolumeAttachment
metadata:
  name: test-volume-attachment-static-pf-${DPU1_SERIAL}
  namespace: dpf-operator-system
spec:
  dpuNodeName: dpu-node-${DPU1_SERIAL}
  dpuVolumeName: test-volume-static-pf-${DPU1_SERIAL}
  functionType: pf
  hotplugFunction: false
---
apiVersion: storage.dpu.nvidia.com/v1alpha1
kind: DPUVolumeAttachment
metadata:
  name: test-volume-attachment-static-pf-${DPU2_SERIAL}
  namespace: dpf-operator-system
spec:
  dpuNodeName: dpu-node-${DPU2_SERIAL}
  dpuVolumeName: test-volume-static-pf-${DPU2_SERIAL}
  functionType: pf
  hotplugFunction: false
```

At this stage, volumes can be created, but they will not be attached to the DPUs yet, because the DPUs are not ready.

Verify that the DPUVolume resources are in the Ready state:

```
kubectl wait --for=condition=Ready --namespace dpf-operator-system
dpuvolumes --all
```

> **Warning**
>
> Be sure to create DPUVolumeAttachments that are using static PFs *before* rebooting the hosts. If you reboot the hosts without creating these attachments, the hardware initialization process on the host can be significantly delayed, due to timeouts caused by partially initialized emulated NVMe controllers. Creating the attachments ahead of time ensures that SNAP services can complete initialization of the emulated NVMe controllers right after the DPUs become ready.

Making the DPUs Ready

In order to make the DPUs ready, we will need to manually power cycle the hosts. This operation should be done in the most graceful manner by gracefully shutting down the Host and DPU, powering off the server and then powering it on to avoid corruption. This should happen when the object gives us the signal. The described flow can be automated by the admin depending on the infrastructure. The following verification command may need to be run multiple times to ensure the condition is met.

```
## Ensure the DPUs are in the Rebooting phase and condition
Rebooted is false with WaitingForManualPowerCycleOrReboot reason
kubectl wait --for=jsonpath='{.status.conditions[?
(@.type=="Rebooted")].reason}'=WaitingForManualPowerCycleOrReboot --
namespace dpf-operator-system dpu --all
```

or with `dpfctl`:

```
$ kubectl -n dpf-operator-system exec deploy/dpf-operator-
controller-manager -- /dpfctl describe dpudeployments
```

At this point, we have to power cycle the hosts.

> **Note**
>
> For the SNAP Block (NVMe) scenario, you do not need to wait for the host to fully boot after a power cycle before removing the annotation below. On some server platforms, device initialization may be waiting for a long time until the SNAP service on the DPU becomes ready, so it is recommended to remove the annotation immediately after initiating the host power cycle.

```
kubectl annotate dpunodes -n dpf-operator-system --all
provisioning.dpu.nvidia.com/dpunode-external-reboot-required-
```

After this is done, we should expect that all DPUs become Ready:

```
kubectl wait --for="jsonpath={.status.phase}=Ready" --namespace dpf-operator-
system dpu --all
```

or with dpfctl:

```
$ kubectl -n dpf-operator-system exec deploy/dpf-operator-
controller-manager -- /dpfctl describe dpudeployments
```

DPUVolumeAttachments that were created in the previous step should become Ready after some time:

```
kubectl wait --for=condition=Ready --namespace dpf-operator-system
dpuvolumeattachments --all
```

Test Block Storage

Test Block Storage with Static PFs

A number of environment variables must be set before running this command.

First we check DPUVolumes that were exposed through the static storage PFs.

On the host that is used to run this guide, get the PCI address of the DPUVolumeAttachment:

```
kubectl wait --for=condition=Ready dpuvolumeattachments/test-volume-attachment-static-pf-${DPU1_SERIAL} -n dpf-operator-system
kubectl wait --for=condition=Ready dpuvolumeattachments/test-volume-attachment-static-pf-${DPU2_SERIAL} -n dpf-operator-system
pci_address_1=$(printf "0000:%s" "$(kubectl get -n dpf-operator-system dpuvolumeattachments.storage.dpu.nvidia.com test-volume-attachment-static-pf-${DPU1_SERIAL} -o jsonpath='{.status.dpu.pciAddress}')")
pci_address_2=$(printf "0000:%s" "$(kubectl get -n dpf-operator-system dpuvolumeattachments.storage.dpu.nvidia.com test-volume-attachment-static-pf-${DPU2_SERIAL} -o jsonpath='{.status.dpu.pciAddress}')")
echo "Worker $DPU1_SERIAL NMMe PCI address: $pci_address_1"
echo "Worker $DPU2_SERIAL NMMe PCI address: $pci_address_2"
```

Connect to the worker nodes with DPUs and set pci address variable to point to the PCI address retrieved in the previous step.

```
pci_address=<set to the PCI address that you retrieved in the previous step>
```

Check the current driver for the device

```
ls -lah /sys/bus/pci/devices/"$pci_address"/driver
```

If the device has no driver, you may need to manually bind it.

```
echo nvme > /sys/bus/pci/devices/$pci_address/driver_override
echo $pci_address > /sys/bus/pci/drivers/nvme/bind
```

Find the block device name

```
block_dev_name=$(basename
/sys/bus/pci/drivers/nvme/"$pci_address"/nvme/*/nvme*)
echo "Block device name: $block_dev_name"
```

Format the device to the filesystem or use it as a raw block device.

Example how to format the device with ext4 filesystem and perform I/O operations:

```
mkfs.ext4 /dev/"$block_dev_name"
mkdir -p /tmp/test-volume-static-pf
mount /dev/"$block_dev_name" /tmp/test-volume-static-pf

dd if=/dev/urandom of=/tmp/test-volume-static-pf/test.txt bs=1M
count=1000 status=progress
```

```
umount /tmp/test-volume-static-pf
```

Test Block Storage with Hot-plug PFs

A number of environment variables must be set before running this command.

Create DPUVolume and DPUVolumeAttachment for hot-plug storage PFs :

```
cat manifests/05.1-storage-test-nvme/hotplug/*.yaml | envsubst |
kubectl apply -f -
```

This will create the following objects:

DPUVolume for hot-plug storage PFs

```
---
apiVersion: storage.dpu.nvidia.com/v1alpha1
kind: DPUVolume
metadata:
  name: test-volume-hotplug-pf-${DPU1_SERIAL}
  namespace: dpf-operator-system
spec:
  dpuStoragePolicyName: policy-block
  resources:
    requests:
      storage: 30Gi
  accessModes:
  - ReadWriteOnce
  volumeMode: Block
---
apiVersion: storage.dpu.nvidia.com/v1alpha1
kind: DPUVolume
metadata:
  name: test-volume-hotplug-pf-${DPU2_SERIAL}
  namespace: dpf-operator-system
spec:
  dpuStoragePolicyName: policy-block
  resources:
    requests:
      storage: 30Gi
  accessModes:
  - ReadWriteOnce
  volumeMode: Block
```

DPUVolumeAttachment for hot-plug storage PFs

```
---
apiVersion: storage.dpu.nvidia.com/v1alpha1
kind: DPUVolumeAttachment
metadata:
  name: test-volume-attachment-hotplug-pf-${DPU1_SERIAL}
  namespace: dpf-operator-system
spec:
  dpuNodeName: dpu-node-${DPU1_SERIAL}
  dpuVolumeName: test-volume-hotplug-pf-${DPU1_SERIAL}
  functionType: pf
  hotplugFunction: true
---
apiVersion: storage.dpu.nvidia.com/v1alpha1
kind: DPUVolumeAttachment
metadata:
  name: test-volume-attachment-hotplug-pf-${DPU2_SERIAL}
  namespace: dpf-operator-system
spec:
  dpuNodeName: dpu-node-${DPU2_SERIAL}
  dpuVolumeName: test-volume-hotplug-pf-${DPU2_SERIAL}
  functionType: pf
  hotplugFunction: true
```

On the host that is used to run this guide, get the PCI address of the DPUVolumeAttachment:

```
kubectl wait --for=condition=Ready dpuvolumeattachments/test-volume-attachment-hotplug-pf-${DPU1_SERIAL} -n dpf-operator-system
kubectl wait --for=condition=Ready dpuvolumeattachments/test-volume-attachment-hotplug-pf-${DPU2_SERIAL} -n dpf-operator-system
pci_address_1=$(printf "0000:%s" "$(kubectl get -n dpf-operator-system dpuvolumeattachments.storage.dpu.nvidia.com test-volume-attachment-hotplug-pf-${DPU1_SERIAL} -o jsonpath='{.status.dpu.pciAddress}')")
pci_address_2=$(printf "0000:%s" "$(kubectl get -n dpf-operator-system dpuvolumeattachments.storage.dpu.nvidia.com test-volume-attachment-hotplug-pf-${DPU2_SERIAL} -o jsonpath='{.status.dpu.pciAddress}')")
echo "Worker $DPU1_SERIAL NMMe PCI address: $pci_address_1"
echo "Worker $DPU2_SERIAL NMMe PCI address: $pci_address_2"
```

After volumes are successfully attached repeat the steps from the Test Block Storage with Static PFs section.

SNAP VirtioFS

Create the DPUDeployment, DPUServiceConfig, DPUServiceTemplate and other necessary objects

> **Warning**
>
> In case more than 1 DPU exists per node, the relevant selector should be applied in the DPUDeployment to select the appropriate DPU. See DPUDeployment - DPUs Configuration to understand more about the selectors.

A number of environment variables must be set before running this command.

```
cat manifests/03.2-dpudeployment-installation-virtiofs/*.yaml |
envsubst | kubectl apply -f -
```

This will deploy the following objects:

BFB to download Bluefield Bitstream to a shared volume

```
---
apiVersion: provisioning.dpu.nvidia.com/v1alpha1
kind: BFB
metadata:
  name: bf-bundle-$TAG
  namespace: dpf-operator-system
spec:
  url: $BFB_URL
```

DPUFlavor with VirtioFS Storage config

```
---
apiVersion: provisioning.dpu.nvidia.com/v1alpha1
kind: DPUFlavor
metadata:
  name: hbn-snap-virtiofs-$TAG
  namespace: dpf-operator-system
spec:
  bfcfgParameters:
  - UPDATE_ATF_UEFI=yes
  - UPDATE_DPU_OS=yes
  - WITH_NIC_FW_UPDATE=yes
  configFiles:
  - operation: override
    path: /etc/mellanox/mlnx-bf.conf
    permissions: "0644"
    raw: |
      ALLOW_SHARED_RQ="no"
      IPSEC_FULL_OFFLOAD="no"
      ENABLE_ESWITCH_MULTIPORT="yes"
      RDMA_SET_NETNS_EXCLUSIVE="no"
  - operation: override
    path: /etc/mellanox/mlnx-ovs.conf
    permissions: "0644"
    raw: |
      CREATE_OVS_BRIDGES="no"
      OVS_DOCA="yes"
  - operation: override
    path: /etc/mellanox/mlnx-sf.conf
    permissions: "0644"
    raw: ""
  grub:
    kernelParameters:
    - console=hvc0
    - console=ttyAMA0
    - earlycon=pl011,0x13010000
```

```
    - fixrttc
    - net.ifnames=0
    - biosdevname=0
    - iommu.passthrough=1
    - cgroup_no_v1=net_prio,net_cls
    - hugepagesz=2048kB
    - hugepages=5120
  nvconfig:
  - device: '*'
    parameters:
    - PF_BAR2_ENABLE=0
    - PER_PF_NUM_SF=1
    - PF_TOTAL_SF=20
    - PF_SF_BAR_SIZE=10
    - NUM_PF_MSIX_VALID=0
    - PF_NUM_PF_MSIX_VALID=1
    - PF_NUM_PF_MSIX=228
    - INTERNAL_CPU_MODEL=1
    - INTERNAL_CPU_OFFLOAD_ENGINE=0
    - SRIOV_EN=1
    - NUM_OF_VFS=46
    - LAG_RESOURCE_ALLOCATION=1
    - PCI_SWITCH_EMULATION_ENABLE=1
    - PCI_SWITCH_EMULATION_NUM_PORT=32
    - VIRTIO_FS_EMULATION_ENABLE=1
    - VIRTIO_FS_EMULATION_NUM_PF=0
    - LINK_TYPE_P1=ETH
    - LINK_TYPE_P2=ETH
  ovs:
    rawConfigScript: |
      _ovs-vsctl() {
        ovs-vsctl --no-wait --timeout 15 "$@"
      }

      _ovs-vsctl set Open_vSwitch . other_config:doca-init=true
```

```
      _ovs-vsctl set Open_vSwitch . other_config:dpdk-max-
memzones=50000
      _ovs-vsctl set Open_vSwitch . other_config:hw-offload=true
      _ovs-vsctl set Open_vSwitch . other_config:pmd-quiet-
idle=true
      _ovs-vsctl set Open_vSwitch . other_config:max-idle=20000
      _ovs-vsctl set Open_vSwitch . other_config:max-
revalidator=5000
      _ovs-vsctl --if-exists del-br ovsbr1
      _ovs-vsctl --if-exists del-br ovsbr2
      _ovs-vsctl --may-exist add-br br-sfc
      _ovs-vsctl set bridge br-sfc datapath_type=netdev
      _ovs-vsctl set bridge br-sfc fail_mode=secure
      _ovs-vsctl --may-exist add-port br-sfc p0
      _ovs-vsctl set Interface p0 type=dpdk
      _ovs-vsctl set Interface p0 mtu_request=9216
      _ovs-vsctl set Port p0 external_ids:dpf-type=physical
      _ovs-vsctl --may-exist add-port br-sfc p1
      _ovs-vsctl set Interface p1 type=dpdk
      _ovs-vsctl set Interface p1 mtu_request=9216
      _ovs-vsctl set Port p1 external_ids:dpf-type=physical
      _ovs-vsctl --may-exist add-br br-hbn
      _ovs-vsctl set bridge br-hbn datapath_type=netdev
      _ovs-vsctl set bridge br-hbn fail_mode=secure
```

DPUDeployment for VirtioFS Storage

```yaml
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUDeployment
metadata:
  name: hbn-snap
  namespace: dpf-operator-system
spec:
  dpus:
    bfb: bf-bundle-$TAG
    flavor: hbn-snap-virtiofs-$TAG
    nodeEffect:
      noEffect: true
    dpuSets:
    - nameSuffix: "dpuset1"
      dpuAnnotations:
        storage.nvidia.com/preferred-dpu: "true"
      nodeSelector:
        matchLabels:
          feature.node.kubernetes.io/dpu-enabled: "true"
  services:
    doca-hbn:
      serviceTemplate: doca-hbn
      serviceConfiguration: doca-hbn
    snap-host-controller:
      serviceTemplate: snap-host-controller
      serviceConfiguration: snap-host-controller
    snap-node-driver:
      serviceTemplate: snap-node-driver
      serviceConfiguration: snap-node-driver
    doca-snap:
      serviceTemplate: doca-snap
      serviceConfiguration: doca-snap
    fs-storage-dpu-plugin:
      serviceTemplate: fs-storage-dpu-plugin
      serviceConfiguration: fs-storage-dpu-plugin
```

```
    nfs-csi-controller:
      serviceTemplate: nfs-csi-controller
      serviceConfiguration: nfs-csi-controller
    nfs-csi-controller-dpu:
      serviceTemplate: nfs-csi-controller-dpu
      serviceConfiguration: nfs-csi-controller-dpu
  serviceChains:
    switches:
      - ports:
        - serviceInterface:
            matchLabels:
              interface: p0
        - service:
            name: doca-hbn
            interface: p0_if
      - ports:
        - serviceInterface:
            matchLabels:
              interface: p1
        - service:
            name: doca-hbn
            interface: p1_if
      - ports:
        - serviceInterface:
            matchLabels:
              interface: pf0hpf
        - service:
            name: doca-hbn
            interface: pf0hpf_if
      - ports:
        - serviceInterface:
            matchLabels:
              interface: pf1hpf
        - service:
            name: doca-hbn
            interface: pf1hpf_if
```

```
      - ports:
        - service:
            name: doca-snap
            interface: app_sf
            ipam:
              matchLabels:
                svc.dpu.nvidia.com/pool: storage-pool
        - service:
            name: fs-storage-dpu-plugin
            interface: app_sf
            ipam:
              matchLabels:
                svc.dpu.nvidia.com/pool: storage-pool
        - service:
            name: doca-hbn
            interface: snap_if
```

DPUServiceIPAM

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceIPAM
metadata:
  name: storage-pool
  namespace: dpf-operator-system
spec:
  metadata:
    labels:
      svc.dpu.nvidia.com/pool: storage-pool
  ipv4Subnet:
    subnet: "10.0.124.0/24"
    gateway: "10.0.124.1"
    perNodeIPCount: 8
```

DPUServiceConfiguration and DPUServiceTemplate for DOCA HBN

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceConfiguration
metadata:
  name: doca-hbn
  namespace: dpf-operator-system
spec:
  deploymentServiceName: "doca-hbn"
  serviceConfiguration:
    serviceDaemonSet:
      annotations:
        k8s.v1.cni.cncf.io/networks: |-
          [
          {"name": "iprequest", "interface": "ip_lo", "cni-args": {"poolNames": ["loopback"], "poolType": "cidrpool"}},
          {"name": "iprequest", "interface": "ip_pf0hpf", "cni-args": {"poolNames": ["pool1"], "poolType": "cidrpool", "allocateDefaultGateway": true}},
          {"name": "iprequest", "interface": "ip_pf1hpf", "cni-args": {"poolNames": ["pool2"], "poolType": "cidrpool", "allocateDefaultGateway": true}}
          ]
    helmChart:
      values:
        configuration:
          perDPUValuesYAML: |
            - hostnamePattern: "*"
              values:
                bgp_peer_group: hbn
                vrf1: RED
                vrf2: BLUE
                l3vni1: 100001
                l3vni2: 100002
            - hostnamePattern: "dpu-node-${DPU1_SERIAL}*"
              values:
                bgp_autonomous_system: 65101
            - hostnamePattern: "dpu-node-${DPU2_SERIAL}*"
```

