NVIDIA NIM LLM with Run:ai and Vanilla Kubernetes for Enterprise RA

Deployment, Scale and Sizing Guide

Version 01

ERA-DAS-002 | July 2025

Table of Contents

Abstract

As enterprises scale their AI initiatives, maximizing return on investment from accelerated infrastructure has become a strategic and crucial objective. Most Enterprises are looking to run Inference services on Large Language models (LLMs) as a start but are also exploring running multiple models for varying use cases, to manage costs or to get better accuracy. Running multiple Inference services while using traditional methods of allocating GPUs statically often leads to underutilization, fragmented workloads, and increased operational overhead. This paper helps guide enterprises on how to pack more Inference models on a given set of NVIDIA GPUs using NVIDIA Run:ai, through intelligent scheduling, fractional GPUs, and dynamic resource management. We also explore the impact on performance with the Run:ai scheduler on utilizing fractional GPUs for NIM LLMs.

Introduction

NVIDIA NIMs simplify the deployment and management of inference services across a wide range of AI models. Delivered as pre-packaged, containerized inference servers, NIMs are readily available through the NVIDIA NGC catalog and are designed for rapid integration into enterprise AI infrastructure. Each NVIDIA NIM has a minimum number of GPUs that are needed to run Inference against a specific model. The minimum no. of GPUs that can support a specific number of users/workloads that can run concurrent Inference sessions against NIM, based on how the NIM is optimized (latency v/s throughput).NIMs can be scaled easily by adding more GPUs as workload demands grow.

The overall workload on a NIM service can vary at any point in the day, the challenge however is all the GPUs that are allocated to a particular NIM are consumed by that NIM regardless of the workload on that NIM, or in case of scale, the system GPUs might be scaled for max capacity, however only a percentage of users are consuming that NIM so the GPUs can be scaled down and can be given to another workload.
At times, enterprises might have powerful GPUs that can fractionally accommodate more than one NIM workload. While the GPUs are idle, they cannot be deallocated and allocated to other workloads because these GPUs are statically assigned.

NVIDIA Run:ai is an intelligent workload manager that helps orchestrate AI workloads across a resource pool of GPUs. It can automatically scale up/down an inference service based on workload and can allocate fractional GPUs for various NIMs, Run:ai can also prioritize GPU allocation based on users, workload etc.

Scope

This paper covers how to optimize and scale NVIDIA GPUs to run NVIDIA NIMs using Run:ai on an Enterprise Reference Architecture. The following NIMs and GPUs are covered.

Table1. Scope of NIMs and GPUs

NVIDIA NIM for LLMs	NVIDIA GPUs
Meta Llama 3.1 8B Instruct	H100 NVL
DeepSeek R1 Distill Llama 8B	H100 NVL

Intended Audience

This Guide is meant to help NVIDIA partners architecting Gen AI-based Large Language Models (LLMs) on Enterprise Infrastructure and sizing for cluster-level deployments. The guide can be used for both new and existing deployments to determine the capacity and scale needed based on the LLM workload. It can also be used by the following Personas as they architect their Enterprise solutions.

Enterprise Architects: Enterprise Architects tasked with designing and defining servers, GPU's and Networking gear to determine what Infrastructure resources will be needed to support specific workloads around LLMs based on NIMs

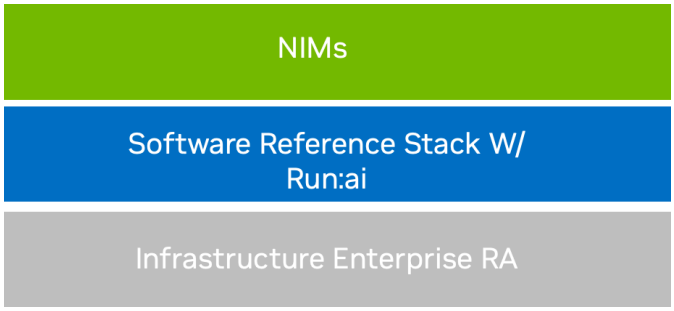
MLOps Engineer: MLOps Engineers can use this to define Infrastructure requirements as they talk to Infrastructure/Cloud teams to carve out resources to run Inference Services on LLMs based on NIMs

Platform Engineer: Platform Engineers can use this guide to determine how to design their Container Environment around Kubernetes, and also what kind of resources will be needed by the cluster to support various LLM workloads based on NIMs

Systems Overview

For this guide, we are leveraging Enterprise Reference Architecture (RA) in the 2-4-3-200 configuration using the Software Reference Stack with Run:ai and deploying NIMs on top of the stack

Figure 1. System Overview



Note: For the detailed hardware design, refer to NVIDIA H100 NVL NVIDIA Spectrum Platforms Enterprise Reference Architecture: NVOnline | NVOnline: 1119885

