NVIDIA BlueField DPU Container Deployment Guide

User Guide

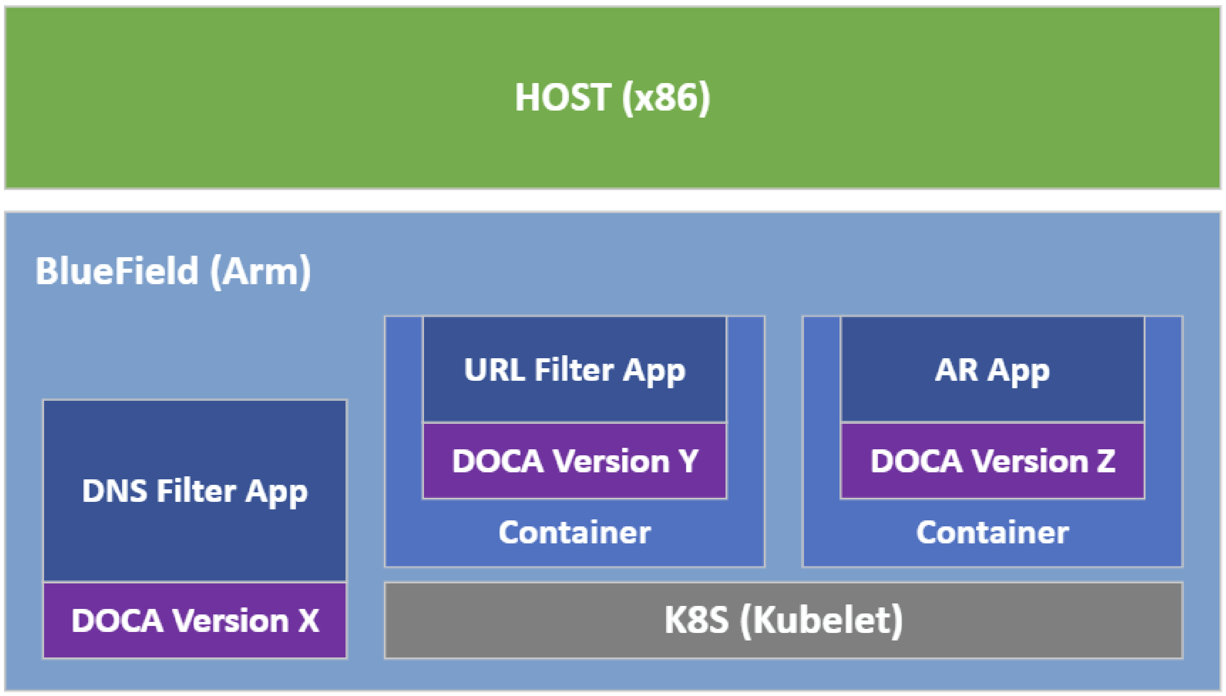
MLNX-15-060500 _v1.2 | January 2022

Table of Contents

Chapter 1. Introduction

DOCA containers allow for easy deployment of ready-made DOCA environments to the DPU, whether it is a DOCA application bundled inside a container and ready to be deployed, or a development environment already containing the desired DOCA version.

Containerized environments enable the users to decouple DOCA applications from the underlying BlueField OS. Each container is pre-built with all needed libraries and configurations to match the specific DOCA version of the application at hand. One only needs to pick the desired version of the application and pull the ready-made container of that version from NVIDIA's repository.



The different DOCA containers are listed on NGC, NVIDIA's container catalog, and can be found under both the "DOCA" and "DPU" labels.