```
            values:
              bgp_autonomous_system: 65201
        startupYAMLJ2: |
          - header:
              model: bluefield
              nvue-api-version: nvue_v1
              rev-id: 1.0
              version: HBN 3.0.0
          - set:
              evpn:
                enable: on
                route-advertise: {}
              bridge:
                domain:
                  br_default:
                    vlan:
                      '10':
                        vni:
                          '10': {}
              interface:
                lo:
                  ip:
                    address:
                      {{ ipaddresses.ip_lo.ip }}/32: {}
                  type: loopback
                p0_if,p1_if,pf0hpf_if,pf1hpf_if:
                  type: swp
                  link:
                    mtu: 9000
                pf0hpf_if:
                  ip:
                    address:
                      {{ ipaddresses.ip_pf0hpf.cidr }}: {}
                    vrf: {{ config.vrf1 }}
                pf1hpf_if:
                  ip:
```

```
            address:
              {{ ipaddresses.ip_pf1hpf.cidr }}: {}
            vrf: {{ config.vrf2 }}
          snap_if:
            bridge:
              domain:
                br_default:
                  access: 10
          vlan10:
            type: svi
            vlan: 10
        nve:
          vxlan:
            arp-nd-suppress: on
            enable: on
            source:
              address: {{ ipaddresses.ip_lo.ip }}
        router:
          bgp:
            enable: on
            graceful-restart:
              mode: full
        vrf:
          default:
            router:
              bgp:
                address-family:
                  ipv4-unicast:
                    enable: on
                    redistribute:
                      connected:
                        enable: on
                    multipaths:
                      ebgp: 16
                  l2vpn-evpn:
                    enable: on
```

```
                        autonomous-system: {{
config.bgp_autonomous_system }}
                        enable: on
                        neighbor:
                          p0_if:
                            peer-group: {{ config.bgp_peer_group
}}
                            type: unnumbered
                            address-family:
                              l2vpn-evpn:
                                enable: on
                                add-path-tx: off
                          p1_if:
                            peer-group: {{ config.bgp_peer_group
}}
                            type: unnumbered
                            address-family:
                              l2vpn-evpn:
                                enable: on
                                add-path-tx: off
                        path-selection:
                          multipath:
                            aspath-ignore: on
                        peer-group:
                          {{ config.bgp_peer_group }}:
                            address-family:
                              ipv4-unicast:
                                enable: on
                              l2vpn-evpn:
                                enable: on
                            remote-as: external
                        router-id: {{ ipaddresses.ip_lo.ip }}
                    {{ config.vrf1 }}:
                      evpn:
                        enable: on
                        vni:
```

```
                        {{ config.l3vni1 }}: {}
                    router:
                      bgp:
                        address-family:
                          ipv4-unicast:
                            enable: on
                            redistribute:
                              connected:
                                enable: on
                            route-export:
                              to-evpn:
                                enable: on
                        autonomous-system: {{
config.bgp_autonomous_system }}
                        enable: on
                  {{ config.vrf2 }}:
                    evpn:
                      enable: on
                      vni:
                        {{ config.l3vni2 }}: {}
                    router:
                      bgp:
                        address-family:
                          ipv4-unicast:
                            enable: on
                            redistribute:
                              connected:
                                enable: on
                            route-export:
                              to-evpn:
                                enable: on
                        autonomous-system: {{
config.bgp_autonomous_system }}
                        enable: on

  interfaces:
```

```
  - name: p0_if
    network: mybrhbn
  - name: p1_if
    network: mybrhbn
  - name: pf0hpf_if
    network: mybrhbn
  - name: pf1hpf_if
    network: mybrhbn
  - name: snap_if
    network: mybrhbn
```

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceTemplate
metadata:
  name: doca-hbn
  namespace: dpf-operator-system
spec:
  deploymentServiceName: "doca-hbn"
  helmChart:
    source:
      repoURL: $HELM_REGISTRY_REPO_URL
      version: 1.0.5
      chart: doca-hbn
    values:
      image:
        repository: $HBN_NGC_IMAGE_URL
        tag: 3.2.1-doca3.2.1
      resources:
        memory: 6Gi
        nvidia.com/bf_sf: 5
```

DPUServiceConfiguration and DPUServiceTemplate for DOCA SNAP

```yaml
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceConfiguration
metadata:
  name: doca-snap
  namespace: dpf-operator-system
spec:
  deploymentServiceName: doca-snap
  serviceConfiguration:
    helmChart:
      values:
        dpu:
          docaSnap:
            enabled: true
            env:
              XLIO_ENABLED: "0"
            image:
              repository: $SNAP_NGC_IMAGE_URL
              tag: 1.5.0-doca3.2.0
  interfaces:
  - name: app_sf
    network: mybrsfc
```

```yaml
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceTemplate
metadata:
  name: doca-snap
  namespace: dpf-operator-system
spec:
  deploymentServiceName: doca-snap
  helmChart:
    source:
      repoURL: $REGISTRY
      version: $TAG
      chart: dpf-storage
    values:
      serviceDaemonSet:
        resources:
          memory: "2Gi"
          hugepages-2Mi: "4Gi"
          cpu: "8"
          nvidia.com/bf_sf: 1
  resourceRequirements:
    memory: "2Gi"
    hugepages-2Mi: "4Gi"
    cpu: "8"
    nvidia.com/bf_sf: 1
```

DPUServiceConfiguration and DPUServiceTemplate for SNAP Node Driver

```yaml
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceConfiguration
metadata:
  name: snap-node-driver
  namespace: dpf-operator-system
spec:
  deploymentServiceName: snap-node-driver
  serviceConfiguration:
    helmChart:
      values:
        dpu:
          deployCrds: true
          snapNodeDriver:
            enabled: true
```

```yaml
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceTemplate
metadata:
  name: snap-node-driver
  namespace: dpf-operator-system
spec:
  deploymentServiceName: snap-node-driver
  helmChart:
    source:
      repoURL: $REGISTRY
      version: $TAG
      chart: dpf-storage
```

DPUServiceConfiguration and DPUServiceTemplate for FS Storage DPU Plugin

```yaml
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceConfiguration
metadata:
  name: fs-storage-dpu-plugin
  namespace: dpf-operator-system
spec:
  deploymentServiceName: fs-storage-dpu-plugin
  serviceConfiguration:
    helmChart:
      values:
        dpu:
          fsStorageVendorDpuPlugin:
            enabled: true
  interfaces:
    - name: app_sf
      network: mybrsfc
```

```yaml
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceTemplate
metadata:
  name: fs-storage-dpu-plugin
  namespace: dpf-operator-system
spec:
  deploymentServiceName: fs-storage-dpu-plugin
  helmChart:
    source:
      repoURL: $REGISTRY
      version: $TAG
      chart: dpf-storage
    values:
      serviceDaemonSet:
        resources:
          nvidia.com/bf_sf: 1
  resourceRequirements:
    nvidia.com/bf_sf: 1
```

DPUServiceConfiguration and DPUServiceTemplate for NFS CSI Controller

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceConfiguration
metadata:
  name: nfs-csi-controller
  namespace: dpf-operator-system
spec:
  deploymentServiceName: nfs-csi-controller
  upgradePolicy:
    applyNodeEffect: false
  serviceConfiguration:
    deployInCluster: true
    helmChart:
      values:
        host:
          enabled: true
          config:
            # required parameter, name of the secret that
contains connection
            # details to access the DPU cluster.
            # this secret should be created by the
DPUServiceCredentialRequest API.
            dpuClusterSecret: nfs-csi-controller-dpu-cluster-
credentials
```

```yaml
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceTemplate
metadata:
  name: nfs-csi-controller
  namespace: dpf-operator-system
spec:
  deploymentServiceName: nfs-csi-controller
  helmChart:
    source:
      repoURL: oci://ghcr.io/mellanox/dpf-storage-vendors-charts
      version: v0.2.0
      chart: nfs-csi-controller
```

DPUServiceConfiguration and DPUServiceTemplate for NFS CSI Controller on DPU

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceConfiguration
metadata:
  name: nfs-csi-controller-dpu
  namespace: dpf-operator-system
spec:
  deploymentServiceName: nfs-csi-controller-dpu
  upgradePolicy:
    applyNodeEffect: false
  serviceConfiguration:
    helmChart:
      values:
        dpu:
          enabled: true
          storageClasses:
            # List of storage classes to be created for nfs-csi
            # These StorageClass names should be used in the
StorageVendor settings
            - name: nfs-csi
              parameters:
                server: 10.0.124.1
                share: /srv/nfs/share
          rbacRoles:
            nfsCsiController:
              # the name of the service account for nfs-csi-
controller
              # this value must be aligned with the value from the
DPUServiceCredentialRequest
              serviceAccount: nfs-csi-controller-sa
```

```yaml
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceTemplate
metadata:
  name: nfs-csi-controller-dpu
  namespace: dpf-operator-system
spec:
  deploymentServiceName: nfs-csi-controller-dpu
  helmChart:
    source:
      repoURL: oci://ghcr.io/mellanox/dpf-storage-vendors-charts
      version: v0.2.0
      chart: nfs-csi-controller
```

DPUServiceCredentialRequest for NFS CSI Controller

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceCredentialRequest
metadata:
  name: nfs-csi-controller-credentials
  namespace: dpf-operator-system
spec:
  duration: 24h
  serviceAccount:
    name: nfs-csi-controller-sa
    namespace: dpf-operator-system
  targetCluster:
    name: dpu-cplane-tenant1
    namespace: dpu-cplane-tenant1
  type: tokenFile
  secret:
    name: nfs-csi-controller-dpu-cluster-credentials
    namespace: dpf-operator-system
```

DPUServiceConfiguration and DPUServiceTemplate for SNAP Host Controller

```yaml
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceConfiguration
metadata:
  name: snap-host-controller
  namespace: dpf-operator-system
spec:
  deploymentServiceName: snap-host-controller
  upgradePolicy:
    applyNodeEffect: false
  serviceConfiguration:
    deployInCluster: true
    helmChart:
      values:
        host:
          snapHostController:
            enabled: true
            config:
              targetNamespace: dpf-operator-system
            affinity:
              nodeAffinity:
                requiredDuringSchedulingIgnoredDuringExecution:
                  nodeSelectorTerms:
                  - matchExpressions:
                      - key: "node-role.kubernetes.io/master"
                        operator: Exists
                  - matchExpressions:
                      - key: "node-role.kubernetes.io/control-plane"
                        operator: Exists
```

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceTemplate
metadata:
  name: snap-host-controller
  namespace: dpf-operator-system
spec:
  deploymentServiceName: snap-host-controller
  helmChart:
    source:
      repoURL: $REGISTRY
      version: $TAG
      chart: dpf-storage
```

Verification

These verification commands may need to be run multiple times to ensure the condition is met.

Note that the DPUService name will have a random suffix. For example, `doca-hbn-l2xsl`.

Verify the DPU and Service installation with:

```
## Ensure the DPUServices are created and have been reconciled.
kubectl wait --for=condition=ApplicationsReconciled --namespace
dpf-operator-system dpuservices -l svc.dpu.nvidia.com/owned-by-
dpudeployment=dpf-operator-system_hbn-snap
## Ensure the DPUServiceIPAMs have been reconciled
kubectl wait --for=condition=DPUIPAMObjectReconciled --namespace
dpf-operator-system dpuserviceipam --all
## Ensure the DPUServiceInterfaces have been reconciled
kubectl wait --for=condition=ServiceInterfaceSetReconciled --
namespace dpf-operator-system dpuserviceinterface --all
## Ensure the DPUServiceChains have been reconciled
kubectl wait --for=condition=ServiceChainSetReconciled --namespace
dpf-operator-system dpuservicechain --all
## Ensure the DPUs have the condition Initialized (this may take
time)
kubectl wait --for=condition=Initialized --namespace dpf-operator-
system dpu --all
```

or with `dpfctl`:

```
$ kubectl -n dpf-operator-system exec deploy/dpf-operator-
controller-manager -- /dpfctl describe dpudeployments
```

Apply Storage Configuration

A number of environment variables must be set before running this command.

```
cat manifests/04.2-storage-configuration-virtiofs/*.yaml |
envsubst | kubectl apply -f -
```

This will create the following objects:

DPUStorageVendor for NFS CSI

```
---
apiVersion: storage.dpu.nvidia.com/v1alpha1
kind: DPUStorageVendor
metadata:
  name: nfs-csi
  namespace: dpf-operator-system
spec:
  storageClassName: nfs-csi
  pluginName: nvidia-fs
```

DPUStoragePolicy for filesystem storage

```
---
apiVersion: storage.dpu.nvidia.com/v1alpha1
kind: DPUStoragePolicy
metadata:
  name: policy-fs
  namespace: dpf-operator-system
spec:
  dpuStorageVendors:
    - nfs-csi
  selectionAlgorithm: "NumberVolumes"
  parameters: {}
```

Check that the objects are ready:

```
kubectl wait --for=condition=Ready --namespace dpf-operator-system
dpustoragevendors --all
kubectl wait --for=condition=Ready --namespace dpf-operator-system
dpustoragepolicies --all
```

Making the DPUs Ready

In order to make the DPUs ready, we will need to manually power cycle the hosts. This operation should be done in the most graceful manner by gracefully shutting down the Host and DPU, powering off the server and then powering it on to avoid corruption. This should happen when the object gives us the signal. The described flow can be automated by the admin depending on the infrastructure. The following verification command may need to be run multiple times to ensure the condition is met.

```
## Ensure the DPUs are in the Rebooting phase and condition
Rebooted is false with WaitingForManualPowerCycleOrReboot reason
kubectl wait --for=jsonpath='{.status.conditions[?
(@.type=="Rebooted")].reason}'=WaitingForManualPowerCycleOrReboot --
namespace dpf-operator-system dpu --all
```

or with `dpfctl`:

```
$ kubectl -n dpf-operator-system exec deploy/dpf-operator-
controller-manager -- /dpfctl describe dpudeployments
```

At this point, we have to power cycle the hosts.

```
kubectl annotate dpunodes -n dpf-operator-system --all
provisioning.dpu.nvidia.com/dpunode-external-reboot-required-
```

After this is done, we should expect that all DPUs become Ready:

```
kubectl wait --for="jsonpath={.status.phase}=Ready" --namespace dpf-operator-system dpu --all
```

or with dpfctl:

```
$ kubectl -n dpf-operator-system exec deploy/dpf-operator-controller-manager -- /dpfctl describe dpudeployments
```

Test VirtioFS Storage

A number of environment variables must be set before running this command.

Create DPUVolume and DPUVolumeAttachment for VirtioFS storage:

```
cat manifests/05.2-storage-test-virtiofs/*.yaml | envsubst | kubectl apply -f -
```

This will create the following objects:

DPUVolume for VirtioFS storage

```
---
apiVersion: storage.dpu.nvidia.com/v1alpha1
kind: DPUVolume
metadata:
  name: test-volume-virtiofs-hotplug-pf-${DPU1_SERIAL}
  namespace: dpf-operator-system
spec:
  dpuStoragePolicyName: policy-fs
  resources:
    requests:
      storage: 10Gi
  accessModes:
  - ReadWriteOnce
  volumeMode: Filesystem
---
apiVersion: storage.dpu.nvidia.com/v1alpha1
kind: DPUVolume
metadata:
  name: test-volume-virtiofs-hotplug-pf-${DPU2_SERIAL}
  namespace: dpf-operator-system
spec:
  dpuStoragePolicyName: policy-fs
  resources:
    requests:
      storage: 10Gi
  accessModes:
  - ReadWriteOnce
  volumeMode: Filesystem
```

DPUVolumeAttachment for VirtioFS storage

```
---
apiVersion: storage.dpu.nvidia.com/v1alpha1
kind: DPUVolumeAttachment
metadata:
  name: test-volume-attachment-virtiofs-hotplug-pf-${DPU1_SERIAL}
  namespace: dpf-operator-system
spec:
  dpuNodeName: dpu-node-${DPU1_SERIAL}
  dpuVolumeName: test-volume-virtiofs-hotplug-pf-${DPU1_SERIAL}
  functionType: pf
  hotplugFunction: true
---
apiVersion: storage.dpu.nvidia.com/v1alpha1
kind: DPUVolumeAttachment
metadata:
  name: test-volume-attachment-virtiofs-hotplug-pf-${DPU2_SERIAL}
  namespace: dpf-operator-system
spec:
  dpuNodeName: dpu-node-${DPU2_SERIAL}
  dpuVolumeName: test-volume-virtiofs-hotplug-pf-${DPU2_SERIAL}
  functionType: pf
  hotplugFunction: true
```

Verify that the DPUVolume resources are in the Ready state:

```
kubectl wait --for=condition=Ready --namespace dpf-operator-system
dpuvolumes --all
```

Wait for the DPUVolumeAttachments to become Ready:

```
kubectl wait --for=condition=Ready --namespace dpf-operator-system
dpuvolumeattachments --all
```

On the host that is used to run this guide, get the VirtioFS tag name of the DPUVolumeAttachment:

```
kubectl wait --for=condition=Ready dpuvolumeattachments/test-
volume-attachment-virtiofs-hotplug-pf-${DPU1_SERIAL} -n dpf-
operator-system
kubectl wait --for=condition=Ready dpuvolumeattachments/test-
volume-attachment-virtiofs-hotplug-pf-${DPU2_SERIAL} -n dpf-
operator-system
tag_1=$(kubectl get -n dpf-operator-system
dpuvolumeattachments.storage.dpu.nvidia.com test-volume-
attachment-virtiofs-hotplug-pf-${DPU1_SERIAL} -o
jsonpath='{.status.dpu.virtioFSAttrs.filesystemTag}')
tag_2=$(kubectl get -n dpf-operator-system
dpuvolumeattachments.storage.dpu.nvidia.com test-volume-
attachment-virtiofs-hotplug-pf-${DPU2_SERIAL} -o
jsonpath='{.status.dpu.virtioFSAttrs.filesystemTag}')
echo "Worker $DPU1_SERIAL VirtioFS tag: $tag_1"
echo "Worker $DPU2_SERIAL VirtioFS tag: $tag_2"
```

Connect to the worker nodes with DPUs and set tag variable to point to the VirtioFS tag retrieved in the previous step.

```
tag=<set to the VirtioFS tag that you retrieved in the previous
step>
```

Make sure the VirtioFS driver is loaded:

```
modprobe virtiofs
```

Create a directory to mount the volume:

```
mkdir -p /tmp/test-volume-virtiofs-hotplug-pf
```

Mount the volume:

```
mount -t virtiofs $tag /tmp/test-volume-virtiofs-hotplug-pf
```

Perform I/O operations to test the storage:

```
dd if=/dev/zero of=/tmp/test-volume-virtiofs-hotplug-pf/test.txt
bs=1M count=1000 status=progress
```

Unmount the volume:

```
umount /tmp/test-volume-virtiofs-hotplug-pf
```

Test Network Traffic

Both Block and VirtioFS scenarios can be tested with the same steps.