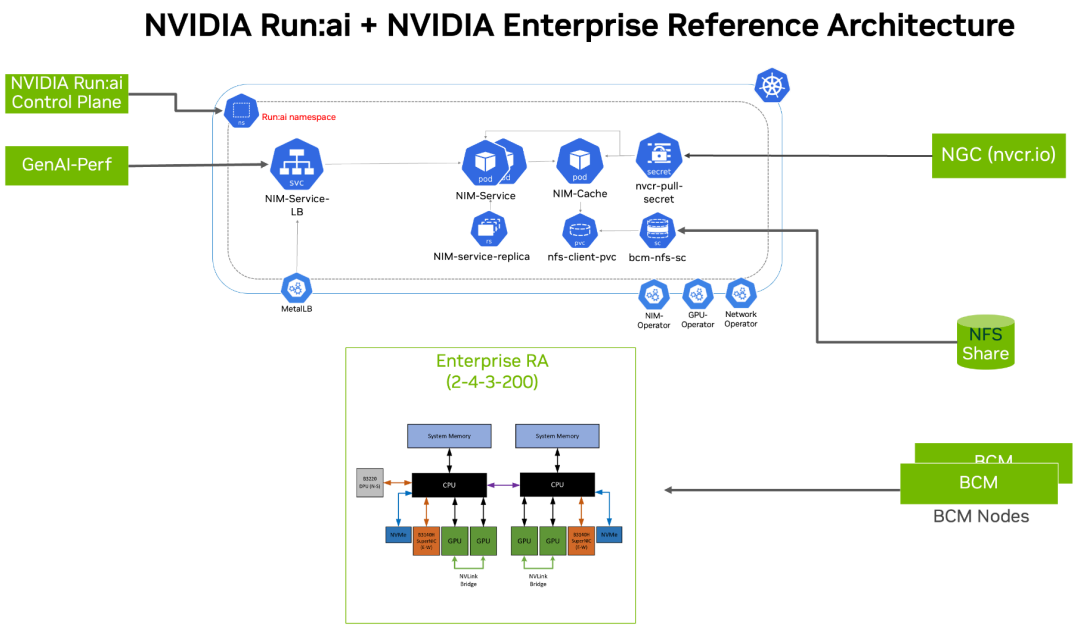
Run:ai Overview

Run:ai is NVIDIA's Kubernetes-native AI workload and GPU orchestration platform, purpose-built to help enterprises manage and scale AI workloads across heterogeneous infrastructure - on-prem, in the cloud and hybrid.
It enables centralized orchestration of AI compute for multiple departments and teams, supporting advanced scheduling, quota enforcement, and GPU resource sharing to maximize infrastructure efficiency and AI workload performance.
Run:ai replaces the default Kubernetes scheduler with a purpose-built AI scheduler, enabling fine-grained control over GPU allocation, including GPU fractioning and advanced scheduling policies tailored to the unique needs of AI and deep learning workloads.

Run.ai has two major components.

1. **The Run:ai Control Plane**: The centralized management layer that orchestrates individual GPU clusters. It enforces policies, manages workloads, and provides the UI and API for interacting with Run:ai environments.
2. **Run.ai Cluster**: A local instance that receives instructions from the Control Plane. It handles resource management and submits workloads at the cluster level. Multiple Run:ai Clusters can be connected to a single Control Plane instance for unified management.

Figure 2. Run:ai Architecture



Run:ai has three modes of operation,

1. **Saas /Classic**: In this architecture, the Run:ai control plane is in the cloud, and the cluster can be in any data center/Cloud
2. **Self-Hosted**: In this mode, both the Run:ai control plane and cluster are installed within the enterprise's datacenter/cloud, but requires connection to the internet to download bits
3. **Air-Gapped**: This mode is Self-Hosted but requires no Internet connection

Note: The scope of this Guide is limited to the SaaS/Classic Installation method.

Enterprise Reference Architecture Overview

This guide is part of NVIDIA Enterprise Reference Architecture, which covers certified hardware, software stack, and sizing recommendations to design, build, and scale an end-to-end accelerated computing cluster deployment with balanced CPU to GPU to NIC patterns. The Enterprise Reference architecture provides guided and detailed hardware and software architecture recommended by NVIDIA for optimal server, cluster, and network configuration needed to build and scale AI factories.

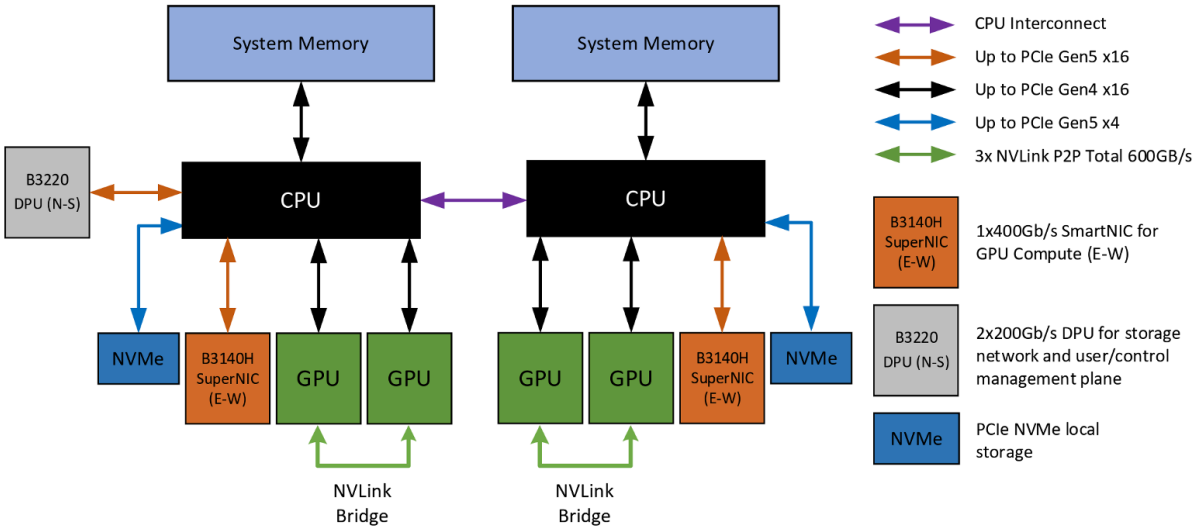
NVIDIA Enterprise Reference Architecture includes hardware design recommendations, software stack configurations, and scalability.

Hardware Enterprise Reference Architecture

For this version of the document, we used NVIDIA's 2-4-3-200 Enterprise RA reference configuration for the overall stack. The PCIe-Optimized 2-4-3 (CPU-GPU-NIC-Bandwidth) reference configuration is for 2U NVIDIA-Certified compute nodes using PCIe, allowing you to deploy up to 4 GPUs with up to 3 NICs balanced with 2 CPUs. This pattern can scale from 4 to up to 32 nodes in a cluster. The Enterprise RA design recommends using NVIDIA Spectrum-X switches, Ethernet Platform - Combining Spectrum-4 Ethernet switches and NVIDIA Bluefield-3 SuperNICs for optimized networking.

	Note: For the detailed hardware design, refer to NVIDIA L40S and H100 NVL NVIDIA Spectrum Platforms Enterprise Reference Architecture: NVOnline : 1119410 | NVOnline : 1119885