Chapter 2. Prerequisites

- Refer to the DOCA Installation Guide for details on how to install BlueField related software
- BlueField OS version required is 3.8.0 and higher (Ubuntu 20.04)

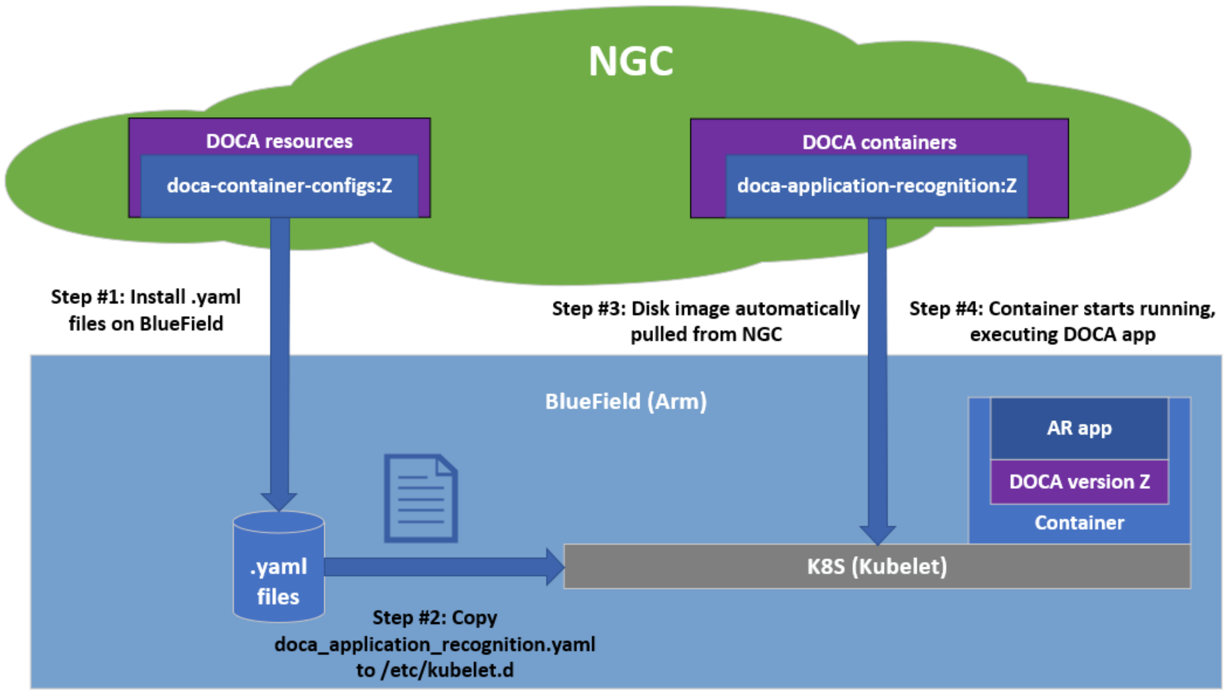
Chapter 3. Container Configuration

Deploying containers on top of the BlueField DPU requires the following setup sequence:

1. NGC Sign-in.
2. Pull the container .yaml configuration files.
3. Modify the container's .yaml configuration file.
4. Deploy container. Image is automatically pulled from NGC.

Some of the steps only need to be performed once, while others are required before the deployment of each container.

What follows is an example of the overall setup sequence using the DOCA application recognition (AR) container as an example.


NGC
DOCA resources
doca-container-configs:Z
DOCA containers
doca-application-recognition:Z
Step #1: Install .yaml files on BlueField
Step #3: Disk image automatically pulled from NGC
Step #4: Container starts running, executing DOCA app
BlueField (Arm)
AR app
DOCA version Z
Container
.yaml files
K8S (Kubelet)
Step #2: Copy doca_application_recognition.yaml to /etc/kubelet.d

3.1. Configure NGC Credentials on BlueField

Containers are deployed on the BlueField using Kubernetes (K8S) using the standalone Kubelet service. While Kubelet can automatically pull the container image directly from NGC, you must first supply it with login credentials.

This step needs to only be performed once per DPU to allow K8S to access NGC.