After the DPUs are provisioned and the rest of the objects are Ready, we can test traffic by assigning an IP to the PF0 on the host for each DPU, and run a simple ping. Although the configuration is enabling both PFs, we focus on the PF0 for testing traffic. Assuming the PF0 is named `ens5f0np0`:

On the host with DPU with serial number DPU1_SERIAL:

```
ip link set dev ens5f0np0 up
ip addr add 10.0.121.1/29 dev ens5f0np0
ip route add 10.0.121.0/24 dev ens5f0np0 via 10.0.121.2
```

On the host with DPU with serial number DPU2_SERIAL:

```
ip link set dev ens5f0np0 up
ip addr add 10.0.121.9/29 dev ens5f0np0
ip route add 10.0.121.0/24 dev ens5f0np0 via 10.0.121.10
```

On the host with DPU with serial number DPU1_SERIAL:

```
$ ping 10.0.121.9 -c3
PING 10.0.121.9 (10.0.121.9) 56(84) bytes of data.
64 bytes from 10.0.121.9: icmp_seq=1 ttl=64 time=0.387 ms
64 bytes from 10.0.121.9: icmp_seq=2 ttl=64 time=0.344 ms
64 bytes from 10.0.121.9: icmp_seq=3 ttl=64 time=0.396 ms

--- 10.0.121.9 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2053ms
rtt min/avg/max/mdev = 0.344/0.375/0.396/0.022 ms
```

Uninstall

This section covers only the DPF related components and not the prerequisites as these must be managed by the admin.

Delete storage resources

> **Warning**
>
> Be sure to unmount all volumes from the worker nodes before deleting the DPUVolumeAttachments or the operation may fail.

```
kubectl delete -n dpf-operator-system dpuvolumeattachments --all
--wait
kubectl delete -n dpf-operator-system dpuvolumes --all --wait
kubectl delete -n dpf-operator-system dpustoragepolicies --all --
wait
kubectl delete -n dpf-operator-system dpustoragevendors --all --
wait
```

Delete the DPF Operator system and DPF Operator

```
kubectl delete -n dpf-operator-system dpfoperatorconfig
dpfoperatorconfig --wait
helm uninstall -n dpf-operator-system dpf-operator --wait
```

Delete DPF Operator PVC

```
kubectl -n dpf-operator-system delete pvc bfb-pvc
kubectl delete pv bfb-pv
```

Note: there can be a race condition with deleting the underlying Kamaji cluster which runs the DPU cluster control plane in this guide. If that happens it may be necessary to remove

finalizers manually from `DPUCluster` and `Datastore` objects.

HBN in DPF Zero Trust

> **Note**
>
> Follow this guide from the source GitHub repo at github.com/NVIDIA/doca-platform and moving to the `docs/public/user-guides/zero-trust/use-cases/hbn/README.md` for better formatting of the code.

This configuration provides instructions for deploying the NVIDIA DOCA Platform Framework (DPF) on high-performance, bare-metal infrastructure in Zero Trust mode, utilizing DPU BMC and Redfish. It focuses on provisioning NVIDIA® BlueField®-3 DPUs using DPF, installing the HBN DPUService on those DPUs and enabling workload traffic to pass through HBN before leaving the DPU.

Prerequisites

This guide should be run by cloning the repo from github.com/NVIDIA/doca-platform and moving to the `docs/public/user-guides/zero-trust/use-cases/hbn` directory.

The system is set up as described in the prerequisites.

In addition, for this use case, the Top of Rack switch(ToR) should be configured to support unnumbered BGP towards the two ports of the DPU, where HBN will act as peer, and advertise routes over BGP to allow for ECMP from the DPU. Additional information about how to do that can be found in the RDG for DPF Zero Trust (DPF-ZT) with HBN DPU Service.

Software prerequisites

The following tools must be installed on the machine where the commands contained in this guide run:

- kubectl
- helm
- envsubst

Installation guide

This guide assumes that the setup includes only 2 workers with DPUs. If your setup has more than 2 workers, then you will need to set additional variables to enable the rest of the DPUs.

0. Required variables

The following variables are required by this guide. A sensible default is provided where it makes sense, but many will be specific to the target infrastructure.

Commands in this guide are run in the same directory that contains this readme.

Environment variables file

```
## IP Address for the Kubernetes API server of the target cluster
on which DPF is installed.
## This should never include a scheme or a port.
## e.g. 10.10.10.10
export TARGETCLUSTER_API_SERVER_HOST=

## Virtual IP used by the load balancer for the DPU Cluster. Must
be a reserved IP from the management subnet and not
## allocated by DHCP.
export DPUCLUSTER_VIP=

## Interface on which the DPUCluster load balancer will listen.
Should be the management interface of the control plane node.
export DPUCLUSTER_INTERFACE=

## IP address to the NFS server used as storage for the BFB.
export NFS_SERVER_IP=

## The repository URL for the NVIDIA Helm chart registry.
## Usually this is the NVIDIA Helm NGC registry. For development
purposes, this can be set to a different repository.
export HELM_REGISTRY_REPO_URL=https://helm.ngc.nvidia.com/nvidia/doca

## The repository URL for the HBN container image.
## Usually this is the NVIDIA NGC registry. For development
purposes, this can be set to a different repository.
export HBN_NGC_IMAGE_URL=nvcr.io/nvidia/doca/doca_hbn

## The DPF REGISTRY is the Helm repository URL where the DPF
Operator Chart resides.
## Usually this is the NVIDIA Helm NGC registry. For development
purposes, this can be set to a different repository.
export REGISTRY=https://helm.ngc.nvidia.com/nvidia/doca
```

```
## The DPF TAG is the version of the DPF components which will be
deployed in this guide.
export TAG=v25.10.0

## URL to the BFB used in the `bfb.yaml` and linked by the
DPUSet.
export BFB_URL="https://content.mellanox.com/BlueField/BFBs/Ubuntu24.04/bf-bundle-3.2.1-
34_25.11_ubuntu-24.04_64k_prod.bfb"

## IP_RANGE_START and IP_RANGE_END
## These define the IP range for DPU discovery via Redfish/BMC
interfaces
## Example: If your DPUs have BMC IPs in range 192.168.1.100-110
## export IP_RANGE_START=192.168.1.100
## export IP_RANGE_END=192.168.1.110
export IP_RANGE_START=

export IP_RANGE_END=

# The password used for DPU BMC root login, must be the same for
all DPUs
export BMC_ROOT_PASSWORD=

## Serial number of DPUs. If you have more than 2 DPUs, you will
need to parameterize the system accordingly and expose
## additional variables.
## All serial numbers must be in lowercase.
export DPU1_SERIAL=

export DPU2_SERIAL=
```

Modify the variables in `manifests/00-env-vars/envvars.env` to fit your environment, then source the file:

```
source manifests/00-env-vars/envvars.env
```

1. DPF Operator installation

Create storage required by the DPF Operator

A number of environment variables must be set before running this command.

```
kubectl create ns dpf-operator-system
cat manifests/01-dpf-operator-installation/*.yaml | envsubst |
kubectl apply -f -
```

This deploys the following objects:

PersistentVolume and PersistentVolumeClaim for the provisioning controller

```
---
apiVersion: v1
kind: PersistentVolume
metadata:
  name: bfb-pv
spec:
  capacity:
    storage: 10Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteMany
  nfs:
    path: /mnt/dpf_share/bfb
    server: $NFS_SERVER_IP
  persistentVolumeReclaimPolicy: Delete
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: bfb-pvc
  namespace: dpf-operator-system
spec:
  accessModes:
  - ReadWriteMany
  resources:
    requests:
      storage: 10Gi
  volumeMode: Filesystem
  storageClassName: ""
```

Create DPU BMC shared password secret

In Zero Trust mode, provisioning DPUs requires authentication with Redfish. In order to do that, you must set the same root password to access the BMC for all DPUs DPF is going

to manage.

For more information on how to set the BMC root password refer to BlueField DPU Administrator Quick Start Guide

The password is provided to DPF by creating the following secret:

```
kubectl create secret generic -n dpf-operator-system bmc-shared-
password --from-literal=password=$BMC_ROOT_PASSWORD
```

Additional Dependencies

Before deploying the DPF Operator, ensure that Helm is properly configured according to the Helm prerequisites.

> ⚠ **Warning**
>
> This is a critical prerequisite step that must be completed for the DPF Operator to function properly.

Deploy the DPF Operator

A number of environment variables must be set before running this command.

HTTP Registry (default)

If the $REGISTRY is an HTTP Registry (default value) use this command:

```
helm repo add --force-update dpf-repository ${REGISTRY}
helm repo update
helm upgrade --install -n dpf-operator-system dpf-operator dpf-
repository/dpf-operator --version=$TAG
```

OCI Registry

For development purposes, if the $REGISTRY is an OCI Registry use this command:

```
helm upgrade --install -n dpf-operator-system dpf-operator
$REGISTRY/dpf-operator --version=$TAG
```

Verification

These verification commands may need to be run multiple times to ensure the condition is met.

Verify the DPF Operator installation with:

```
## Ensure the DPF Operator deployment is available.
kubectl rollout status deployment --namespace dpf-operator-system
dpf-operator-controller-manager
## Ensure all pods in the DPF Operator system are ready.
kubectl wait --for=condition=ready --namespace dpf-operator-system
pods --all
```

2. DPF system installation

This section involves creating the DPF system components and some basic infrastructure required for a functioning DPF-enabled cluster.

Deploy the DPF System components

A number of environment variables must be set before running this command.

```
kubectl create ns dpu-cplane-tenant1
cat manifests/02-dpf-system-installation/*.yaml | envsubst |
kubectl apply -f -
```

This will create the following objects:

DPFOperatorConfig to install the DPF System components

```
---
apiVersion: operator.dpu.nvidia.com/v1alpha1
kind: DPFOperatorConfig
metadata:
  name: dpfoperatorconfig
  namespace: dpf-operator-system
spec:
  dpuDetector:
    disable: true
  provisioningController:
    bfbPVCName: "bfb-pvc"
    dmsTimeout: 900
    installInterface:
      installViaRedfish:
        # Set this to the IP of one of your control plane nodes +
8080 port
        bfbRegistryAddress: "$TARGETCLUSTER_API_SERVER_HOST:8080"
        skipDPUNodeDiscovery: false
  kamajiClusterManager:
    disable: false
```

DPUCluster to serve as Kubernetes control plane for DPU nodes

```
---
apiVersion: provisioning.dpu.nvidia.com/v1alpha1
kind: DPUCluster
metadata:
  name: dpu-cplane-tenant1
  namespace: dpu-cplane-tenant1
spec:
  type: kamaji
  maxNodes: 10
  clusterEndpoint:
    # deploy keepalived instances on the nodes that match the
given nodeSelector.
    keepalived:
      # interface on which keepalived will listen. Should be the oob
interface of the control plane node.
      interface: $DPUCLUSTER_INTERFACE
      # Virtual IP reserved for the DPU Cluster load balancer.
Must not be allocatable by DHCP.
      vip: $DPUCLUSTER_VIP
      # virtualRouterID must be in range [1,255], make sure the
given virtualRouterID does not duplicate with any existing
keepalived process running on the host
      virtualRouterID: 126
      nodeSelector:
        node-role.kubernetes.io/control-plane: ""
```

DPUDiscovery to discover DPUDevices or DPUNodes

```
---
apiVersion: provisioning.dpu.nvidia.com/v1alpha1
kind: DPUDiscovery
metadata:
  name: dpu-discovery
  namespace: dpf-operator-system
spec:
  ipRangeSpec:
    ipRange:
      startIP: $IP_RANGE_START
      endIP: $IP_RANGE_END
```

Verification

These verification commands may need to be run multiple times to ensure the condition is met.

Verify the DPF System with:

```
## Ensure the provisioning and DPUService controller manager
deployments are available.
kubectl rollout status deployment --namespace dpf-operator-system
dpf-provisioning-controller-manager dpuservice-controller-manager
## Ensure all other deployments in the DPF Operator system are
Available.
kubectl rollout status deployment --namespace dpf-operator-system
## Ensure bfb registry daemonset is available
kubectl rollout status daemonset --namespace dpf-operator-system
bfb-registry
## Ensure the DPUCluster is ready for nodes to join.
kubectl wait --for=condition=ready --namespace dpu-cplane-tenant1
dpucluster --all
```

3. DPU Provisioning and Service Installation

There are 2 types of installation a user can do. The first one is using the PFs of the host and the second one is using both PFs and VFs. You should choose the one that fits best on your use case.

In the following section, we provision our DPUs and the services tht will run on them. The user is expected to create a DPUDeployment object that reflects a set of DPUServices that should run on a set of DPUs.

> If you want to learn more about `DPUDeployments`, feel free to check the DPUDeployment documentation.

Using PFs

In this scenario, the PF0 and PF1 are connected to separate VRFs which means that: * PF0 on Host 1 will be able to communicate with PF0 on Host 2 * PF0 on Host 1 will not be able to communicate with PF1 on Host 1 and 2

- PF1 on Host 1 will be able to communicate with PF1 on Host 2
- PF1 on Host 1 will not be able to communicate with PF0 on Host 1 and 2

We make use of a PF on the host to test traffic.

Create the DPUDeployment, DPUServiceConfig, DPUServiceTemplate and other necessary objects

> ⚠ **Warning**
>
> In case more than 1 DPU exists per node, the relevant selector should be applied in the DPUDeployment to select the appropriate DPU. See DPUDeployment - DPUs Configuration to understand more about the selectors.