Figure 3. System architecture of Enterprise RA 2-4-3-200 reference configuration



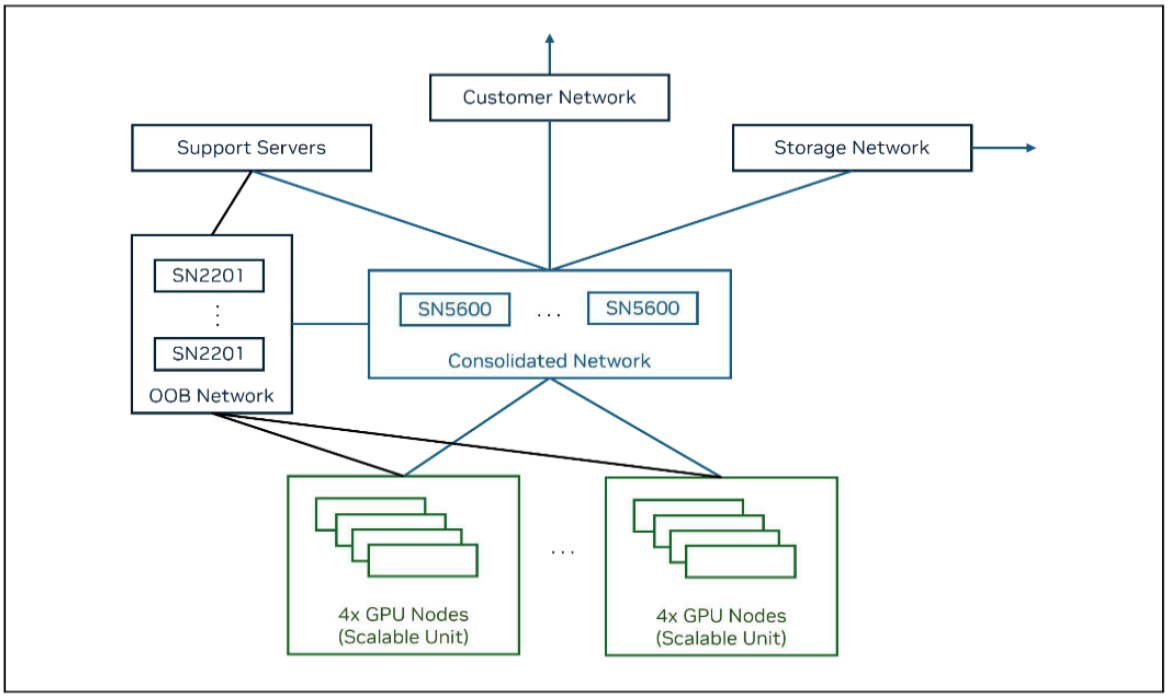
The server used while running the test on the Software Enterprise RA had the following characteristics.

Table 2. Specification of Individual Components in the server with NVIDIA H100 GPUs

Component	Specification
CPUs	2x INTEL(R) XEON(R) GOLD 6548Y+ 232 cores; 64 threads
GPUs	4 x NVIDIA H100 NVL
Networking – E/W	2 x NVIDIA BlueField-3, B3140H
Networking – N/S	1 x NVIDIA BlueField-3, B3220
Host Memory	32x 64 GB DRAM (2048)
Host Boot Drive	2x 896 GB (~1.8 TB)
Host Storage	8x 1787.88 GB (~14TB)

The Systems are connected with NVIDIA SN5600 switches, and below is the reference architecture for Networking. The Scalable Units are a pool of servers with the 2-4-3-200 architecture.

Figure 4. Networking Reference for Enterprise RA



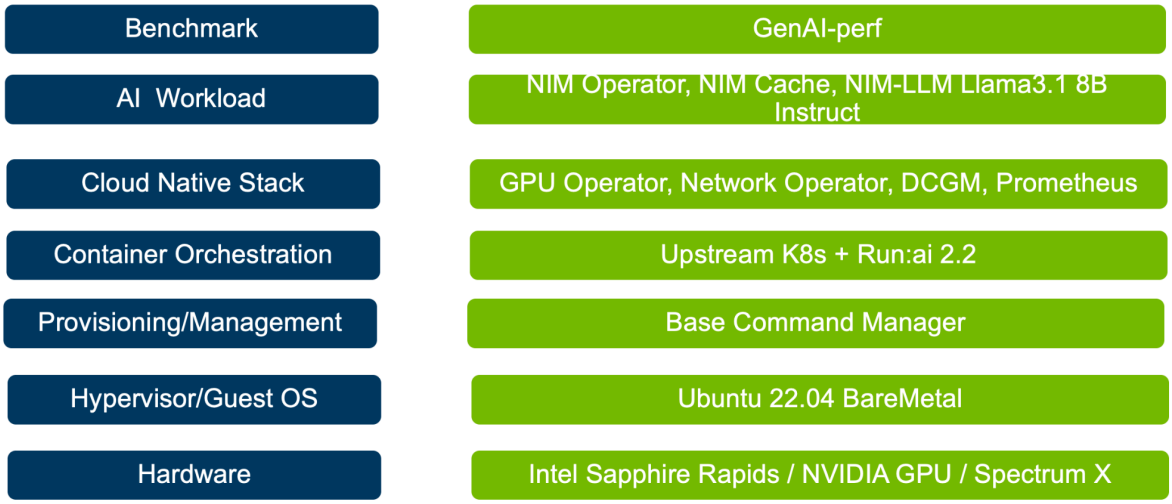
For this particular guide, we are using NFS Storage that has been deployed on the BCMe head node to provide Persistent Volumes to NIM workloads. Enterprise customers can use their storage systems to provide storage volumes needed by Kubernetes and the Inference Services (NIMs).

Our current Enterprise RA data center deployment consists of a pool of 16 servers with 4 NVIDIA H100 NVL GPUs per node. The complete cluster has 64 GPUs in total.

Software Reference Stack

The software reference stack leverages the bare-metal servers with Kubernetes as the cluster orchestration tool. BCMe is part of the NVIDIA AI Enterprise software suite, and it provides all the tools you need to deploy and manage an AI datacenter. It also helps in deploying a Kubernetes cluster on top of the bare-metal servers to create a pool of GPU resources. BCM is then used to deploy all the operators like the GPU Operator, Network Operator, NIM Operator, etc. to run the GPUs and Network cards effectively. We then deploy a NIM service, picking the model for which the Inference service needs to be tested and scaled. We installed Run:ai version 2.2 using the SaaS format.

Figure 5. Overview of the overall deployment stack



This Enterprise Reference Architecture leverages NVIDIA AI Enterprise software to install and configure the necessary software and tools required to efficiently deploy and operate an AI factory. Once servers are racked and networked, BCMe can be used to image individual servers, deploying Ubuntu 22.04 as the operating system, installing NVIDIA GPU and network drivers, and setting up Kubernetes clusters.

Beyond these core capabilities, the deployment also aligns the software dependencies for various components required for Kubernetes cluster operations, such as a **Container Network Interface (CNI)** for managing container networks, **NGINX Ingress Controller** for handling cluster ingress traffic, and **MetalLB** for load balancing services within the Kubernetes environment also provides firmware to configure and optimize OS like Cumulus Linux for the NVIDIA Spectrum switches, firmware for the Bluefield Super NICs etc.

	Note: For detailed Software reference design, refer to NVIDIA Software Reference Stack and automation for Enterprise RA - vanilla Kubernetes: NVOnline : 1130533

In this document, we will leverage NVIDIA NIM and install it on top of the Enterprise RA stack, test the performance of the inference server and show how the inference service scales as workloads change.