1. Create an NGC user accoding to the NGC Guide.

   Note: The privileges of the NGC user required for pulling the container images from NGC to the DPU, are the **minimal** privileges of an NGC user. When used in production environments, it is **highly recommended** to create a dedicated NGC user to **only** be used by K8S for deploying the containers on the BlueField.

   This reduces the risk for possible exposure of the API key as it only grants pull privileges from public NGC content and cannot be used for read/write access to any sensitive/privileged content.

2. Generate an API key for your user accoding to the NGC Guide.
3. Log into NGC via docker.

```
# Start docker’s service
systemctl start docker
# Login to NGC via docker (using the user’s API Key)
docker login nvcr.io
username: $oauthtoken # Yes, "$oauthtoken" is the username that should be used
password: <Your-API-Key>
```

   The expected output should roughly include the following:

```
"
WARNING! Your password will be stored unencrypted in /root/.docker/config.json.
…
Login Succeeded
"
```

4. Extract the auth token from docker's file. In this example it is located at `/root/.docker/config.json` as shown above.

```
# The config file should look roughly like this:
"
{
    "auths": {
        "nvcr.io": {
            "auth": "<long hexa-decimal string. This is the token we need.>"
        }
    }
}
"
```

5. Update containerd's configuration file at `/etc/containerd/config.toml`:

   a). Remove the comments from the 3 configuration lines.

   b). Insert your auth token as described in the line itself.

```
# Add the nvcr.io auth token (taken from docker’s config.json file above)
"
```

```
...
      [plugins."io.containerd.grpc.v1.cri".registry.mirrors."nvcr.io"]
          endpoint = ["https://nvcr.io"]
      [plugins."io.containerd.grpc.v1.cri".registry.configs]
        [plugins."io.containerd.grpc.v1.cri".registry.configs."nvcr.io".auth]
          auth = "<auth token as copied from docker's config.json file>"
...
"
```

Note: The file is **extremely** sensitive to spaces and indentation. Please make sure to use only spaces (' '), and to use two spaces per indentation level.

6. Restart the `containerd` service to apply your changes:
```
systemctl restart containerd
```
7. Log out from docker to erase the API key from docker's `config.json`:
```
docker logout nvcr.io
```

3.2. Pull Container YAML Configurations

This step pulls the `.yaml` configurations from NGC. If you have already performed this step for other DOCA containers you may skip to the next section.

1. Install unzip:
```
apt install unzip
```
2. Install NGC CLI according to the NGC Guide.
3. Configure the NGC CLI with your API key using `ngc config set`.
4. Download the set of `.yaml` configuration files, stored as a resource on NGC, using the CLI command shown on the resource page.

The resource contains a `configs` directory, under which can be found a dedicated folder per DOCA version. For example, `1.2` will include all currently available `.yaml` configuration files for DOCA 1.2 containers.

3.3. Activate Container-related Services

Start the following services required for the K8S deployment of your container:

```
# Going forward, enable both services
systemctl enable kubelet
systemctl enable containerd
# For now, start them both
systemctl start kubelet
systemctl restart containerd
```

3.4. Container-specific Instructions

Some containers require specific configuration steps for the resources used by the application running inside the container and modifications for the `.yaml` configuration file of the container itself.

Please refer to the container-specific instructions as listed under the container's respective page on NGC.

3.5. Spawn Container

Once the desired `.yaml` file is updated, simply copy the configuration file to Kubelet's input folder. Here is an example using the `doca_application_recognition.yaml`, corresponding to the DOCA AR application.

```
cp doca_application_recognition.yaml /etc/kubelet.d
```

Kubelet automatically pulls the container image from NGC and spawns a pod executing the container. In this example, the DOCA AR application starts executing right away, and its printouts would be seen via the container's logs.