A number of environment variables must be set before running this command.

```
cat manifests/03.1-dpudeployment-installation-pf/*.yaml | envsubst
| kubectl apply -f -
```

This will deploy the following objects:

BFB to download Bluefield Bitstream to a shared volume

```
---
apiVersion: provisioning.dpu.nvidia.com/v1alpha1
kind: BFB
metadata:
  name: bf-bundle-$TAG
  namespace: dpf-operator-system
spec:
  url: $BFB_URL
```

HBN DPUFlavor to correctly configure the DPUs on provisioning

```
---
apiVersion: provisioning.dpu.nvidia.com/v1alpha1
kind: DPUFlavor
metadata:
  name: hbn-$TAG
  namespace: dpf-operator-system
spec:
  dpuMode: zero-trust
  bfcfgParameters:
  - UPDATE_ATF_UEFI=yes
  - UPDATE_DPU_OS=yes
  - WITH_NIC_FW_UPDATE=yes
  configFiles:
  - operation: override
    path: /etc/mellanox/mlnx-bf.conf
    permissions: "0644"
    raw: |
      ALLOW_SHARED_RQ="no"
      IPSEC_FULL_OFFLOAD="no"
      ENABLE_ESWITCH_MULTIPORT="yes"
  - operation: override
    path: /etc/mellanox/mlnx-ovs.conf
    permissions: "0644"
    raw: |
      CREATE_OVS_BRIDGES="no"
      OVS_DOCA="yes"
  - operation: override
    path: /etc/mellanox/mlnx-sf.conf
    permissions: "0644"
    raw: ""
  grub:
    kernelParameters:
    - console=hvc0
    - console=ttyAMA0
    - earlycon=pl011,0x13010000
```

```
    - fixrttc
    - net.ifnames=0
    - biosdevname=0
    - iommu.passthrough=1
    - cgroup_no_v1=net_prio,net_cls
    - hugepagesz=2048kB
    - hugepages=3072
  nvconfig:
  - device: '*'
    parameters:
    - PF_BAR2_ENABLE=0
    - PER_PF_NUM_SF=1
    - PF_TOTAL_SF=20
    - PF_SF_BAR_SIZE=10
    - NUM_PF_MSIX_VALID=0
    - PF_NUM_PF_MSIX_VALID=1
    - PF_NUM_PF_MSIX=228
    - INTERNAL_CPU_MODEL=1
    - INTERNAL_CPU_OFFLOAD_ENGINE=0
    - SRIOV_EN=1
    - NUM_OF_VFS=46
    - LAG_RESOURCE_ALLOCATION=1
    - LINK_TYPE_P1=ETH
    - LINK_TYPE_P2=ETH
  ovs:
    rawConfigScript: |
      _ovs-vsctl() {
        ovs-vsctl --no-wait --timeout 15 "$@"
      }

      _ovs-vsctl set Open_vSwitch . other_config:doca-init=true
      _ovs-vsctl set Open_vSwitch . other_config:dpdk-max-
memzones=50000
      _ovs-vsctl set Open_vSwitch . other_config:hw-offload=true
      _ovs-vsctl set Open_vSwitch . other_config:pmd-quiet-
idle=true
```

```
      _ovs-vsctl set Open_vSwitch . other_config:max-idle=20000
      _ovs-vsctl set Open_vSwitch . other_config:max-
revalidator=5000
      _ovs-vsctl --if-exists del-br ovsbr1
      _ovs-vsctl --if-exists del-br ovsbr2
      _ovs-vsctl --may-exist add-br br-sfc
      _ovs-vsctl set bridge br-sfc datapath_type=netdev
      _ovs-vsctl set bridge br-sfc fail_mode=secure
      _ovs-vsctl --may-exist add-port br-sfc p0
      _ovs-vsctl set Interface p0 type=dpdk
      _ovs-vsctl set Interface p0 mtu_request=9216
      _ovs-vsctl set Port p0 external_ids:dpf-type=physical
      _ovs-vsctl --may-exist add-br br-hbn
      _ovs-vsctl set bridge br-hbn datapath_type=netdev
      _ovs-vsctl set bridge br-hbn fail_mode=secure
```

DPUDeployment to provision DPUs on worker nodes

```yaml
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUDeployment
metadata:
  name: hbn
  namespace: dpf-operator-system
spec:
  dpus:
    bfb: bf-bundle-$TAG
    flavor: hbn-$TAG
    nodeEffect:
      noEffect: true
    dpuSets:
    - nameSuffix: "dpuset1"
      nodeSelector:
        matchLabels:
          feature.node.kubernetes.io/dpu-enabled: "true"
  services:
    doca-hbn:
      serviceTemplate: doca-hbn
      serviceConfiguration: doca-hbn
  serviceChains:
    switches:
      - ports:
        - serviceInterface:
            matchLabels:
              interface: p0
        - service:
            name: doca-hbn
            interface: p0_if
      - ports:
        - serviceInterface:
            matchLabels:
              interface: p1
        - service:
```

```
            name: doca-hbn
            interface: p1_if
        - ports:
          - serviceInterface:
              matchLabels:
                interface: pf0hpf
          - service:
              name: doca-hbn
              interface: pf0hpf_if
        - ports:
          - serviceInterface:
              matchLabels:
                interface: pf1hpf
          - service:
              name: doca-hbn
              interface: pf1hpf_if
```

DPUServiceConfig and DPUServiceTemplate to deploy HBN workloads to the DPUs

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceConfiguration
metadata:
  name: doca-hbn
  namespace: dpf-operator-system
spec:
  deploymentServiceName: "doca-hbn"
  serviceConfiguration:
    serviceDaemonSet:
      annotations:
        k8s.v1.cni.cncf.io/networks: |-
          [
          {"name": "iprequest", "interface": "ip_lo", "cni-args": {"poolNames": ["loopback"], "poolType": "cidrpool"}},
          {"name": "iprequest", "interface": "ip_pf0hpf", "cni-args": {"poolNames": ["pool1"], "poolType": "cidrpool", "allocateDefaultGateway": true}},
          {"name": "iprequest", "interface": "ip_pf1hpf", "cni-args": {"poolNames": ["pool2"], "poolType": "cidrpool", "allocateDefaultGateway": true}}
          ]
    helmChart:
      values:
        configuration:
          perDPUValuesYAML: |
            - hostnamePattern: "*"
              values:
                bgp_peer_group: hbn
                vrf1: RED
                vrf2: BLUE
                l3vni1: 100001
                l3vni2: 100002
            - hostnamePattern: "dpu-node-${DPU1_SERIAL}*"
              values:
                bgp_autonomous_system: 65101
            - hostnamePattern: "dpu-node-${DPU2_SERIAL}*"
```

```
            values:
              bgp_autonomous_system: 65201
        startupYAMLJ2: |
          - header:
              model: bluefield
              nvue-api-version: nvue_v1
              rev-id: 1.0
              version: HBN 2.4.0
          - set:
              evpn:
                enable: on
                route-advertise: {}
              interface:
                lo:
                  ip:
                    address:
                      {{ ipaddresses.ip_lo.ip }}/32: {}
                  type: loopback
                p0_if,p1_if,pf0hpf_if,pf1hpf_if:
                  type: swp
                  link:
                    mtu: 9000
                pf0hpf_if:
                  ip:
                    address:
                      {{ ipaddresses.ip_pf0hpf.cidr }}: {}
                    vrf: {{ config.vrf1 }}
                pf1hpf_if:
                  ip:
                    address:
                      {{ ipaddresses.ip_pf1hpf.cidr }}: {}
                    vrf: {{ config.vrf2 }}
              nve:
                vxlan:
                  arp-nd-suppress: on
                  enable: on
```

```
                    source:
                      address: {{ ipaddresses.ip_lo.ip }}
                router:
                  bgp:
                    enable: on
                    graceful-restart:
                      mode: full
                vrf:
                  default:
                    router:
                      bgp:
                        address-family:
                          ipv4-unicast:
                            enable: on
                            redistribute:
                              connected:
                                enable: on
                            multipaths:
                              ebgp: 16
                          l2vpn-evpn:
                            enable: on
                        autonomous-system: {{
config.bgp_autonomous_system }}
                        enable: on
                        neighbor:
                          p0_if:
                            peer-group: {{ config.bgp_peer_group
}}
                            type: unnumbered
                            address-family:
                              l2vpn-evpn:
                                enable: on
                                add-path-tx: off
                          p1_if:
                            peer-group: {{ config.bgp_peer_group
}}
```

```
                            type: unnumbered
                            address-family:
                              l2vpn-evpn:
                                enable: on
                                add-path-tx: off
                        path-selection:
                          multipath:
                            aspath-ignore: on
                        peer-group:
                          {{ config.bgp_peer_group }}:
                            address-family:
                              ipv4-unicast:
                                enable: on
                              l2vpn-evpn:
                                enable: on
                            remote-as: external
                        router-id: {{ ipaddresses.ip_lo.ip }}
                    {{ config.vrf1 }}:
                      evpn:
                        enable: on
                        vni:
                          {{ config.l3vni1 }}: {}
                      router:
                        bgp:
                          address-family:
                            ipv4-unicast:
                              enable: on
                              redistribute:
                                connected:
                                  enable: on
                              route-export:
                                to-evpn:
                                  enable: on
                          autonomous-system: {{
config.bgp_autonomous_system }}
                          enable: on
```

```
                    {{ config.vrf2 }}:
                      evpn:
                        enable: on
                        vni:
                          {{ config.l3vni2 }}: {}
                      router:
                        bgp:
                          address-family:
                            ipv4-unicast:
                              enable: on
                              redistribute:
                                connected:
                                  enable: on
                              route-export:
                                to-evpn:
                                  enable: on
                          autonomous-system: {{
config.bgp_autonomous_system }}
                          enable: on

  interfaces:
  - name: p0_if
    network: mybrhbn
  - name: p1_if
    network: mybrhbn
  - name: pf0hpf_if
    network: mybrhbn
  - name: pf1hpf_if
    network: mybrhbn
```

```yaml
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceTemplate
metadata:
  name: doca-hbn
  namespace: dpf-operator-system
spec:
  deploymentServiceName: "doca-hbn"
  helmChart:
    source:
      repoURL: $HELM_REGISTRY_REPO_URL
      version: 1.0.5
      chart: doca-hbn
    values:
      image:
        repository: $HBN_NGC_IMAGE_URL
        tag: 3.2.1-doca3.2.1
      resources:
        memory: 6Gi
        nvidia.com/bf_sf: 4
```

DPUServiceInterfaces for physical ports on the DPU

```yaml
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceInterface
metadata:
  name: p0
  namespace: dpf-operator-system
spec:
  template:
    spec:
      template:
        metadata:
          labels:
            interface: "p0"
        spec:
          interfaceType: physical
          physical:
            interfaceName: p0
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceInterface
metadata:
  name: p1
  namespace: dpf-operator-system
spec:
  template:
    spec:
      template:
        metadata:
          labels:
            interface: "p1"
        spec:
          interfaceType: physical
          physical:
            interfaceName: p1
---
```

```yaml
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceInterface
metadata:
  name: pf0hpf
  namespace: dpf-operator-system
spec:
  template:
    spec:
      template:
        metadata:
          labels:
            interface: "pf0hpf"
        spec:
          interfaceType: pf
          pf:
            pfID: 0
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceInterface
metadata:
  name: pf1hpf
  namespace: dpf-operator-system
spec:
  template:
    spec:
      template:
        metadata:
          labels:
            interface: "pf1hpf"
        spec:
          interfaceType: pf
          pf:
            pfID: 1
```

DPUServiceIPAM to set up IP Address Management on the DPUCluster

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceIPAM
metadata:
  name: pool1
  namespace: dpf-operator-system
spec:
  ipv4Network:
    network: "10.0.121.0/24"
    gatewayIndex: 2
    prefixSize: 29
    # These preallocations are not necessary. We specify them so
that the validation commands are straightforward.
    allocations:
      dpu-node-${DPU1_SERIAL}-${DPU1_SERIAL}: 10.0.121.0/29
      dpu-node-${DPU2_SERIAL}-${DPU2_SERIAL}: 10.0.121.8/29
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceIPAM
metadata:
  name: pool2
  namespace: dpf-operator-system
spec:
  ipv4Network:
    network: "10.0.122.0/24"
    gatewayIndex: 2
    prefixSize: 29
```

DPUServiceIPAM for the loopback interface in HBN

```yaml
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceIPAM
metadata:
  name: loopback
  namespace: dpf-operator-system
spec:
  ipv4Network:
    network: "11.0.0.0/24"
    prefixSize: 32
```

Verification

These verification commands may need to be run multiple times to ensure the condition is met.

Note that the DPUService name will have a random suffix. For example, `doca-hbn-l2xsl`.

Verify the DPU and Service installation with:

```
## Ensure the DPUServices are created and have been reconciled.
kubectl wait --for=condition=ApplicationsReconciled --namespace
dpf-operator-system dpuservices -l svc.dpu.nvidia.com/owned-by-
dpudeployment=dpf-operator-system_hbn
## Ensure the DPUServiceIPAMs have been reconciled
kubectl wait --for=condition=DPUIPAMObjectReconciled --namespace
dpf-operator-system dpuserviceipam --all
## Ensure the DPUServiceInterfaces have been reconciled
kubectl wait --for=condition=ServiceInterfaceSetReconciled --
namespace dpf-operator-system dpuserviceinterface --all
## Ensure the DPUServiceChains have been reconciled
kubectl wait --for=condition=ServiceChainSetReconciled --namespace
dpf-operator-system dpuservicechain --all
## Ensure the DPUs have the condition Initialized (this may take
time)
kubectl wait --for=condition=Initialized --namespace dpf-operator-
system dpu --all
```

or with `dpfctl`:

```
$ kubectl -n dpf-operator-system exec deploy/dpf-operator-
controller-manager -- /dpfctl describe dpudeployments
NAME                                                    NAMESPACE
STATUS       REASON                SINCE  MESSAGE
DPFOperatorConfig/dpfoperatorconfig                     dpf-operator-
system
              Ready
False        Pending               17m    The following
conditions are not ready:

* SystemComponentsReady
              SystemComponentsReady
False        Error                 16m    System components must
be ready for DPF Operator to continue:

* nvidia-k8s-ipam: DPUService dpf-operator-system/nvidia-k8s-ipam
is not ready
  DPUDeployments
    DPUDeployment/hbn                                   dpf-operator-
system
                  Ready
False        Pending               11m    The following
conditions are not ready:

* DPUSetsReady
                  DPUSetsReady
False        Pending               11m    Objects are not ready:

* dpf-operator-system/hbn-dpuset1
      DPUServiceChains
        DPUServiceChain/hbn-8kkjz                       dpf-operator-
system  Ready: True  Success               11m
      DPUServiceInterfaces
        4 DPUServiceInterfaces...                       dpf-operator-
system  Ready: True  Success               11m    See doca-hbn-
```

```
p0-if-mcqp4, doca-hbn-p1-if-6x2hh, doca-hbn-pf0hpf-if-q9lvk, doca-
hbn-pf1hpf-if-979t7
      DPUSets
        DPUSet/hbn-dpuset1                                dpf-operator-
system
          BFB/bf-bundle                                   dpf-operator-
system  Ready: True  Ready                 13m    File: bf-
bundle-3.2.1-34_25.11_ubuntu-24.04_64k_prod.bfb, DOCA: 3.2.1
          DPUs
            DPU/dpu-node-mt2402xz0f6v-mt2402xz0f6v  dpf-operator-
system
                          Ready
False        OS Installing         8m39s
            DPU/dpu-node-mt2404xz0c98-mt2404xz0c98  dpf-operator-
system
                          Ready
False        OS Installing         8m39s
      Services
        DPUServiceTemplates
          DPUServiceTemplate/doca-hbn                     dpf-operator-
system  Ready: True  Success               13m
        DPUServices
          DPUService/doca-hbn-jmj45                       dpf-operator-
system  Ready: True  Success               11m
```

Making the DPUs Ready

In order to make the DPUs ready, we will need to manually power cycle the host. This operation should be done in the most graceful manner by gracefully shutting down the Host and DPU, powering off the server and then powering it on to avoid corruption. This should happen when the object gives us the signal. The described flow can be automated by the admin depending on the infrastructure.

The following verification command may need to be run multiple times to ensure the condition is met.

```
## Ensure the DPUs have the condition
WaitingForManualPowerCycleOrReboot (this may take time)
kubectl wait --for=condition=WaitingForManualPowerCycleOrReboot --
namespace dpf-operator-system dpu --all
```

or with `dpfctl`:

```
$ kubectl -n dpf-operator-system exec deploy/dpf-operator-
controller-manager -- /dpfctl describe dpudeployments
NAME                                                NAMESPACE
STATUS       REASON                                SINCE  MESSAGE
DPFOperatorConfig/dpfoperatorconfig                 dpf-operator-
system
              Ready
False        Pending                               66m    The
following conditions are not ready:

* SystemComponentsReady
              SystemComponentsReady
False        Error                                 66m    System
components must be ready for DPF Operator to continue:

* nvidia-k8s-ipam: DPUService dpf-operator-system/nvidia-k8s-ipam
is not ready
  DPUDeployments
    DPUDeployment/hbn                               dpf-operator-
system
                  Ready
False        Pending                               61m    The
following conditions are not ready:

* DPUSetsReady
                  DPUSetsReady
False        Pending                               61m    Objects
are not ready:

* dpf-operator-system/hbn-dpuset1
      DPUServiceChains
        DPUServiceChain/hbn-8kkjz                   dpf-operator-
system  Ready: True  Success                         61m
      DPUServiceInterfaces
```

```
        4 DPUServiceInterfaces...                          dpf-operator-
system  Ready: True  Success                                    61m
See doca-hbn-p0-if-mcqp4, doca-hbn-p1-if-6x2hh, doca-hbn-pf0hpf-if-
q9lvk, doca-hbn-pf1hpf-if-979t7
      DPUSets
        DPUSet/hbn-dpuset1                                 dpf-operator-
system
          BFB/bf-bundle                                    dpf-operator-
system  Ready: True  Ready                                      62m
File: bf-bundle-3.2.1-34_25.11_ubuntu-24.04_64k_prod.bfb, DOCA:
3.2.1
          DPUs
            DPU/dpu-node-mt2402xz0f6v-mt2402xz0f6v  dpf-operator-
system
                          Rebooted
False        WaitingForManualPowerCycleOrReboot  11m
                          Ready
False        Rebooting                           11m
            DPU/dpu-node-mt2404xz0c98-mt2404xz0c98  dpf-operator-
system
                          Rebooted
False        WaitingForManualPowerCycleOrReboot  5m49s
                          Ready
False        Rebooting                           5m49s
      Services
        DPUServiceTemplates
          DPUServiceTemplate/doca-hbn                      dpf-operator-
system  Ready: True  Success                                    62m
        DPUServices
          DPUService/doca-hbn-jmj45                        dpf-operator-
system  Ready: True  Success                                    61m
```

At this point, we have to power cycle the hosts. Once **all the hosts are back online**, we have to remove an annotation from the DPUNodes. The user can choose to remove this annotation node by node but to make it simpler in this guide, we do that all at once.

```
kubectl annotate dpunodes -n dpf-operator-system --all
provisioning.dpu.nvidia.com/dpunode-external-reboot-required-
```