NVIDIA Inference Microservice (NIM)

NVIDIA Inference Microservice (NIM) is a pre-trained, customized AI model packaged in the form factor of a Container optimized to run on NVIDIA Systems like Data Centers, RTX AI PCs and workstations. NVIDIA NGC Catalog hosts many NIMs for various AI model domains like Large Language Models (LLMs), Vision Language Models (VLMs), models for speech, medical imaging, 3D, Videos etc.

For this guide, we are scaling and testing the following Large Language (LLMs) Model packed in NVIDIA NIMs, NVAIE also includes Support for these NIMs.

Table 3. NVIDIA NIM - Models & versions

Model	NIM Version
Llama 3.1 8B Instruct	1.8.4
DeepSeek R1 Distill Llama 8B	1.5.2

System Configuration

Pre-requisites for installing Run:ai and NIM LLM

- A Kubernetes Cluster with a supported version installed with NVIDIA GPU Operator and NVIDIA Network Operator installed, which should already be in place if the NVIDIA Software Enterprise RA is followed.
- Active Subscription to NVAIE and Access to the NGC Enterprise Catalog. Please generate and download your nvcr.io access token in NGC. For more information on how to get the access token, refer to the following guide.
- Install Helm in the Kubernetes Cluster and download the helm CLI. This should have been installed if the Software Enterprise RA is followed.

- Access to Kubernetes Clusters config file and `kubectl` CLI installed, this file is in the Enterprise RA cluster BCMe head node under `/<user>/.kube/config`
- A Storage Class with present to create Persistent Volume Claims by Kubernetes, this can be an NFS or Block-based storage class, the storage class name is `default.`
- Get the Run.ai SaaS Login, the run.ai portal, where the control pane requires a login to be created by NVIDIA. Please work with Account/SA teams to get an org carved out for this.
- Access to the BCM node through which the Kubernetes Cluster was installed

Pre-reqs for RunAI

Please install the Run:ai Cluster and Control plane components on the Enterprise RA cluster. The reference implementation steps have been provided at the end of this document in Appendix A.

At the end of the Run:ai installation you should have

- The Enterprise RA cluster added to the Run:ai Control Plane
- A Project created in Run:ai that has all the Nodes and 64 GPUs allocated to the cluster and a different namespace is used for Run:ai Project
- The node pool in Run:ai has the placement strategy set to `Spread` instead of `Bin-Pack` for GPUs and CPUs
- Users are created in Run:ai

Please refer to Appendix A for reference implementation in a SaaS mode/Classic mode.

Deploy and Configure NIM LLM on Run:ai

Create Data Source for NIM

The NIM inference service will need storage to load the NIM Cache on to, for that we need to create a Persistent Volume drive in the project Run:ai is going to use.

Create a Persistent Volume for the Data Store that Run:ai Inference workload will use, replace <namespace-project> with the project name created in step above.

Also, replace storage class with the StorageClass Name with the name in the cluster, change the server to the NFS server's IP address, and the Path to the share/mount the NFS server is configured with

```
kubectl apply -f nfs-pv.yaml -n <namespace-project>

apiVersion: v1
kind: PersistentVolume
metadata:
 name: nfs-pv
spec:
 capacity:
   storage: 100Gi
 accessModes:
   - ReadWriteMany
 persistentVolumeReclaimPolicy: Retain
 storageClassName: nfs-storage
 nfs:
   server: 10.185.118.25
   path: /mnt/cm-nfs
```

Create a PVC that can be used, replace <namespace-project> with the namespace in k8s Run:ai is using

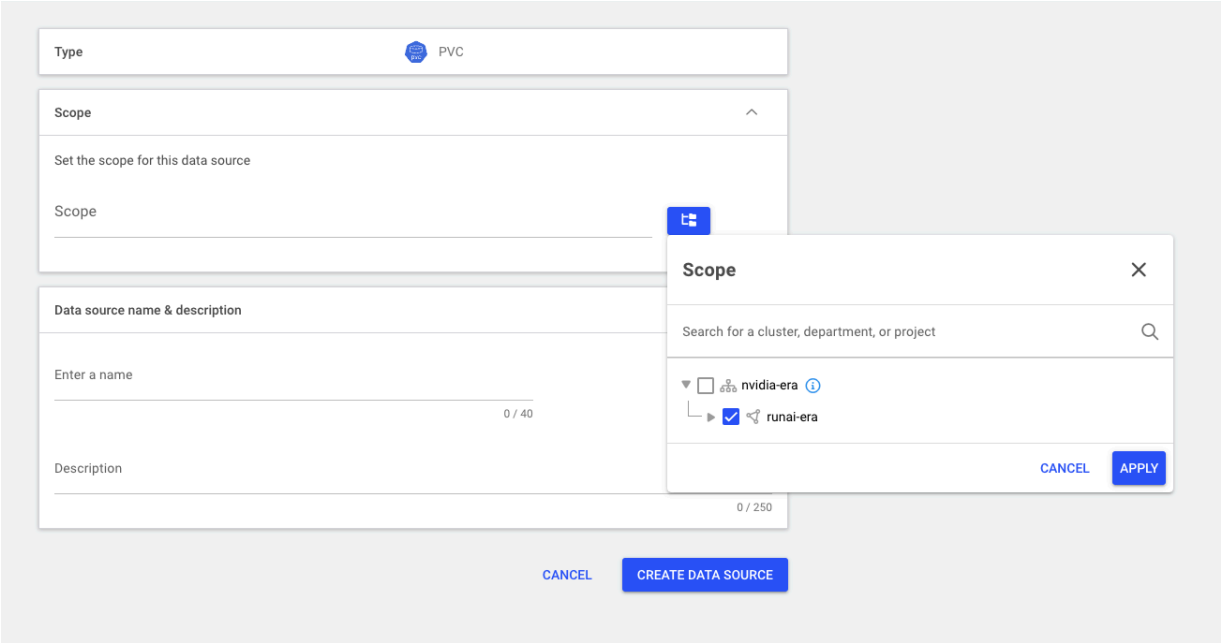
```
kubectl apply -f nfs-pvc.yaml -n <namespace-project>

apiVersion: v1
kind: PersistentVolumeClaim
metadata:
 name: nfs-pvc-nim
spec:
 accessModes:
   - ReadWriteMany
 resources:
   requests:
     storage: 100Gi
 storageClassName: nfs-storage
```

Create a Data Source in Run:ai to use the PVC created above

Go to `Workload Manager` → `Assets` → `Data Sources`, click on `New Data Source` → `PVC`