3.6. Stop Container

The recommended way to stop a pod and its containers is as follows:

1. Delete the .yaml configuration file so that Kubelet will stop the pod:
   ```
   rm /etc/kubelet.d/<file name>.yaml
   ```
2. Stop the pod directly (only if it still shows "Ready"):
   ```
   crictl stopp <Pod ID>
   ```
3. Once the pod stops, it may also be necessary to stop the container itself:
   ```
   crictl stop <Container ID>
   ```

3.7. Useful Container Commands

- View currently active pods and their IDs (it might take up to 20 seconds for the pod to start):
  ```
  crictl pods
  ```
- View currently active containers and their IDs:
  ```
  crictl ps
  ```
- View all containers, including containers that recently finished their execution:
  ```
  crictl ps -a
  ```
- Examine the logs of a given container:
  ```
  crictl logs <Container ID>
  ```
- Attach a shell to a running container:
  ```
  crictl exec -it <Container ID> /bin/bash
  ```
- Examine the Kubelet logs, in case something didn't work as expected:
  ```
  journalctl -u kubelet
  ```

For additional information and guides on using `crictl`, refer to Kubernetes own documentation.

Chapter 4. Application Execution Within Container

4.1. Using Entrypoint Script

When possible, DOCA application containers are shipped with an init script, `entrypoint.sh`. This script is the first thing to spawn once a container boots and is responsible for executing the DOCA application. Using an application's `.yaml` file, we can control the command line arguments that the script passes to the application.

The exact command-line arguments are described per application on the application's respective reference guide, and the matching `.yaml` fields are described per application on the application's page on NGC.

4.2. Manual Execution From Within Container

Although most containers define the entrypoint.sh script as the container's ENTRYPOINT, this option is only valid for interaction-less sessions. As some DOCA applications expect an interactive shell session, the .yaml file supports an additional execution option.

Uncommenting the following 2 lines in the `.yaml` file causes the container to boot without spawning the application.

```
# command: ["sleep"]
# args: ["infinity"]
```

In this execution mode, you can attach a shell to the spawned container:

```
crictl exec -it <container-id> /bin/bash
```

Once attached, you get a full shell session, and you can execute the application as if it were running directly on the DPU, using the exact same command-line arguments.

When dealing with an application that spawns an interactive shell session, this option allows you to interact with the application directly through the shell.

Chapter 5. Troubleshooting Common Errors

Whenever there is some error with spawning a given container, it is recommended to first go over the list of common errors provided in this section. These errors account for the vast majority of deployment errors, and it is usually easier to verify them first before trying to parse the Kubelet journal log.

5.1. Yaml Syntax

The syntax of the `.yaml` file is **extremely sensitive**, and minor changes could break it and cause it to stop working. Things you should pay attention to are:

- Indentation – the file uses spaces (' ') for indentations (2 per indent). Using any other number of spaces causes an undefined behavior.
- Naming conventions – Both the pod name and container name have a strict alphabet (RFC 1123). This means that you must only use "-" and not "_", as the latter is an illegal character and cannot be used in the pod/container name. However, for the container's image name we will use "_" instead of "-". This helps differentiate between the two.

5.2. Resources

The container only spawns once all needed resources are allocated on the DPU and can be reserved for the container. The most notable resource in this case is the huge pages required for most DOCA applications.

Make sure that the huge pages are allocated as required per container. Both the amount and size of the pages are important and must match precisely.

5.3. Shared Folders and Files

If the `.yaml` file defines a shared folder between the container and the DPU, the folder must exist prior to spawning the container. If the application searches for a specific file within said folder, this file must exist as well. Otherwise, the application aborts and stops the container.

Chapter 6. DOCA Development Container

The set of DOCA-based containers hosted on NGC also includes a development container that can be used as part of two development workflows:

- To serve as a BlueField OS-like development environment
- Used for a multi-staged build of containers

The DOCA development container, doca:devel, is one of several flavors of the DOCA base image.

The `.yaml` file associated with this container is `doca_devel.yaml`, and the complete list of instructions for deploying the container can be found on the container's NGC page.