After this is done, we should expect that all DPUs become Ready:

```
kubectl wait --for="jsonpath={.status.phase}=Ready" --namespace dpf-operator-
system dpu --all
```

or with dpfctl:

```
$ kubectl -n dpf-operator-system exec deploy/dpf-operator-
controller-manager -- /dpfctl describe dpudeployments
NAME                                      NAMESPACE
STATUS       REASON    SINCE  MESSAGE
DPFOperatorConfig/dpfoperatorconfig    dpf-operator-system
Ready: True  Success   8m19s
  DPUDeployments
    DPUDeployment/hbn                  dpf-operator-system
Ready: True  Success   19s
      DPUServiceChains
        DPUServiceChain/hbn-8kkjz      dpf-operator-system
Ready: True  Success   90m
      DPUServiceInterfaces
        4 DPUServiceInterfaces...      dpf-operator-system
Ready: True  Success   48s    See doca-hbn-p0-if-mls69, doca-hbn-
p1-if-dv6ds, doca-hbn-pf0hpf-if-q9lvk, doca-hbn-pf1hpf-if-979t7
      DPUSets
        DPUSet/hbn-dpuset1             dpf-operator-system
          BFB/bf-bundle                dpf-operator-system
Ready: True  Ready     91m    File: bf-bundle-3.2.1-
34_25.11_ubuntu-24.04_64k_prod.bfb, DOCA: 3.2.1
          DPUs
            2 DPUs...                  dpf-operator-system
Ready: True  DPUReady  25m    See dpu-node-mt2402xz0f6v-
mt2402xz0f6v, dpu-node-mt2404xz0c98-mt2404xz0c98
      Services
        DPUServiceTemplates
          DPUServiceTemplate/doca-hbn  dpf-operator-system
Ready: True  Success   91m
        DPUServices
          DPUService/doca-hbn-6rhsx    dpf-operator-system
Ready: True  Success   21s
```

Test Traffic

After the DPUs are provisioned and the rest of the objects are Ready, we can test traffic by assigning an IP to the PF0 on the host for each DPU, and run a simple ping. Although the configuration is enabling both PFs, we focus on the PF0 for testing traffic. Assuming the PF0 is named `ens5f0np0`:

On the host with DPU with serial number DPU1_SERIAL:

```
ip link set dev ens5f0np0 up
ip addr add 10.0.121.1/29 dev ens5f0np0
ip route add 10.0.121.0/24 dev ens5f0np0 via 10.0.121.2
```

On the host with DPU with serial number DPU2_SERIAL:

```
ip link set dev ens5f0np0 up
ip addr add 10.0.121.9/29 dev ens5f0np0
ip route add 10.0.121.0/24 dev ens5f0np0 via 10.0.121.10
```

On the host with DPU with serial number DPU1_SERIAL:

```
$ ping 10.0.121.9 -c3
PING 10.0.121.9 (10.0.121.9) 56(84) bytes of data.
64 bytes from 10.0.121.9: icmp_seq=1 ttl=64 time=0.387 ms
64 bytes from 10.0.121.9: icmp_seq=2 ttl=64 time=0.344 ms
64 bytes from 10.0.121.9: icmp_seq=3 ttl=64 time=0.396 ms

--- 10.0.121.9 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2053ms
rtt min/avg/max/mdev = 0.344/0.375/0.396/0.022 ms
```

Using PFs + VFs

In this scenario, the PF0, PF1, VF10 of the PF0 and VF10 of the PF1 are connected to separate VRFs which means that:

- PF0 on Host 1 will be able to communicate with PF0 on Host 2
- PF0 on Host 1 will not be able to communicate with PF1 on Host 1 and 2
- PF0 on Host 1 will not be able to communicate with PF0VF10 on Host 1 and 2
- PF0 on Host 1 will not be able to communicate with PF1VF10 on Host 1 and 2
- PF1 on Host 1 will be able to communicate with PF1 on Host 2
- PF1 on Host 1 will not be able to communicate with PF0 on Host 1 and 2
- PF1 on Host 1 will not be able to communicate with PF0VF10 on Host 1 and 2
- PF1 on Host 1 will not be able to communicate with PF1VF10 on Host 1 and 2
- PF0VF10 on Host 1 will be able to communicate with PF0VF10 on Host 2
- PF0VF10 on Host 1 will not be able to communicate with PF0 on Host 1 and 2
- PF0VF10 on Host 1 will not be able to communicate with PF1 on Host 1 and 2
- PF0VF10 on Host 1 will not be able to communicate with PF1VF10 on Host 1 and 2
- PF1VF10 on Host 1 will be able to communicate with PF1VF10 on Host 2
- PF1VF10 on Host 1 will not be able to communicate with PF0 on Host 1 and 2
- PF1VF10 on Host 1 will not be able to communicate with PF1 on Host 1 and 2
- PF1VF10 on Host 1 will not be able to communicate with PF0VF10 on Host 1 and 2

We make use of a PF and a VF on the host to test traffic.

Create the DPUDeployment, DPUServiceConfig, DPUServiceTemplate and other necessary objects

A number of environment variables must be set before running this command.

```
cat manifests/03.2-dpudeployment-installation-pf-vf/*.yaml |
envsubst | kubectl apply -f -
```

This will deploy the following objects:

BFB to download Bluefield Bitstream to a shared volume

```
---
apiVersion: provisioning.dpu.nvidia.com/v1alpha1
kind: BFB
metadata:
  name: bf-bundle-$TAG
  namespace: dpf-operator-system
spec:
  url: $BFB_URL
```

HBN DPUFlavor to correctly configure the DPUs on provisioning

```
---
apiVersion: provisioning.dpu.nvidia.com/v1alpha1
kind: DPUFlavor
metadata:
  name: hbn-$TAG
  namespace: dpf-operator-system
spec:
  dpuMode: zero-trust
  bfcfgParameters:
  - UPDATE_ATF_UEFI=yes
  - UPDATE_DPU_OS=yes
  - WITH_NIC_FW_UPDATE=yes
  configFiles:
  - operation: override
    path: /etc/mellanox/mlnx-bf.conf
    permissions: "0644"
    raw: |
      ALLOW_SHARED_RQ="no"
      IPSEC_FULL_OFFLOAD="no"
      ENABLE_ESWITCH_MULTIPORT="yes"
  - operation: override
    path: /etc/mellanox/mlnx-ovs.conf
    permissions: "0644"
    raw: |
      CREATE_OVS_BRIDGES="no"
      OVS_DOCA="yes"
  - operation: override
    path: /etc/mellanox/mlnx-sf.conf
    permissions: "0644"
    raw: ""
  grub:
    kernelParameters:
    - console=hvc0
    - console=ttyAMA0
    - earlycon=pl011,0x13010000
```

```
    - fixrttc
    - net.ifnames=0
    - biosdevname=0
    - iommu.passthrough=1
    - cgroup_no_v1=net_prio,net_cls
    - hugepagesz=2048kB
    - hugepages=3072
  nvconfig:
  - device: '*'
    parameters:
    - PF_BAR2_ENABLE=0
    - PER_PF_NUM_SF=1
    - PF_TOTAL_SF=20
    - PF_SF_BAR_SIZE=10
    - NUM_PF_MSIX_VALID=0
    - PF_NUM_PF_MSIX_VALID=1
    - PF_NUM_PF_MSIX=228
    - INTERNAL_CPU_MODEL=1
    - INTERNAL_CPU_OFFLOAD_ENGINE=0
    - SRIOV_EN=1
    - NUM_OF_VFS=46
    - LAG_RESOURCE_ALLOCATION=1
    - LINK_TYPE_P1=ETH
    - LINK_TYPE_P2=ETH
  ovs:
    rawConfigScript: |
      _ovs-vsctl() {
        ovs-vsctl --no-wait --timeout 15 "$@"
      }

      _ovs-vsctl set Open_vSwitch . other_config:doca-init=true
      _ovs-vsctl set Open_vSwitch . other_config:dpdk-max-
memzones=50000
      _ovs-vsctl set Open_vSwitch . other_config:hw-offload=true
      _ovs-vsctl set Open_vSwitch . other_config:pmd-quiet-
idle=true
```

```
      _ovs-vsctl set Open_vSwitch . other_config:max-idle=20000
      _ovs-vsctl set Open_vSwitch . other_config:max-
revalidator=5000
      _ovs-vsctl --if-exists del-br ovsbr1
      _ovs-vsctl --if-exists del-br ovsbr2
      _ovs-vsctl --may-exist add-br br-sfc
      _ovs-vsctl set bridge br-sfc datapath_type=netdev
      _ovs-vsctl set bridge br-sfc fail_mode=secure
      _ovs-vsctl --may-exist add-port br-sfc p0
      _ovs-vsctl set Interface p0 type=dpdk
      _ovs-vsctl set Interface p0 mtu_request=9216
      _ovs-vsctl set Port p0 external_ids:dpf-type=physical
      _ovs-vsctl --may-exist add-br br-hbn
      _ovs-vsctl set bridge br-hbn datapath_type=netdev
      _ovs-vsctl set bridge br-hbn fail_mode=secure
```

DPUDeployment to provision DPUs on worker nodes

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUDeployment
metadata:
  name: hbn
  namespace: dpf-operator-system
spec:
  dpus:
    bfb: bf-bundle-$TAG
    flavor: hbn-$TAG
    nodeEffect:
      noEffect: true
    dpuSets:
    - nameSuffix: "dpuset1"
      nodeSelector:
        matchLabels:
          feature.node.kubernetes.io/dpu-enabled: "true"
  services:
    doca-hbn:
      serviceTemplate: doca-hbn
      serviceConfiguration: doca-hbn
  serviceChains:
    switches:
      - ports:
        - serviceInterface:
            matchLabels:
              interface: p0
        - service:
            name: doca-hbn
            interface: p0_if
      - ports:
        - serviceInterface:
            matchLabels:
              interface: p1
        - service:
```

```
            name: doca-hbn
            interface: p1_if
        - ports:
          - serviceInterface:
              matchLabels:
                interface: pf0hpf
          - service:
              name: doca-hbn
              interface: pf0hpf_if
        - ports:
          - serviceInterface:
              matchLabels:
                interface: pf1hpf
          - service:
              name: doca-hbn
              interface: pf1hpf_if
        - ports:
          - serviceInterface:
              matchLabels:
                interface: pf0vf10
          - service:
              name: doca-hbn
              interface: pf0vf10_if
        - ports:
          - serviceInterface:
              matchLabels:
                interface: pf1vf10
          - service:
              name: doca-hbn
              interface: pf1vf10_if
```

DPUServiceConfig and DPUServiceTemplate to deploy HBN workloads to the DPUs

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceConfiguration
metadata:
  name: doca-hbn
  namespace: dpf-operator-system
spec:
  deploymentServiceName: "doca-hbn"
  serviceConfiguration:
    serviceDaemonSet:
      annotations:
        k8s.v1.cni.cncf.io/networks: |-
          [
          {"name": "iprequest", "interface": "ip_lo", "cni-args": {"poolNames": ["loopback"], "poolType": "cidrpool"}},
          {"name": "iprequest", "interface": "ip_pf0hpf", "cni-args": {"poolNames": ["pool1"], "poolType": "cidrpool", "allocateDefaultGateway": true}},
          {"name": "iprequest", "interface": "ip_pf1hpf", "cni-args": {"poolNames": ["pool2"], "poolType": "cidrpool", "allocateDefaultGateway": true}},
          {"name": "iprequest", "interface": "ip_pf0vf10", "cni-args": {"poolNames": ["pool3"], "poolType": "cidrpool", "allocateDefaultGateway": true}},
          {"name": "iprequest", "interface": "ip_pf1vf10", "cni-args": {"poolNames": ["pool4"], "poolType": "cidrpool", "allocateDefaultGateway": true}}
          ]
    helmChart:
      values:
        configuration:
          perDPUValuesYAML: |
            - hostnamePattern: "*"
              values:
                bgp_peer_group: hbn
                vrf1: RED
                vrf2: BLUE
                vrf3: GREEN
                vrf4: YELLOW
```

```
                l3vni1: 100001
                l3vni2: 100002
                l3vni3: 100003
                l3vni4: 100004
            - hostnamePattern: "dpu-node-${DPU1_SERIAL}*"
              values:
                bgp_autonomous_system: 65101
            - hostnamePattern: "dpu-node-${DPU2_SERIAL}*"
              values:
                bgp_autonomous_system: 65201
          startupYAMLJ2: |
            - header:
                model: bluefield
                nvue-api-version: nvue_v1
                rev-id: 1.0
                version: HBN 2.4.0
            - set:
                evpn:
                  enable: on
                  route-advertise: {}
                interface:
                  lo:
                    ip:
                      address:
                        {{ ipaddresses.ip_lo.ip }}/32: {}
                    type: loopback

p0_if,p1_if,pf0vf10_if,pf1vf10_if,pf0hpf_if,pf1hpf_if:
                    type: swp
                    link:
                      mtu: 9000
                  pf0vf10_if:
                    ip:
                      address:
                        {{ ipaddresses.ip_pf0vf10.cidr }}: {}
                      vrf: {{ config.vrf1 }}
```

```
      pf1vf10_if:
        ip:
          address:
            {{ ipaddresses.ip_pf1vf10.cidr }}: {}
          vrf: {{ config.vrf2 }}
      pf0hpf_if:
        ip:
          address:
            {{ ipaddresses.ip_pf0hpf.cidr }}: {}
          vrf: {{ config.vrf3 }}
      pf1hpf_if:
        ip:
          address:
            {{ ipaddresses.ip_pf1hpf.cidr }}: {}
          vrf: {{ config.vrf4 }}
    nve:
      vxlan:
        arp-nd-suppress: on
        enable: on
        source:
          address: {{ ipaddresses.ip_lo.ip }}
    router:
      bgp:
        enable: on
        graceful-restart:
          mode: full
    vrf:
      default:
        router:
          bgp:
            address-family:
              ipv4-unicast:
                enable: on
                redistribute:
                  connected:
                    enable: on
```

```
                            multipaths:
                              ebgp: 16
                          l2vpn-evpn:
                            enable: on
                        autonomous-system: {{
config.bgp_autonomous_system }}
                        enable: on
                        neighbor:
                          p0_if:
                            peer-group: {{ config.bgp_peer_group
}}
                            type: unnumbered
                            address-family:
                              l2vpn-evpn:
                                enable: on
                                add-path-tx: off
                          p1_if:
                            peer-group: {{ config.bgp_peer_group
}}
                            type: unnumbered
                            address-family:
                              l2vpn-evpn:
                                enable: on
                                add-path-tx: off
                        path-selection:
                          multipath:
                            aspath-ignore: on
                        peer-group:
                          {{ config.bgp_peer_group }}:
                            address-family:
                              ipv4-unicast:
                                enable: on
                              l2vpn-evpn:
                                enable: on
                            remote-as: external
                        router-id: {{ ipaddresses.ip_lo.ip }}
```

```
                    {{ config.vrf1 }}:
                      evpn:
                        enable: on
                        vni:
                          {{ config.l3vni1 }}: {}
                      router:
                        bgp:
                          address-family:
                            ipv4-unicast:
                              enable: on
                              redistribute:
                                connected:
                                  enable: on
                              route-export:
                                to-evpn:
                                  enable: on
                          autonomous-system: {{
config.bgp_autonomous_system }}
                          enable: on
                    {{ config.vrf2 }}:
                      evpn:
                        enable: on
                        vni:
                          {{ config.l3vni2 }}: {}
                      router:
                        bgp:
                          address-family:
                            ipv4-unicast:
                              enable: on
                              redistribute:
                                connected:
                                  enable: on
                              route-export:
                                to-evpn:
                                  enable: on
```

```
                        autonomous-system: {{
config.bgp_autonomous_system }}
                        enable: on
                  {{ config.vrf3 }}:
                    evpn:
                      enable: on
                      vni:
                        {{ config.l3vni3 }}: {}
                    router:
                      bgp:
                        address-family:
                          ipv4-unicast:
                            enable: on
                            redistribute:
                              connected:
                                enable: on
                            route-export:
                              to-evpn:
                                enable: on
                        autonomous-system: {{
config.bgp_autonomous_system }}
                        enable: on
                  {{ config.vrf4 }}:
                    evpn:
                      enable: on
                      vni:
                        {{ config.l3vni4 }}: {}
                    router:
                      bgp:
                        address-family:
                          ipv4-unicast:
                            enable: on
                            redistribute:
                              connected:
                                enable: on
                            route-export:
```

```
                                to-evpn:
                                  enable: on
                          autonomous-system: {{
config.bgp_autonomous_system }}
                          enable: on

  interfaces:
  - name: p0_if
    network: mybrhbn
  - name: p1_if
    network: mybrhbn
  - name: pf0vf10_if
    network: mybrhbn
  - name: pf1vf10_if
    network: mybrhbn
  - name: pf0hpf_if
    network: mybrhbn
  - name: pf1hpf_if
    network: mybrhbn
```

```yaml
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceTemplate
metadata:
  name: doca-hbn
  namespace: dpf-operator-system
spec:
  deploymentServiceName: "doca-hbn"
  helmChart:
    source:
      repoURL: $HELM_REGISTRY_REPO_URL
      version: 1.0.5
      chart: doca-hbn
    values:
      image:
        repository: $HBN_NGC_IMAGE_URL
        tag: 3.2.1-doca3.2.1
      resources:
        memory: 6Gi
        nvidia.com/bf_sf: 6
```

DPUServiceInterfaces for physical ports on the DPU

```yaml
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceInterface
metadata:
  name: p0
  namespace: dpf-operator-system
spec:
  template:
    spec:
      template:
        metadata:
          labels:
            interface: "p0"
        spec:
          interfaceType: physical
          physical:
            interfaceName: p0
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceInterface
metadata:
  name: p1
  namespace: dpf-operator-system
spec:
  template:
    spec:
      template:
        metadata:
          labels:
            interface: "p1"
        spec:
          interfaceType: physical
          physical:
            interfaceName: p1
---
```

```yaml
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceInterface
metadata:
  name: pf0vf10-rep
  namespace: dpf-operator-system
spec:
  template:
    spec:
      template:
        metadata:
          labels:
            interface: "pf0vf10"
        spec:
          interfaceType: vf
          vf:
            parentInterfaceRef: p0
            pfID: 0
            vfID: 10
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceInterface
metadata:
  name: pf1vf10-rep
  namespace: dpf-operator-system
spec:
  template:
    spec:
      template:
        metadata:
          labels:
            interface: "pf1vf10"
        spec:
          interfaceType: vf
          vf:
            parentInterfaceRef: p1
            pfID: 1
```

```
            vfID: 10
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceInterface
metadata:
  name: pf0hpf
  namespace: dpf-operator-system
spec:
  template:
    spec:
      template:
        metadata:
          labels:
            interface: "pf0hpf"
        spec:
          interfaceType: pf
          pf:
            pfID: 0
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceInterface
metadata:
  name: pf1hpf
  namespace: dpf-operator-system
spec:
  template:
    spec:
      template:
        metadata:
          labels:
            interface: "pf1hpf"
        spec:
          interfaceType: pf
          pf:
            pfID: 1
```

DPUServiceIPAM to set up IP Address Management on the DPUCluster

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceIPAM
metadata:
  name: pool1
  namespace: dpf-operator-system
spec:
  ipv4Network:
    network: "10.0.121.0/24"
    gatewayIndex: 2
    prefixSize: 29
    # These preallocations are not necessary. We specify them so
that the validation commands are straightforward.
    allocations:
      dpu-node-${DPU1_SERIAL}-${DPU1_SERIAL}: 10.0.121.0/29
      dpu-node-${DPU2_SERIAL}-${DPU2_SERIAL}: 10.0.121.8/29
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceIPAM
metadata:
  name: pool2
  namespace: dpf-operator-system
spec:
  ipv4Network:
    network: "10.0.122.0/24"
    gatewayIndex: 2
    prefixSize: 29
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceIPAM
metadata:
  name: pool3
  namespace: dpf-operator-system
spec:
  ipv4Network:
```

```
    network: "10.0.123.0/24"
    gatewayIndex: 2
    prefixSize: 29
    # These preallocations are not necessary. We specify them so
that the validation commands are straightforward.
    allocations:
      dpu-node-${DPU1_SERIAL}-${DPU1_SERIAL}: 10.0.123.0/29
      dpu-node-${DPU2_SERIAL}-${DPU2_SERIAL}: 10.0.123.8/29
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceIPAM
metadata:
  name: pool4
  namespace: dpf-operator-system
spec:
  ipv4Network:
    network: "10.0.124.0/24"
    gatewayIndex: 2
    prefixSize: 29
```

DPUServiceIPAM for the loopback interface in HBN

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceIPAM
metadata:
  name: loopback
  namespace: dpf-operator-system
spec:
  ipv4Network:
    network: "11.0.0.0/24"
    prefixSize: 32
```

Verification

These verification commands may need to be run multiple times to ensure the condition is met.