Select the scope to your project, click apply

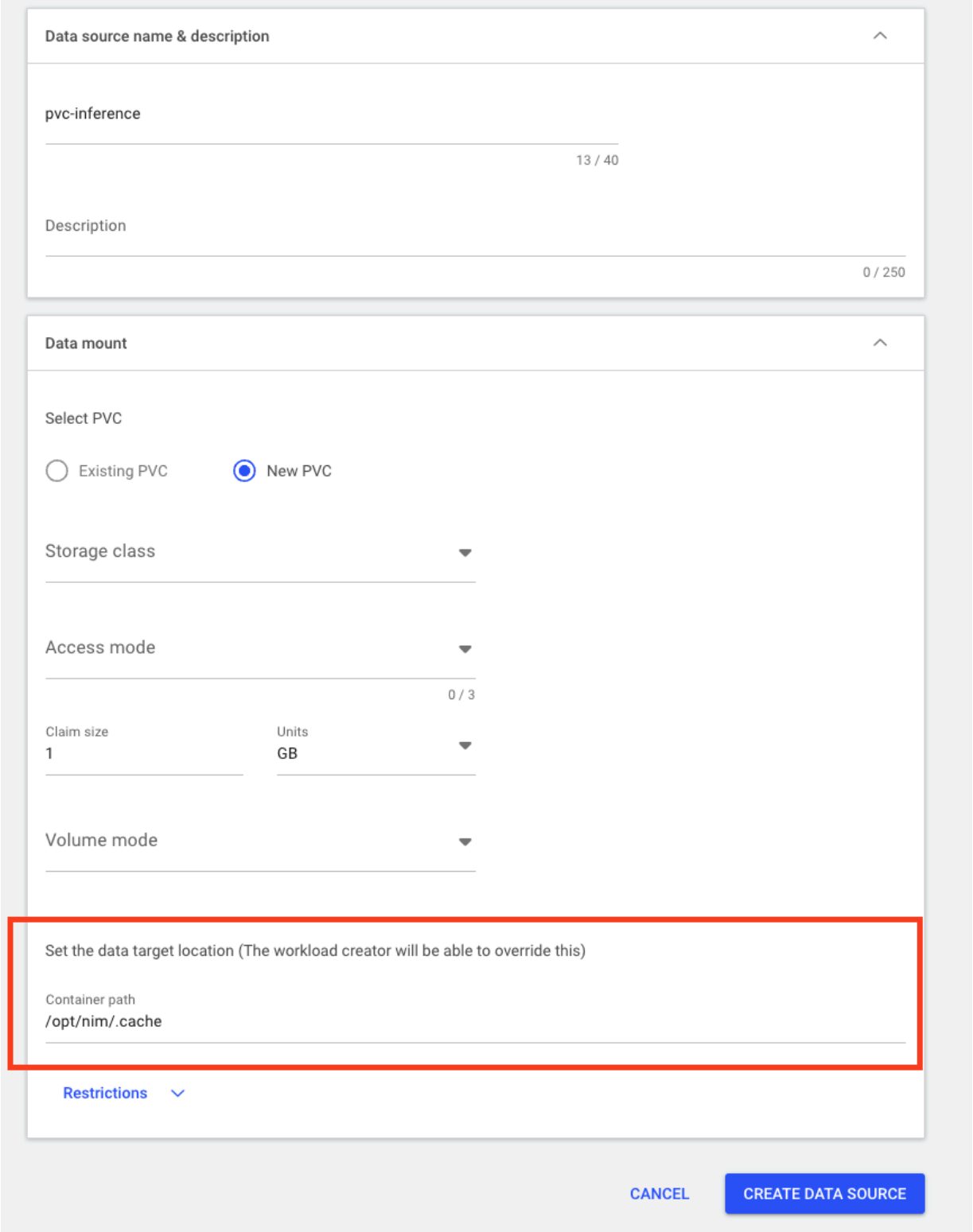
Figure 6. Create a Data Source - Assign Scope



Enter a name, select `Existing PVC` in the `Data Mount` field. Select the PVC name you created in the step above. Make sure to enter `/opt/nim/.cache` as the `Container Path;` this is where the NIM service will download the NIM cache.

Click `Create Data Source`

Figure 7. Create DataSource- Details


Data source name & description
pvc-inference
13 / 40
Description
0 / 250
Data mount
Select PVC
Existing PVC
New PVC
Storage class
Access mode
0 / 3
Claim size
1
Units
GB
Volume mode
Set the data target location (The workload creator will be able to override this)
Container path
/opt/nim/.cache
Restrictions
CANCEL
CREATE DATA SOURCE

Create a Secret

A secret is needed for the workload to access NGC and download the correct NIMs.

Create a NGC-API Secret

Login to the Run:ai UI , Go to `Workloads` → `Assets` → `Credentials`

Click on `New Credential` → `Generic Secret`

Select the Scope to the Project in use, enter a Name for the Credential

Under Secret, click `New Secret` and add `NGC_API_KEY` in the key and your NGC API token in the `Value` field

Click `Create Credentials`

Add Docker Registry Secret

Docker Registry secret is needed by run.ai to access nvcr.io.

Login to Run.ai, Go to `Workloads` → `Assets` → `Credentials`

Click on `New Credential` → `Docker Registry`

Click `Create Credentials`

Select the Scope of the Project in use, and enter a Name for the Credential.