Note that the DPUService name will have a random suffix. For example, `doca-hbn-l2xsl`.

Verify the DPU and Service installation with:

```
## Ensure the DPUServices are created and have been reconciled.
kubectl wait --for=condition=ApplicationsReconciled --namespace
dpf-operator-system dpuservices -l svc.dpu.nvidia.com/owned-by-
dpudeployment=dpf-operator-system_hbn
## Ensure the DPUServiceIPAMs have been reconciled
kubectl wait --for=condition=DPUIPAMObjectReconciled --namespace
dpf-operator-system dpuserviceipam --all
## Ensure the DPUServiceInterfaces have been reconciled
kubectl wait --for=condition=ServiceInterfaceSetReconciled --
namespace dpf-operator-system dpuserviceinterface --all
## Ensure the DPUServiceChains have been reconciled
kubectl wait --for=condition=ServiceChainSetReconciled --namespace
dpf-operator-system dpuservicechain --all
## Ensure the DPUs have the condition Initialized (this may take
time)
kubectl wait --for=condition=Initialized --namespace dpf-operator-
system dpu --all
```

or with `dpfctl`:

```
$ kubectl -n dpf-operator-system exec deploy/dpf-operator-
controller-manager -- /dpfctl describe dpudeployments
NAME                                                    NAMESPACE
STATUS       REASON         SINCE  MESSAGE
DPFOperatorConfig/dpfoperatorconfig                     dpf-operator-
system
              Ready
False        Pending        3m13s  The following conditions are
not ready:

* SystemComponentsReady
              SystemComponentsReady
False        Error          2m28s  System components must be
ready for DPF Operator to continue:

* nvidia-k8s-ipam: DPUService dpf-operator-system/nvidia-k8s-ipam
is not ready
  DPUDeployments
    DPUDeployment/hbn                                   dpf-operator-
system
                 Ready
False        Pending        77s    The following conditions are
not ready:

* DPUSetsReady
                 DPUSetsReady
False        Pending        79s    Objects are not ready:

* dpf-operator-system/hbn-dpuset1
      DPUServiceChains
        DPUServiceChain/hbn-5zgs4                       dpf-operator-
system  Ready: True  Success        79s
      DPUServiceInterfaces
        6 DPUServiceInterfaces...                       dpf-operator-
system  Ready: True  Success        79s    See doca-hbn-p0-if-
```

```
w6f6b, doca-hbn-p1-if-p7565, doca-hbn-pf0hpf-if-wb84j, doca-hbn-
pf0vf10-if-mr6fj,

doca-hbn-pf1hpf-if-cnbz8, doca-hbn-pf1vf10-if-7r6r6
      DPUSets
        DPUSet/hbn-dpuset1                          dpf-operator-
system
          BFB/bf-bundle                             dpf-operator-
system  Ready: True  Ready          105s   File: bf-bundle-3.2.1-
34_25.11_ubuntu-24.04_64k_prod.bfb, DOCA: 3.2.1
          DPUs
            DPU/dpu-node-mt2402xz0f6v-mt2402xz0f6v  dpf-operator-
system
                          Ready
False        OS Installing  72s
            DPU/dpu-node-mt2404xz0c98-mt2404xz0c98  dpf-operator-
system
                          Ready
False        OS Installing  69s
      Services
        DPUServiceTemplates
          DPUServiceTemplate/doca-hbn               dpf-operator-
system  Ready: True  Success        104s
        DPUServices
          DPUService/doca-hbn-bjqbh                 dpf-operator-
system  Ready: True  Success        77s
```

Making the DPUs Ready

In order to make the DPUs ready, we will need to manually power cycle the host. This operation should be done in the most graceful manner by gracefully shutting down the Host and DPU, powering off the server and then powering it on to avoid corruption. This should happen when the object gives us the signal. The described flow can be automated by the admin depending on the infrastructure.

The following verification command may need to be run multiple times to ensure the condition is met.

```
## Ensure the DPUs have the condition
WaitingForManualPowerCycleOrReboot (this may take time)
kubectl wait --for=condition=WaitingForManualPowerCycleOrReboot --
namespace dpf-operator-system dpu --all
```

or with `dpfctl`:

```
$ kubectl -n dpf-operator-system exec deploy/dpf-operator-
controller-manager -- /dpfctl describe dpudeployments
NAME                                                NAMESPACE
STATUS       REASON                                SINCE  MESSAGE
DPFOperatorConfig/dpfoperatorconfig                 dpf-operator-
system
              Ready
False        Pending                               17m    The
following conditions are not ready:

* SystemComponentsReady
              SystemComponentsReady
False        Error                                 16m    System
components must be ready for DPF Operator to continue:

* nvidia-k8s-ipam: DPUService dpf-operator-system/nvidia-k8s-ipam
is not ready
  DPUDeployments
    DPUDeployment/hbn                               dpf-operator-
system
                  Ready
False        Pending                               15m    The
following conditions are not ready:

* DPUSetsReady
                  DPUSetsReady
False        Pending                               15m    Objects
are not ready:

* dpf-operator-system/hbn-dpuset1
      DPUServiceChains
        DPUServiceChain/hbn-5zgs4                   dpf-operator-
system  Ready: True  Success                         15m
      DPUServiceInterfaces
```

```
        6 DPUServiceInterfaces...                          dpf-operator-
system  Ready: True  Success                                    15m
See doca-hbn-p0-if-w6f6b, doca-hbn-p1-if-p7565, doca-hbn-pf0hpf-if-
wb84j, doca-hbn-pf0vf10-if-mr6fj,

doca-hbn-pf1hpf-if-cnbz8, doca-hbn-pf1vf10-if-7r6r6
      DPUSets
        DPUSet/hbn-dpuset1                                 dpf-operator-
system
          BFB/bf-bundle                                    dpf-operator-
system  Ready: True  Ready                                      15m
File: bf-bundle-3.2.1-34_25.11_ubuntu-24.04_64k_prod.bfb, DOCA:
3.2.1
          DPUs
            DPU/dpu-node-mt2402xz0f6v-mt2402xz0f6v  dpf-operator-
system
                          Rebooted
False        WaitingForManualPowerCycleOrReboot  2m36s
                          Ready
False        Rebooting                           2m36s
            DPU/dpu-node-mt2404xz0c98-mt2404xz0c98  dpf-operator-
system
                          Rebooted
False        WaitingForManualPowerCycleOrReboot  2m36s
                          Ready
False        Rebooting                           2m36s
      Services
        DPUServiceTemplates
          DPUServiceTemplate/doca-hbn                      dpf-operator-
system  Ready: True  Success                                    15m
        DPUServices
          DPUService/doca-hbn-bjqbh                        dpf-operator-
system  Ready: True  Success                                    15m
```

At this point, we have to power cycle the hosts. Once **all the hosts are back online**, we have to remove an annotation from the DPUNodes. The user can choose to remove this annotation node by node but to make it simpler in this guide, we do that all at once.

```
kubectl annotate dpunodes -n dpf-operator-system --all
provisioning.dpu.nvidia.com/dpunode-external-reboot-required-
```

After this is done, we should expect that all DPUs become Ready:

```
kubectl wait --for="jsonpath={.status.phase}=Ready" --namespace dpf-operator-
system dpu --all
```

or with dpfctl:

```
$ kubectl -n dpf-operator-system exec deploy/dpf-operator-
controller-manager -- /dpfctl describe dpudeployments
NAME                                      NAMESPACE
STATUS       REASON    SINCE  MESSAGE
NAME                                      NAMESPACE
STATUS       REASON    SINCE  MESSAGE
DPFOperatorConfig/dpfoperatorconfig    dpf-operator-system
Ready: True  Success   6m5s
  DPUDeployments
    DPUDeployment/hbn                  dpf-operator-system
Ready: True  Success   2s
      DPUServiceChains
        DPUServiceChain/hbn-5zgs4      dpf-operator-system
Ready: True  Success   36s
      DPUServiceInterfaces
        6 DPUServiceInterfaces...      dpf-operator-system
Ready: True  Success   6s     See doca-hbn-p0-if-w6f6b, doca-hbn-
p1-if-p7565, doca-hbn-pf0hpf-if-wb84j, doca-hbn-pf0vf10-if-mr6fj,

doca-hbn-pf1hpf-if-cnbz8, doca-hbn-pf1vf10-if-7r6r6
      DPUSets
        DPUSet/hbn-dpuset1             dpf-operator-system
          BFB/bf-bundle                dpf-operator-system
Ready: True  Ready     28m    File: bf-bundle-3.2.1-
34_25.11_ubuntu-24.04_64k_prod.bfb, DOCA: 3.2.1
          DPUs
            2 DPUs...                  dpf-operator-system
Ready: True  DPUReady  5m52s  See dpu-node-mt2402xz0f6v-
mt2402xz0f6v, dpu-node-mt2404xz0c98-mt2404xz0c98
      Services
        DPUServiceTemplates
          DPUServiceTemplate/doca-hbn  dpf-operator-system
Ready: True  Success   28m
        DPUServices
```

```
          DPUService/doca-hbn-bjqbh    dpf-operator-system
Ready: True  Success   3s
```

Test Traffic

After the DPUs are provisioned and the rest of the objects are Ready, we can test traffic by assigning an IP to the PF0 on the host for each DPU, and run a simple ping. Although the configuration is enabling both PFs, we focus on the PF0 for testing traffic. Assuming the PF0 is named `ens5f0np0`:

On the host with DPU with serial number DPU1_SERIAL:

```
ip link set dev ens5f0np0 up
ip addr add 10.0.121.1/29 dev ens5f0np0
ip route add 10.0.121.0/24 dev ens5f0np0 via 10.0.121.2
```

On the host with DPU with serial number DPU2_SERIAL:

```
ip link set dev ens5f0np0 up
ip addr add 10.0.121.9/29 dev ens5f0np0
ip route add 10.0.121.0/24 dev ens5f0np0 via 10.0.121.10
```

On the host with DPU with serial number DPU1_SERIAL:

```
$ ping 10.0.121.9 -c3
PING 10.0.121.9 (10.0.121.9) 56(84) bytes of data.
64 bytes from 10.0.121.9: icmp_seq=1 ttl=64 time=0.387 ms
64 bytes from 10.0.121.9: icmp_seq=2 ttl=64 time=0.344 ms
64 bytes from 10.0.121.9: icmp_seq=3 ttl=64 time=0.396 ms

--- 10.0.121.9 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2053ms
rtt min/avg/max/mdev = 0.344/0.375/0.396/0.022 ms
```

In addition, we can test traffic by assigning an IP to the 10th VF of PF0 on the host for each DPU, and run a simple ping. We could use any VF, but the DPUDeployment and DPUServiceInterface will need to be adjusted accordingly. First thing to do is to create the VFs on the hosts where the each DPU belongs to:

```
echo 12 > /sys/class/net/ens5f0np0/device/sriov_numvfs
```

Then, assuming the VF is named `ens5f0v10`:

On the host with DPU with serial number DPU1_SERIAL:

```
ip link set dev ens5f0v10 up
ip addr add 10.0.123.1/29 dev ens5f0v10
ip route add 10.0.123.0/24 dev ens5f0v10 via 10.0.123.2
```

On the host with DPU with serial number DPU2_SERIAL:

```
ip link set dev ens5f0v10 up
ip addr add 10.0.123.9/29 dev ens5f0v10
ip route add 10.0.123.0/24 dev ens5f0v10 via 10.0.123.10
```

On the host with DPU with serial number DPU1_SERIAL:

```
$ ping 10.0.123.9 -c3
PING 10.0.123.9 (10.0.123.9) 56(84) bytes of data.
64 bytes from 10.0.123.9: icmp_seq=1 ttl=64 time=0.387 ms
64 bytes from 10.0.123.9: icmp_seq=2 ttl=64 time=0.344 ms
64 bytes from 10.0.123.9: icmp_seq=3 ttl=64 time=0.396 ms

--- 10.0.123.9 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2053ms
rtt min/avg/max/mdev = 0.344/0.375/0.396/0.022 ms
```

Uninstall

This section covers only the DPF related components and not the prerequisites as these must be managed by the admin.

Delete the DPF Operator system and DPF Operator

```
kubectl delete -n dpf-operator-system dpfoperatorconfig
dpfoperatorconfig --wait
helm uninstall -n dpf-operator-system dpf-operator --wait
```

Delete DPF Operator PVC

```
kubectl -n dpf-operator-system delete pvc bfb-pvc
kubectl delete pv bfb-pv
```

Note: there can be a race condition with deleting the underlying Kamaji cluster which runs the DPU cluster control plane in this guide. If that happens it may be necessary to remove finalizers manually from `DPUCluster` and `Datastore` objects.

OVN VPC Service Deployment Guide

> **Note**
>
> OVN VPC service is considered tech preview and is not recommended for production use.

This configuration provides instructions for deploying the NVIDIA DOCA Platform Framework (DPF) on high-performance, bare-metal infrastructure in Zero Trust mode, utilizing DPU BMC and Redfish. It focuses on provisioning NVIDIA® BlueField®-3 DPUs using DPF, Deploying VPC OVN Service and enabling hosts to communicate through an isolated VPC.

Prerequisites

This guide should be run by cloning the repo from github.com/NVIDIA/doca-platform and moving to the `docs/public/user-guides/zero-trust/use-cases/vpc` directory.

The system is set up as described in the prerequisites.

Software Prerequisites

Install the following tools on the machine where you will run the commands in this guide:

- kubectl
- helm
- envsubst

Network Prerequisites

Worker Nodes

- Only a single DPU uplink is used with this deployment (p0)
- All worker nodes are connected to the same L2 broadcast domain (VLAN) on the high-speed network

Installation Guide

Commands in this guide are run in the same directory that contains this readme.

> [NOTE!] This deployment guide assumes that two subnets are available in the network for use by the VPC service for tunneled and external traffic. It is possible to use a single subnet for both traffic types with minor modifications to the deployment manifests. Refer to OVN VPC Deployment for more information

0. Required Variables

The following variables are required. Sensible defaults are provided where possible, but many values will be specific to your target infrastructure.

Environment variables file

```
## IP Address for the Kubernetes API server of the target cluster
on which DPF is installed.
## This should never include a scheme or a port.
## e.g. 10.10.10.10
export TARGETCLUSTER_API_SERVER_HOST=

## Virtual IP used by the load balancer for the DPU Cluster. Must
be a reserved IP from the management subnet and not
## allocated by DHCP.
export DPUCLUSTER_VIP=

## Interface on which the DPUCluster load balancer will listen.
Should be the management interface of the control plane
## node.
export DPUCLUSTER_INTERFACE=

## IP address to the NFS server used as storage for the BFB.
export NFS_SERVER_IP=

## The DPF REGISTRY is the Helm repository URL where the DPF
Operator Chart resides.
## Usually this is the NVIDIA Helm NGC registry. For development
purposes, this can be set to a different repository.
export REGISTRY=https://helm.ngc.nvidia.com/nvidia/doca

## The repository URL for the NVIDIA Helm chart registry.
## Usually this is the NVIDIA Helm NGC registry. For development
purposes, this can be set to a different repository.
export HELM_REGISTRY_REPO_URL=https://helm.ngc.nvidia.com/nvidia/doca

## IP_RANGE_START and IP_RANGE_END
## These define the IP range for DPU discovery via Redfish/BMC
interfaces
## Example: If your DPUs have BMC IPs in range 192.168.1.100-110
## export IP_RANGE_START=192.168.1.100
```

```
## export IP_RANGE_END=192.168.1.110
export IP_RANGE_START=

export IP_RANGE_END=

# The password used for DPU BMC root login, must be the same for
all DPUs
export BMC_ROOT_PASSWORD=

## IP Address through which ovn-central service (exposed as
NodePort)
## is accessible. This can be a VIP or one of the control-plane
node IP
## in the host k8s cluster.
## This should never include a scheme or a port.
## e.g. 10.10.10.10
export
TARGETCLUSTER_OVN_CENTRAL_IP=${TARGETCLUSTER_API_SERVER_HOST}

## IP address range for VTEPs used by VPC OVN Service on the high
speed fabric.
## This is a CIDR in the form e.g. 20.20.0.0/16
export VTEP_CIDR=20.20.0.0/16

## The Gateway address of the VTEP subnet
## This is an IP in the form e.g. 20.20.0.1
export VTEP_GATEWAY=20.20.0.1

## IP address range for external network used by VPC OVN Service
on the high speed fabric.
## This is a CIDR in the form e.g. 30.30.0.0/16
export EXTERNAL_CIDR=30.30.0.0/16

## The Gateway address of the external subnet
## This is an IP in the form e.g. 30.30.0.1
export EXTERNAL_GATEWAY=30.30.0.1
```

```
## The DPF TAG is the version of the DPF components which will be
deployed in this guide.
export TAG=v25.10.0

## URL to the BFB used in the `bfb.yaml` and linked by the
DPUSet.
export BFB_URL="https://content.mellanox.com/BlueField/BFBs/Ubuntu24.04/bf-bundle-3.2.1-
34_25.11_ubuntu-24.04_64k_prod.bfb"
```

Modify the variables in `manifests/00-env-vars/envvars.env` to fit your environment, then source the file:

```
source manifests/00-env-vars/envvars.env
```

1. DPF Operator Installation

Create storage required by the DPF Operator

```
kubectl create ns dpf-operator-system
cat manifests/01-dpf-operator-installation/*.yaml | envsubst |
kubectl apply -f -
```

This deploys the following objects:

PersistentVolume and PersistentVolumeClaim for the provisioning controller

```
---
apiVersion: v1
kind: PersistentVolume
metadata:
  name: bfb-pv
spec:
  capacity:
    storage: 10Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteMany
  nfs:
    path: /mnt/dpf_share/bfb
    server: $NFS_SERVER_IP
  persistentVolumeReclaimPolicy: Delete
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: bfb-pvc
  namespace: dpf-operator-system
spec:
  accessModes:
  - ReadWriteMany
  resources:
    requests:
      storage: 10Gi
  volumeMode: Filesystem
  storageClassName: ""
```

Create DPU BMC shared password secret

In Zero Trust mode, provisioning DPUs requires authentication with Redfish. In order to do that, you must set the same root password to access the BMC for all DPUs DPF is going

to manage.

For more information on how to set the BMC root password refer to BlueField DPU Administrator Quick Start Guide

The password is provided to DPF by creating the following secret:

```
kubectl create secret generic -n dpf-operator-system bmc-shared-
password --from-literal=password=$BMC_ROOT_PASSWORD
```

Additional Dependencies

Before deploying the DPF Operator, ensure that Helm is properly configured according to the Helm prerequisites.

> ⚠ **Warning**
>
> This is a critical prerequisite step that must be completed for the DPF Operator to function properly.

Deploy the DPF Operator

HTTP Registry (default)

If the $REGISTRY is an HTTP Registry (default value) use this command:

```
helm repo add --force-update dpf-repository ${REGISTRY}
helm repo update
helm upgrade --install -n dpf-operator-system dpf-operator dpf-
repository/dpf-operator --version=$TAG
```

OCI Registry

For development purposes, if the $REGISTRY is an OCI Registry use this command:

```
helm upgrade --install -n dpf-operator-system dpf-operator
$REGISTRY/dpf-operator --version=$TAG
```

Verification

These verification commands may need to be run multiple times to ensure the condition is met.

Verify the DPF Operator installation with:

```
## Ensure the DPF Operator deployment is available.
kubectl rollout status deployment --namespace dpf-operator-system
dpf-operator-controller-manager
## Ensure all pods in the DPF Operator system are ready.
kubectl wait --for=condition=ready --namespace dpf-operator-system
pods --all
```

2. DPF System Installation

This section involves creating the DPF system components and some basic infrastructure required for a functioning DPF-enabled cluster.

Deploy the DPF System components

```
kubectl create ns dpu-cplane-tenant1
cat manifests/02-dpf-system-installation/*.yaml | envsubst |
kubectl apply -f -
```

This will create the following objects:

DPFOperatorConfig to install the DPF System components

```
---
apiVersion: operator.dpu.nvidia.com/v1alpha1
kind: DPFOperatorConfig
metadata:
  name: dpfoperatorconfig
  namespace: dpf-operator-system
spec:
  dpuDetector:
    disable: true
  provisioningController:
    bfbPVCName: "bfb-pvc"
    dmsTimeout: 900
    installInterface:
      installViaRedfish:
        # Set this to the IP of one of your control plane nodes +
8080 port
        bfbRegistryAddress: "$TARGETCLUSTER_API_SERVER_HOST:8080"
        skipDPUNodeDiscovery: false
  kamajiClusterManager:
    disable: false
```

DPUCluster to serve as Kubernetes control plane for DPU nodes

```
---
apiVersion: provisioning.dpu.nvidia.com/v1alpha1
kind: DPUCluster
metadata:
  name: dpu-cplane-tenant1
  namespace: dpu-cplane-tenant1
spec:
  type: kamaji
  maxNodes: 10
  clusterEndpoint:
    # deploy keepalived instances on the nodes that match the
given nodeSelector.
    keepalived:
      # interface on which keepalived will listen. Should be the oob
interface of the control plane node.
      interface: $DPUCLUSTER_INTERFACE
      # Virtual IP reserved for the DPU Cluster load balancer.
Must not be allocatable by DHCP.
      vip: $DPUCLUSTER_VIP
      # virtualRouterID must be in range [1,255], make sure the
given virtualRouterID does not duplicate with any existing
keepalived process running on the host
      virtualRouterID: 126
      nodeSelector:
        node-role.kubernetes.io/control-plane: ""
```

DPUDiscovery to discover DPUDevices or DPUNodes

```
---
apiVersion: provisioning.dpu.nvidia.com/v1alpha1
kind: DPUDiscovery
metadata:
  name: dpu-discovery
  namespace: dpf-operator-system
spec:
  ipRangeSpec:
    ipRange:
      startIP: $IP_RANGE_START
      endIP: $IP_RANGE_END
```

Verification

These verification commands may need to be run multiple times to ensure the condition is met.

Verify the DPF System with:

```
## Ensure the provisioning and DPUService controller manager
deployments are available.
kubectl rollout status deployment --namespace dpf-operator-system
dpf-provisioning-controller-manager dpuservice-controller-manager
## Ensure all other deployments in the DPF Operator system are
Available.
kubectl rollout status deployment --namespace dpf-operator-system
## Ensure bfb registry daemonset is available
kubectl rollout status daemonset --namespace dpf-operator-system
bfb-registry
## Ensure the DPUCluster is ready for nodes to join.
kubectl wait --for=condition=ready --namespace dpu-cplane-tenant1
dpucluster --all
```

3. Create BFB and DPUFlavor

Create a BFB and DPUFlavor to be used for the DPU provisioning process

```
cat manifests/03-bfb-and-flavor/* | envsubst | kubectl apply -f -
```

This will deploy the following objects:

OVN VPC DPUDeployment

```
---
apiVersion: provisioning.dpu.nvidia.com/v1alpha1
kind: BFB
metadata:
  name: bf-bundle-$TAG
  namespace: dpf-operator-system
spec:
  url: $BFB_URL
```

```
---
apiVersion: provisioning.dpu.nvidia.com/v1alpha1
kind: DPUFlavor
metadata:
  name: vpc-flavor-$TAG
  namespace: dpf-operator-system
spec:
  dpuMode: zero-trust
  bfcfgParameters:
  - UPDATE_ATF_UEFI=yes
  - UPDATE_DPU_OS=yes
  - WITH_NIC_FW_UPDATE=yes
  configFiles:
  - operation: override
    path: /etc/mellanox/mlnx-bf.conf
    permissions: "0644"
    raw: |
      ALLOW_SHARED_RQ="no"
      IPSEC_FULL_OFFLOAD="no"
      ENABLE_ESWITCH_MULTIPORT="yes"
  - operation: override
    path: /etc/mellanox/mlnx-ovs.conf
    permissions: "0644"
    raw: |
      CREATE_OVS_BRIDGES="no"
      OVS_DOCA="yes"
  - operation: override
    path: /etc/mellanox/mlnx-sf.conf
    permissions: "0644"
    raw: ""
  grub:
    kernelParameters:
    - console=hvc0
    - console=ttyAMA0
    - earlycon=pl011,0x13010000
```

```
    - fixrttc
    - net.ifnames=0
    - biosdevname=0
    - iommu.passthrough=1
    - cgroup_no_v1=net_prio,net_cls
    - hugepagesz=2048kB
    - hugepages=3072
  nvconfig:
  - device: '*'
    parameters:
    - PF_BAR2_ENABLE=0
    - PER_PF_NUM_SF=1
    - PF_TOTAL_SF=20
    - PF_SF_BAR_SIZE=10
    - NUM_PF_MSIX_VALID=0
    - PF_NUM_PF_MSIX_VALID=1
    - PF_NUM_PF_MSIX=228
    - INTERNAL_CPU_MODEL=1
    - INTERNAL_CPU_OFFLOAD_ENGINE=0
    - SRIOV_EN=1
    - NUM_OF_VFS=46
    - LAG_RESOURCE_ALLOCATION=1
    - LINK_TYPE_P1=ETH
    - LINK_TYPE_P2=ETH
  ovs:
    rawConfigScript: |
      _ovs-vsctl() {
        ovs-vsctl --no-wait --timeout 15 "$@"
      }

      _ovs-vsctl set Open_vSwitch . other_config:doca-init=true
      _ovs-vsctl set Open_vSwitch . other_config:dpdk-max-
memzones=50000
      _ovs-vsctl set Open_vSwitch . other_config:hw-offload=true
      _ovs-vsctl set Open_vSwitch . other_config:pmd-quiet-
idle=true
```

```
      _ovs-vsctl set Open_vSwitch . other_config:max-idle=20000
      _ovs-vsctl set Open_vSwitch . other_config:max-
revalidator=5000
      _ovs-vsctl --if-exists del-br ovsbr1
      _ovs-vsctl --if-exists del-br ovsbr2
      _ovs-vsctl --may-exist add-br br-sfc
      _ovs-vsctl set bridge br-sfc datapath_type=netdev
      _ovs-vsctl set bridge br-sfc fail_mode=secure
      _ovs-vsctl --may-exist add-port br-sfc p0
      _ovs-vsctl set Interface p0 type=dpdk
      _ovs-vsctl set Interface p0 mtu_request=9216
      _ovs-vsctl set Port p0 external_ids:dpf-type=physical
```

4. OVN VPC Deployment

The OVN VPC service consists of the following components:

1. **ovn-central**: Deployed in the target cluster (runs northd, sb_db, nb_db)
2. **ovn-controller**: Deployed in the DPU cluster
3. **vpc-ovn-controller**: VPC controller in the target cluster
4. **vpc-ovn-node**: VPC node agent in the DPU cluster

Deploy OVN VPC DPUDeployment

> ⚠ **Warning**
>
> In case more than 1 DPU exists per node, the relevant selector should be applied in the DPUDeployment to select the appropriate DPU. See DPUDeployment - DPUs Configuration to understand more about the selectors.

```
cat manifests/04-vpc-ovn-dpudeployment/* | envsubst | kubectl
apply -f -
```

This will deploy the following objects:

OVN VPC DPUDeployment

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUDeployment
metadata:
  name: vpc-ovn
  namespace: dpf-operator-system
spec:
  dpus:
    bfb: bf-bundle-$TAG
    flavor: vpc-flavor-$TAG
    nodeEffect:
      noEffect: true
    dpuSets:
    - nameSuffix: "dpuset1"
      nodeSelector:
        matchLabels:
          feature.node.kubernetes.io/dpu-enabled: "true"
  services:
    ovn-central:
      serviceTemplate: ovn-central
      serviceConfiguration: ovn-central
    ovn-controller:
      serviceTemplate: ovn-controller
      serviceConfiguration: ovn-controller
    vpc-ovn-controller:
      serviceTemplate: vpc-ovn-controller
      serviceConfiguration: vpc-ovn-controller
    vpc-ovn-node:
      serviceTemplate: vpc-ovn-node
      serviceConfiguration: vpc-ovn-node
  serviceChains:
    switches:
      - ports:
        - serviceInterface:
            matchLabels:
```

```
            ovn.vpc.dpu.nvidia.com/interface: p0
        - serviceInterface:
            matchLabels:
              ovn.vpc.dpu.nvidia.com/interface: ovn-ext-patch
```

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceConfiguration
metadata:
  name: ovn-central
  namespace: dpf-operator-system
spec:
  deploymentServiceName: ovn-central
  upgradePolicy:
    applyNodeEffect: false
  serviceConfiguration:
    deployInCluster: true
    helmChart:
      values:
        exposedPorts:
          ports:
            ovnnb: true
            ovnsb: true
        management:
          ovnCentral:
            enabled: true
            affinity:
              nodeAffinity:
                requiredDuringSchedulingIgnoredDuringExecution:
                  nodeSelectorTerms:
                    - matchExpressions:
                        - key: "node-role.kubernetes.io/master"
                          operator: Exists
                    - matchExpressions:
                        - key: "node-role.kubernetes.io/control-plane"
                          operator: Exists
            tolerations:
              - key: node-role.kubernetes.io/master
                operator: Exists
                effect: NoSchedule
```

```
              - key: node-role.kubernetes.io/control-plane
                operator: Exists
                effect: NoSchedule
```

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceConfiguration
metadata:
  name: ovn-controller
  namespace: dpf-operator-system
spec:
  deploymentServiceName: ovn-controller
  upgradePolicy:
    applyNodeEffect: false
  serviceConfiguration:
    helmChart:
      values:
        dpu:
          ovnController:
            enabled: true
```

```yaml
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceConfiguration
metadata:
  name: vpc-ovn-controller
  namespace: dpf-operator-system
spec:
  deploymentServiceName: vpc-ovn-controller
  upgradePolicy:
    applyNodeEffect: false
  serviceConfiguration:
    deployInCluster: true
    helmChart:
      values:
        host:
          vpcOVNController:
            enabled: true
            affinity:
              nodeAffinity:
                requiredDuringSchedulingIgnoredDuringExecution:
                  nodeSelectorTerms:
                  - matchExpressions:
                    - key: "node-role.kubernetes.io/master"
                      operator: Exists
                  - matchExpressions:
                    - key: "node-role.kubernetes.io/control-plane"
                      operator: Exists
```

```yaml
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceConfiguration
metadata:
  name: vpc-ovn-node
  namespace: dpf-operator-system
spec:
  deploymentServiceName: vpc-ovn-node
  upgradePolicy:
    applyNodeEffect: false
  serviceConfiguration:
    helmChart:
      values:
        dpu:
          vpcOVNNode:
            enabled: true
            initContainers:
              vpcOVNDpuProvisioner:
                env:
                  ovnSbEndpoint: "tcp:$TARGETCLUSTER_OVN_CENTRAL_IP:30642"
            ipRequests:
              - name: "vtep"
                poolName: "vpc-ippool-vtep"
                allocateIPWithIndex: 1
              - name: "gateway"
                poolName: "vpc-ippool-gateway"
                allocateIPWithIndex: 1
```

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceTemplate
metadata:
  name: ovn-central
  namespace: dpf-operator-system
spec:
  deploymentServiceName: ovn-central
  helmChart:
    source:
      repoURL: $HELM_REGISTRY_REPO_URL
      version: $TAG
      chart: ovn-chart
```

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceTemplate
metadata:
  name: ovn-controller
  namespace: dpf-operator-system
spec:
  deploymentServiceName: ovn-controller
  helmChart:
    source:
      repoURL: $HELM_REGISTRY_REPO_URL
      version: $TAG
      chart: ovn-chart
```

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceTemplate
metadata:
  name: vpc-ovn-controller
  namespace: dpf-operator-system
spec:
  deploymentServiceName: vpc-ovn-controller
  helmChart:
    source:
      repoURL: $HELM_REGISTRY_REPO_URL
      version: $TAG
      chart: dpf-vpc-ovn
```

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceTemplate
metadata:
  name: vpc-ovn-node
  namespace: dpf-operator-system
spec:
  deploymentServiceName: vpc-ovn-node
  helmChart:
    source:
      repoURL: $HELM_REGISTRY_REPO_URL
      version: $TAG
      chart: dpf-vpc-ovn
```

```yaml
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceIPAM
metadata:
  name: vpc-ippool-vtep
  namespace: dpf-operator-system
spec:
  metadata:
    labels:
      ovn.vpc.dpu.nvidia.com/pool: vpc-ippool-vtep
  ipv4Subnet:
    subnet: $VTEP_CIDR
    gateway: $VTEP_GATEWAY
    perNodeIPCount: 4
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceIPAM
metadata:
  name: vpc-ippool-gateway
  namespace: dpf-operator-system
spec:
  metadata:
    labels:
      ovn.vpc.dpu.nvidia.com/pool: vpc-ippool-gateway
  ipv4Subnet:
    subnet: $EXTERNAL_CIDR
    gateway: $EXTERNAL_GATEWAY
    perNodeIPCount: 4
```