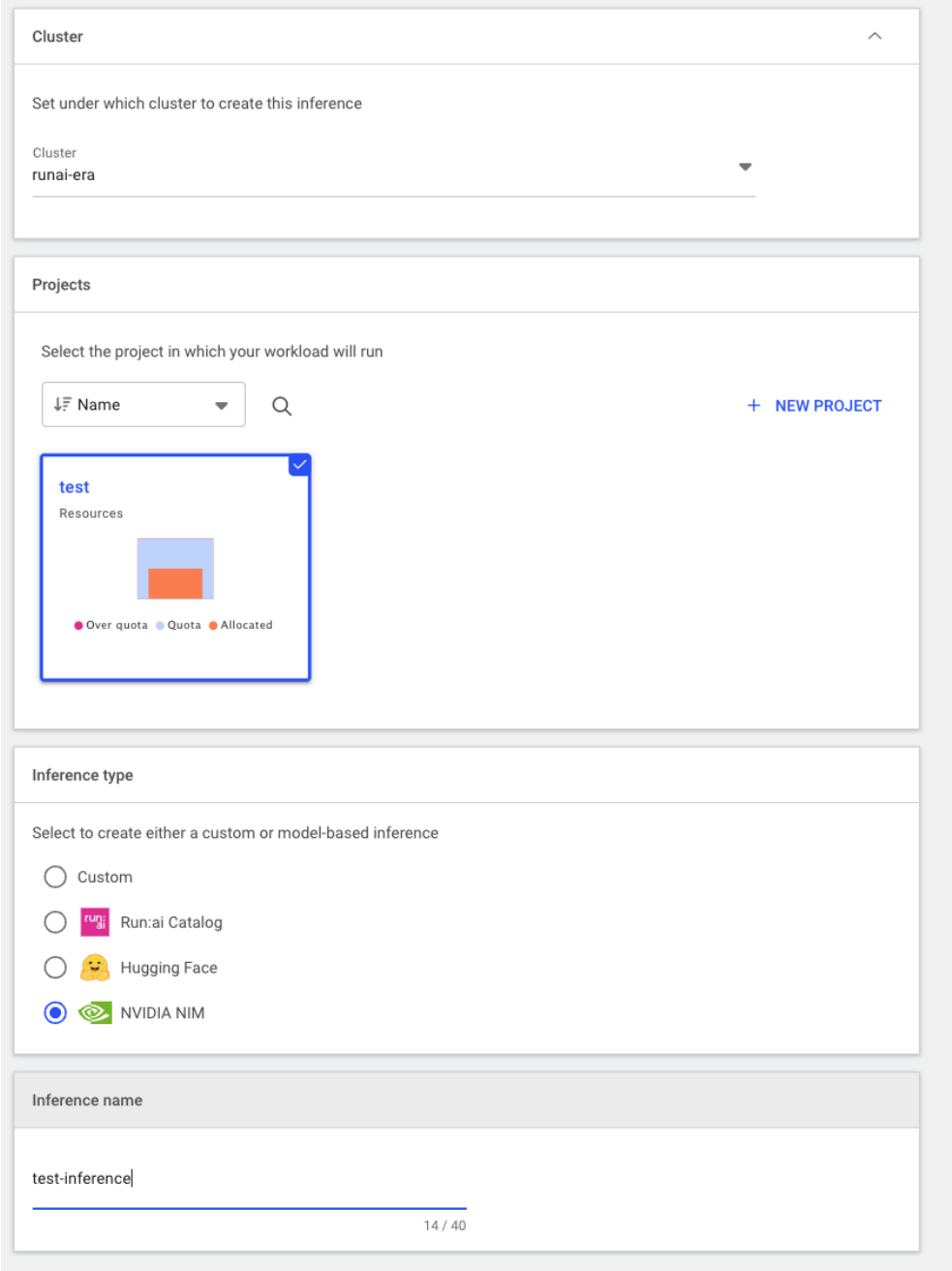
Under Secret, click `New Secret` and add `$oauthtoken` in the key and your NGC API token key in the `Value` field

Deploy a Specific NIM LLM

Go to Run:ai, click on `Workload Manager` → `Workloads` → Click on `New Workload`

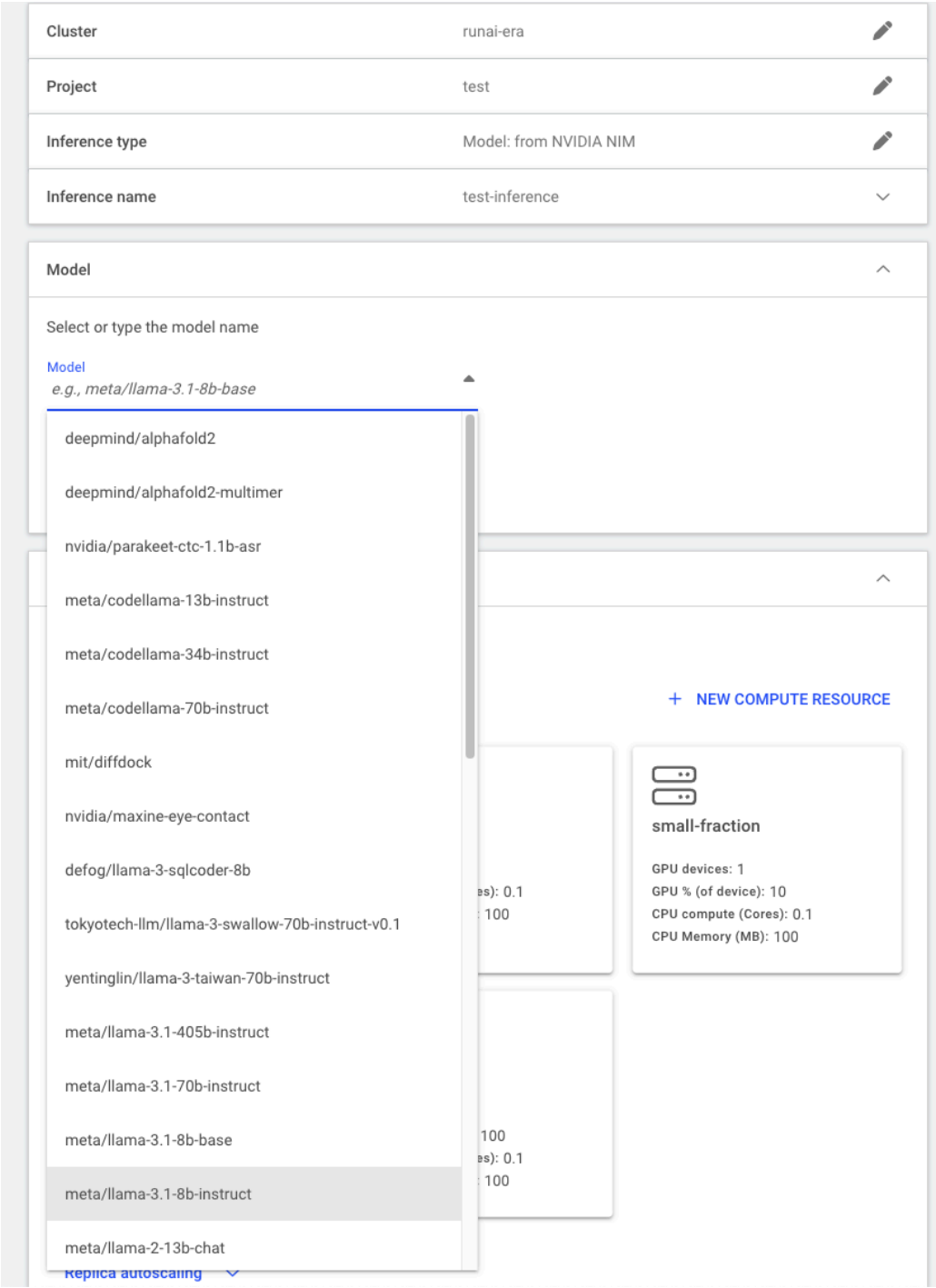
Select `inference`, select the Project you will be using, Select `Inference Type` as `Nvidia NIM`, give the Inference workload a name, and Click `Continue.`