```
---
apiVersion: "svc.dpu.nvidia.com/v1alpha1"
kind: DPUServiceInterface
metadata:
  name: p0
  namespace: dpf-operator-system
spec:
  template:
    spec:
      template:
        metadata:
          labels:
            ovn.vpc.dpu.nvidia.com/interface: "p0"
        spec:
          interfaceType: physical
          physical:
            interfaceName: p0
---
apiVersion: "svc.dpu.nvidia.com/v1alpha1"
kind: DPUServiceInterface
metadata:
  name: ovn-ext-patch
  namespace: dpf-operator-system
spec:
  template:
    spec:
      template:
        metadata:
          labels:
            ovn.vpc.dpu.nvidia.com/interface: "ovn-ext-patch"
        spec:
          interfaceType: ovn
          ovn:
```

```
            externalBridge: br-ovn-ext
```

[NOTE!] In the above deployment we assume separate network subnets for VTEP(tunneled) and external networks. In case its desirable to use only a single network subnet for both traffic types (tunneled, external), simply modify `vpc-ovn-node` DPUServiceConfiguration to reference the same IP pool under `ipRequests` field.

Example:

```
yaml ipRequests: - name: "vtep" poolName: "vpc-ippool-vtep"
allocateIPWithIndex: 1 - name: "gateway" poolName: "vpc-
ippool-vtep" allocateIPWithIndex: 2
```

Make DPUs Ready

In order to make the DPUs ready, we will need to manually power cycle the host. This operation should be done in the most graceful manner by gracefully shutting down the Host and DPU, powering off the server and then powering it on to avoid corruption. This should happen when the DPU object gives us the signal. The described flow can be automated by the administrator depending on the infrastructure.

The following verification commands may need to be run multiple times to ensure the condition is met.

1. Wait for DPU `OSInstalled` condition to become `ready`

```
kubectl wait --for=condition=OSInstalled --namespace dpf-operator-
system dpu --all
```

2. Ensure `Rebooted` condition type has `reason=WaitingForManualPowerCycleOrReboot`

```
kubectl wait --namespace dpf-operator-system dpu --all --
for=jsonpath='{.status.conditions[?
(@.type=="Rebooted")].reason}'=WaitingForManualPowerCycleOrReboot
```

3. Power cycle DPU worker hosts - manual operation by the user

4. Once all nodes have rebooted, remove `provisioning.dpu.nvidia.com/dpunode-external-reboot-required` annotation from `DPUNodes`

```
kubectl -n dpf-operator-system annotate dpunode --all
provisioning.dpu.nvidia.com/dpunode-external-reboot-required-
```

5. Ensure `DPUs` are ready

```
kubectl wait --for=condition=ready --namespace dpf-operator-system
dpus --all
```

Validate deployed DPUServices

You may need to run these verification commands multiple times until the condition is met.

```
kubectl wait --for=condition=ready --namespace dpf-operator-system
dpudeployment vpc-ovn
```

or with `dpfctl`:

```
$ kubectl -n dpf-operator-system exec deploy/dpf-operator-
controller-manager -- /dpfctl describe dpudeployments
NAME                                      NAMESPACE
STATUS        REASON    SINCE  MESSAGE
DPFOperatorConfig/dpfoperatorconfig    dpf-operator-system
Ready: True   Success   11m
  DPUDeployments
    DPUDeployment/vpc-ovn              dpf-operator-system
Ready: True   Success   24m
      DPUServiceChains
        DPUServiceChain/vpc-ovn-tjktv  dpf-operator-system
Ready: True   Success   57m
      DPUServices
        4 DPUServices...               dpf-operator-system
Ready: True   Success   55m    See ovn-central-fdjg9, ovn-
controller-bj85w, vpc-ovn-controller-f8qgn, vpc-ovn-node-7bhd8
      DPUSets
        DPUSet/vpc-ovn-dpuset1         dpf-operator-system
          BFB/bf-bundle                dpf-operator-system
Ready: True   Ready     58m    File: bf-bundle-3.2.1-
34_25.11_ubuntu-24.04_64k_prod.bfb, DOCA: 3.2.1
          DPU/worker1-0000-c8-00       dpf-operator-system  Ready:
True   DPUReady  2m13s
          DPU/worker2-0000-c8-00       dpf-operator-system  Ready:
True   DPUReady  2m30s
```

5. Additional VPC Resources Deployment

In this step, you will deploy the `IsolationClass` resource, which will be used by subsequent user-created `DPUVPC` and `DPUVirtualNetwork` resources.

Deploy IsolationClass

```
cat manifests/05-vpc-resources/* | envsubst | kubectl apply -f -
```

This will deploy the following objects:

Additional VPC Resources

```
---
apiVersion: vpc.dpu.nvidia.com/v1alpha1
kind: IsolationClass
metadata:
  name: ovn.vpc.dpu.nvidia.com
spec:
  provisioner: ovn.vpc.dpu.nvidia.com
  parameters:
    ovn-nb-endpoint: "tcp:$TARGETCLUSTER_OVN_CENTRAL_IP:30641"
    ovn-sb-endpoint: "tcp:$TARGETCLUSTER_OVN_CENTRAL_IP:30642"
```

6. Optional - Test Traffic

At this point, your cluster should be set up and ready with all VPC components.

In this section we will demonstrate how to connect a host to VPC in two ways.

1. Using Host PFs (The DPU's host facing PCI physical functions)
2. Using Host PFs and VFs (The DPU's host facing PCI physical and virtual functions)

1. Using Host PFs

In this step, we will deploy the following VPC objects:

- One `DPUVPC` named `myvpc`

- One `DPUVirtualNetwork` named `pfnet` in `myvpc` VPC
- One `DPUServiceInterface` of type `PF`, referencing `pfnet` virtual network.
- for DPU PF 0
- spanning all worker nodes

Outcome: Hosts will be able to get DHCP from VPC on DPU PF 0 and communicate with each other and external networks.

Ensure you have SSH access to your worker hosts from the management or out-of-band (OOB) network.

Deploy test topology

```
cat manifests/06-optional-test-traffic/vpc-topology-pf-only.yaml
| envsubst | kubectl apply -f -
```

This will deploy the following objects:

VPC Test Topology

```yaml
---
apiVersion: vpc.dpu.nvidia.com/v1alpha1
kind: DPUVPC
metadata:
  name: myvpc
  namespace: default
spec:
  tenant: foo
  isolationClassName: ovn.vpc.dpu.nvidia.com
  interNetworkAccess: false
  nodeSelector: {}
---
apiVersion: vpc.dpu.nvidia.com/v1alpha1
kind: DPUVirtualNetwork
metadata:
  name: pfnet
  namespace: default
spec:
  vpcName: myvpc
  type: Bridged
  externallyRouted: true
  masquerade: true
  bridgedNetwork:
    ipam:
      ipv4:
        dhcp: true
        subnet: 10.100.0.0/16
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceInterface
metadata:
  name: pf0
  namespace: default
spec:
  template:
```

```
    spec:
      template:
        spec:
          interfaceType: pf
          pf:
            pfID: 0
            virtualNetwork: pfnet
```

Validate deployed resources

```
kubectl wait --for=condition=ready dpuvpc myvpc
kubectl wait --for=condition=ready dpuvirtualnetwork pfnet
kubectl wait --for=condition=ready dpuserviceinterface pf0
```

Test traffic between hosts

- SSH into each node and run `dhclient` for the network device associated with PF index 0 to obtain a DHCP address.

An example output for a node named `node1` and PF 0 network interface `enp8s0f0`:

```
root@node1:~# ip link set enp8s0f0 up
root@node1:~# dhclient -1 -v enp8s0f0
Internet Systems Consortium DHCP Client 4.4.3-P1
Copyright 2004-2022 Internet Systems Consortium.
All rights reserved.
For info, please visit https://www.isc.org/software/dhcp/

Listening on LPF/enp8s0f0/26:3a:60:48:81:cf
Sending on   LPF/enp8s0f0/26:3a:60:48:81:cf
Sending on   Socket/fallback
DHCPREQUEST for 10.100.0.2 on enp8s0f0 to 255.255.255.255 port 67
(xid=0x7cbe87ca)
DHCPACK of 10.100.0.2 from 10.100.0.1 (xid=0xca87be7c)
bound to 10.100.0.2 -- renewal in 1367 seconds.
```

Repeat this process on another node.

- Test connectivity by running traffic between nodes.

In the example below, the other node's PF 0 network interface was assigned the IP 10.100.0.3:

```
root@node1:~# ping 10.100.0.3
```

2. Using Host PFs and VFs

In this step, we will deploy the following VPC objects:

- One `DPUVPC` named `myvpc`
- One `DPUVirtualNetwork` named `pfnet` in `myvpc` VPC
- One `DPUVirtualNetwork` named `vfnet` in `myvpc` VPC

- One `DPUServiceInterface` of type `PF`, referencing `pfnet` virtual network.
- for DPU PF 0
- spanning all worker nodes
- Two `DPUServiceInterface` of type `VF`, referencing `vfnet` virtual network.
- for VF indexes 0,1 of PF 0
- spanning all worker nodes

Outcome: Hosts will be able to get DHCP from VPC on the configured DPU PFs and VFs and communicate in the following manner:

1. PFs can communicate with other PFs
2. VFs can communicate with other VFs
3. PFs cannot communicate with VFs
4. PFs and VFs can access external network

Ensure you have SSH access to your worker hosts from the management or out-of-band (OOB) network.

Create SR-IOV virtual functions for each DPU

Login to each host and create SR-IOV virtual functions(VFs)

Example for creating VFs on node1, do the same on the other node. the DPU is assumed to have PCI address of `0000:08:00.0`

```
root@node1:~# echo 2 > /sys/bus/pci/devices/0000:08:00.0/sriov_numvfs
```

Deploy test topology

```
cat manifests/06-optional-test-traffic/* | envsubst | kubectl
apply -f -
```

This will deploy the following objects:

VPC Test Topology

```
---
apiVersion: vpc.dpu.nvidia.com/v1alpha1
kind: DPUVPC
metadata:
  name: myvpc
  namespace: default
spec:
  tenant: foo
  isolationClassName: ovn.vpc.dpu.nvidia.com
  interNetworkAccess: false
  nodeSelector: {}
---
apiVersion: vpc.dpu.nvidia.com/v1alpha1
kind: DPUVirtualNetwork
metadata:
  name: pfnet
  namespace: default
spec:
  vpcName: myvpc
  type: Bridged
  externallyRouted: true
  masquerade: true
  bridgedNetwork:
    ipam:
      ipv4:
        dhcp: true
        subnet: 10.100.0.0/16
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceInterface
metadata:
  name: pf0
  namespace: default
spec:
  template:
```

```
  spec:
    template:
      spec:
        interfaceType: pf
        pf:
          pfID: 0
          virtualNetwork: pfnet
```

```yaml
---
apiVersion: vpc.dpu.nvidia.com/v1alpha1
kind: DPUVirtualNetwork
metadata:
  name: vfnet
  namespace: default
spec:
  vpcName: myvpc
  type: Bridged
  externallyRouted: true
  masquerade: true
  bridgedNetwork:
    ipam:
      ipv4:
        dhcp: true
        subnet: 10.200.0.0/16
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceInterface
metadata:
  name: vf0
  namespace: default
spec:
  template:
    spec:
      template:
        spec:
          interfaceType: vf
          vf:
            pfID: 0
            vfID: 0
            virtualNetwork: vfnet
            parentInterfaceRef: ""
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
```

```
kind: DPUServiceInterface
metadata:
  name: vf1
  namespace: default
spec:
  template:
    spec:
      template:
        spec:
          interfaceType: vf
          vf:
            pfID: 0
            vfID: 1
            virtualNetwork: vfnet
            parentInterfaceRef: ""
```

Validate deployed resources

```
kubectl wait --for=condition=ready dpuvpc myvpc
kubectl wait --for=condition=ready dpuvirtualnetwork pfnet vfnet
kubectl wait --for=condition=ready dpuserviceinterface pf0 vf0 vf1
```

Test traffic between hosts

In this section we will demonstrate how to request DHCP for a VF interfaces and run basic traffic between VFs on different hosts.

To do the same for PF interfaces refer to Test traffic between hosts of the previous section.

- SSH into each node and run `dhclient` for the network device associated with VF index 0 to obtain a DHCP address.

An example output for a node named `node1` and VF 0 network interface `enp8s0f0`:

```
# send dhcp request
root@node1:~# ip link set enp8s0f0v0 up
root@node1:~# dhclient -1 -v enp8s0f0v0
Internet Systems Consortium DHCP Client 4.4.3-P1
Copyright 2004-2022 Internet Systems Consortium.
All rights reserved.
For info, please visit https://www.isc.org/software/dhcp/

Listening on LPF/enp8s0f0v0/26:3a:60:48:81:cf
Sending on   LPF/enp8s0f0v0/26:3a:60:48:81:cf
Sending on   Socket/fallback
DHCPREQUEST for 10.200.0.2 on enp8s0f0v0 to 255.255.255.255 port 67
(xid=0x7cbe87ca)
DHCPACK of 10.200.0.2 from 10.200.0.1 (xid=0xca87be7c)
bound to 10.200.0.2 -- renewal in 1367 seconds.
```

Repeat this process for the second VF on this node and on another node.

- Test connectivity by running traffic between nodes.

In the example below, the other node's VF 0 network interface was assigned the IP 10.200.0.3:

```
root@node1:~# ping 10.200.0.3
```

Uninstall

This section covers only the DPF related components and not the prerequisites as these must be managed by the administrator.

1. Remove VPC Resources from the Cluster

```
cat manifests/06-optional-test-traffic/* | kubectl delete --wait
-f -
cat manifests/05-vpc-resources/* | kubectl delete --wait -f -
```

2. Remove DPF System and Operator Installation

```
kubectl delete -n dpf-operator-system dpfoperatorconfig
dpfoperatorconfig --wait
helm uninstall -n dpf-operator-system dpf-operator --wait
```

3. Delete DPF Operator PVC

```
kubectl -n dpf-operator-system delete pvc bfb-pvc
kubectl delete pv bfb-pv
```

Note

There can be a race condition with deleting the underlying Kamaji cluster which runs the DPU cluster control plane in this guide. If that happens it may be necessary to remove finalizers manually from `DPUCluster` and `Datastore` objects.

Notice

This document is provided for information purposes only and shall not be regarded as a warranty of a certain functionality, condition, or quality of a product. NVIDIA Corporation ("NVIDIA") makes no representations or warranties, expressed or implied, as to the accuracy or completeness of the information contained in this document and assumes no responsibility for any errors contained herein. NVIDIA shall have no liability for the consequences or use of such information or for any infringement of patents or other rights of third parties that may result from its use. This document is not a commitment to develop, release, or deliver any Material (defined below), code, or functionality.

NVIDIA reserves the right to make corrections, modifications, enhancements, improvements, and any other changes to this document, at any time without notice.

Customer should obtain the latest relevant information before placing orders and should verify that such information is current and complete.

NVIDIA products are sold subject to the NVIDIA standard terms and conditions of sale supplied at the time of order acknowledgement, unless otherwise agreed in an individual sales agreement signed by authorized representatives of NVIDIA and customer ("Terms of Sale"). NVIDIA hereby expressly objects to applying any customer general terms and conditions with regards to the purchase of the NVIDIA product referenced in this document. No contractual obligations are formed either directly or indirectly by this document.

NVIDIA products are not designed, authorized, or warranted to be suitable for use in medical, military, aircraft, space, or life support equipment, nor in applications where failure or malfunction of the NVIDIA product can reasonably be expected to result in personal injury, death, or property or environmental damage. NVIDIA accepts no liability for inclusion and/or use of NVIDIA products in such equipment or applications and therefore such inclusion and/or use is at customer's own risk.

NVIDIA makes no representation or warranty that products based on this document will be suitable for any specified use. Testing of all parameters of each product is not necessarily performed by NVIDIA. It is customer's sole responsibility to evaluate and determine the applicability of any information contained in this document, ensure the product is suitable and fit for the application planned by customer, and perform the necessary testing for the application in order to avoid a default of the application or the product. Weaknesses in customer's product designs may affect the quality and reliability of the NVIDIA product and may result in additional or different conditions and/or requirements beyond those contained in this document. NVIDIA accepts no liability related to any default, damage, costs, or problem which may be based on or attributable to: (i) the use of the NVIDIA product in any manner that is contrary to this document or (ii) customer product designs.

No license, either expressed or implied, is granted under any NVIDIA patent right, copyright, or other NVIDIA intellectual property right under this document. Information published by NVIDIA regarding third-party products or services does not constitute a license from NVIDIA to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property rights of the third party, or a license from NVIDIA under the patents or other intellectual property rights of NVIDIA.

Reproduction of information in this document is permissible only if approved in advance by NVIDIA in writing, reproduced without alteration and in full compliance with all applicable export laws and regulations, and accompanied by all associated conditions, limitations, and notices.

THIS DOCUMENT AND ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, "MATERIALS") ARE BEING PROVIDED "AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE. TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL NVIDIA BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF

ANY USE OF THIS DOCUMENT, EVEN IF NVIDIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. Notwithstanding any damages that customer might incur for any reason whatsoever, NVIDIA's aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms of Sale for the product.

Trademarks

NVIDIA and the NVIDIA logo are trademarks and/or registered trademarks of NVIDIA Corporation in the U.S. and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

© Copyright 2025, NVIDIA. PDF Generated on 12/23/2025