Figure 8 . Deploy NIM- Provide Name



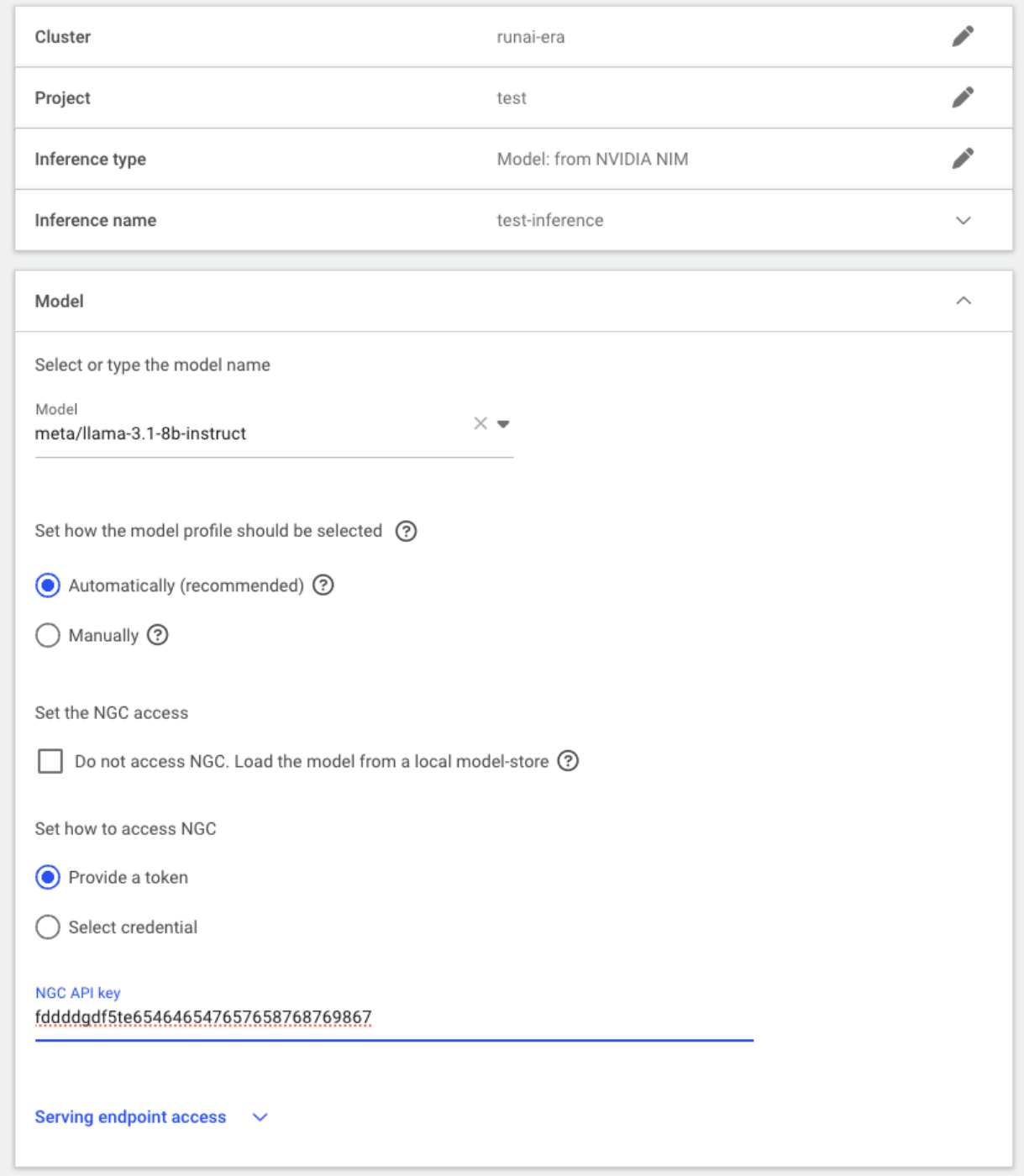
Select `Meta/Llama-3.1-8b-instruct` from the dropdown list for Model names

Figure 9 . Create NIM- Select Model


Cluster
runai-era
Project
test
Inference type
Model: from NVIDIA NIM
Inference name
test-inference
Model
Select or type the model name
Model
e.g., meta/llama-3.1-8b-base
deepmind/alphafold2
deepmind/alphafold2-multimer
nvidia/parakeet-ctc-1.1b-asr
meta/codellama-13b-instruct
meta/codellama-34b-instruct
meta/codellama-70b-instruct
mit/diffdock
nvidia/maxine-eye-contact
defog/llama-3-sqlcoder-8b
tokyotech-llm/llama-3-swallow-70b-instruct-v0.1
yentinglin/llama-3-taiwan-70b-instruct
meta/llama-3.1-405b-instruct
meta/llama-3.1-70b-instruct
meta/llama-3.1-8b-base
meta/llama-3.1-8b-instruct
meta/llama-2-13b-chat
+ NEW COMPUTE RESOURCE
small-fraction
GPU devices: 1
GPU % (of device): 10
CPU compute (Cores): 0.1
CPU Memory (MB): 100

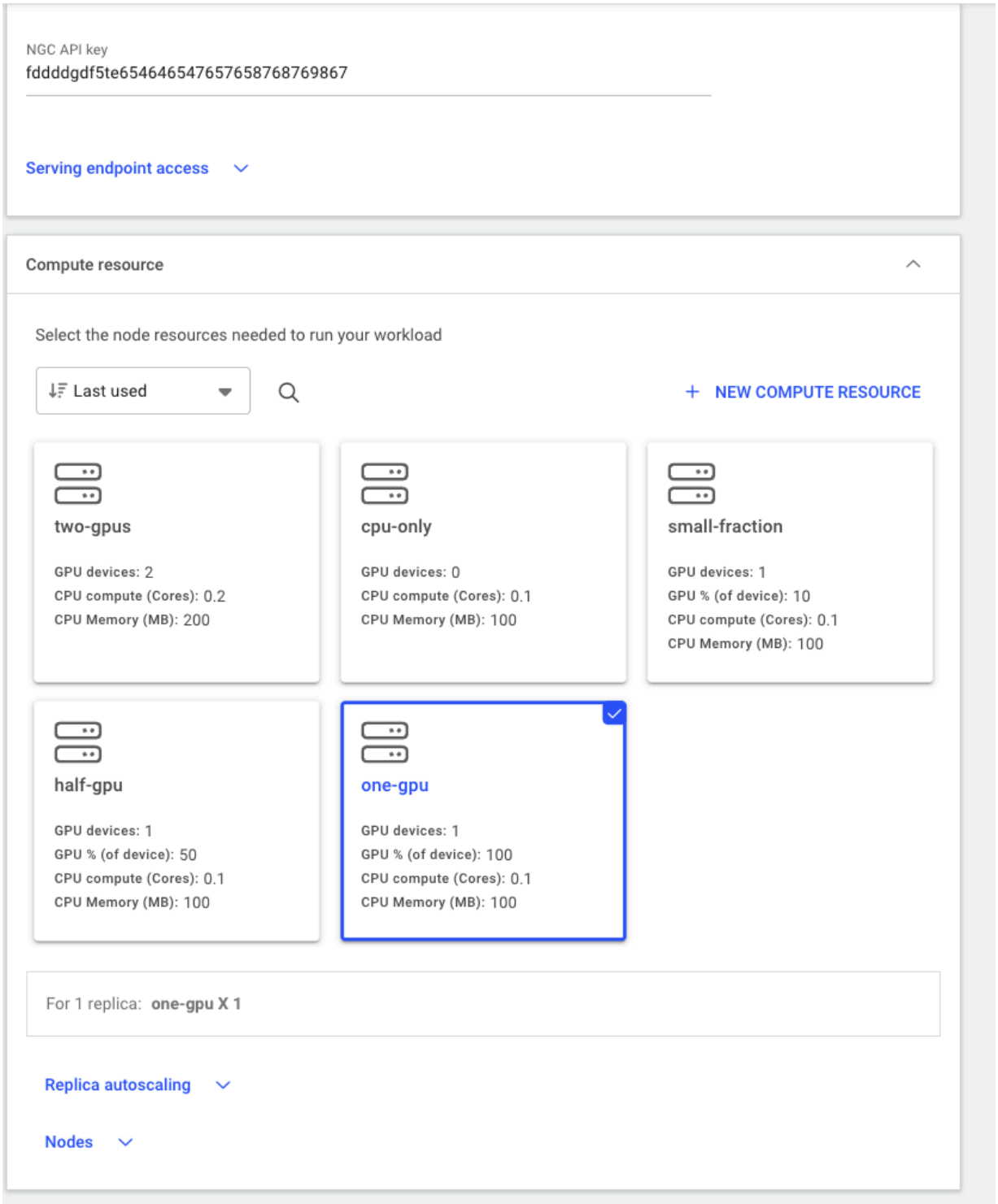
Make sure the `Provide a token` radio button is selected and enter your `NGC API Key` in the field.

Figure 10. Create NIM- Provide NGC Key



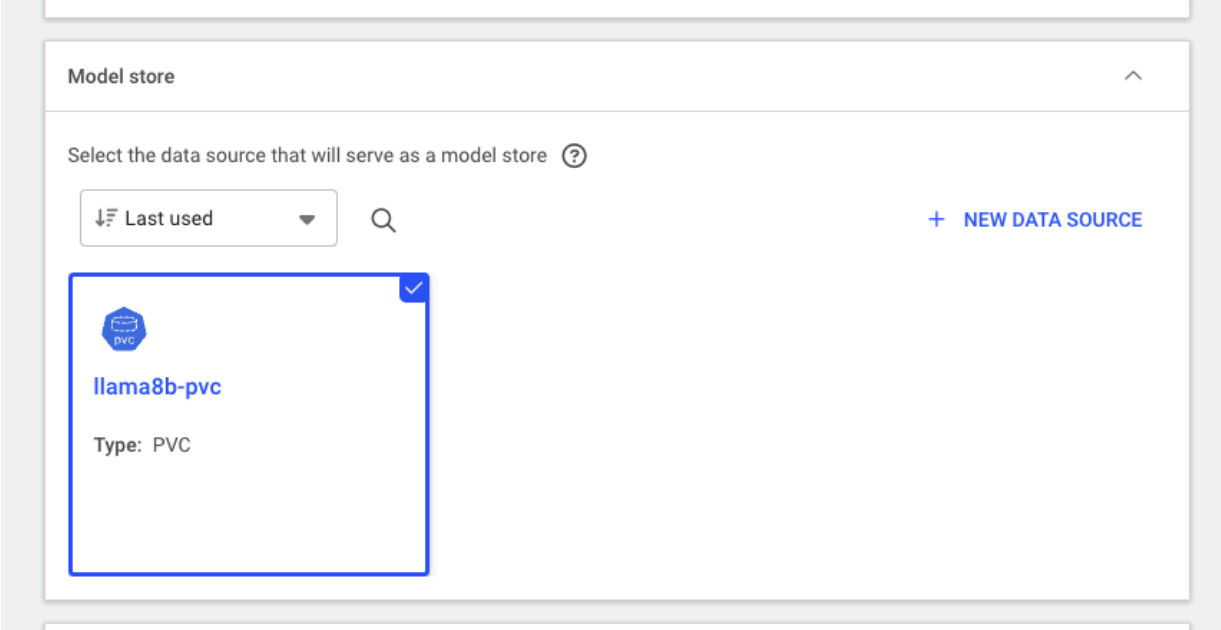
Select `one-gpu` tile; this varies by the workload you select

Figure 11. Create NIM- Provide Resources


NGC API key
fddddgdf5te6546465476576587687698 67
Serving endpoint access
Compute resource
Select the node resources needed to run your workload
Last used
NEW COMPUTE RESOURCE
two-gpus
GPU devices: 2
CPU compute (Cores): 0.2
CPU Memory (MB): 200
cpu-only
GPU devices: 0
CPU compute (Cores): 0.1
CPU Memory (MB): 100
small-fraction
GPU devices: 1
GPU % (of device): 10
CPU compute (Cores): 0.1
CPU Memory (MB): 100
half-gpu
GPU devices: 1
GPU % (of device): 50
CPU compute (Cores): 0.1
CPU Memory (MB): 100
one-gpu
GPU devices: 1
GPU % (of device): 100
CPU compute (Cores): 0.1
CPU Memory (MB): 100
For 1 replica: one-gpu X 1
Replica autoscaling
Nodes

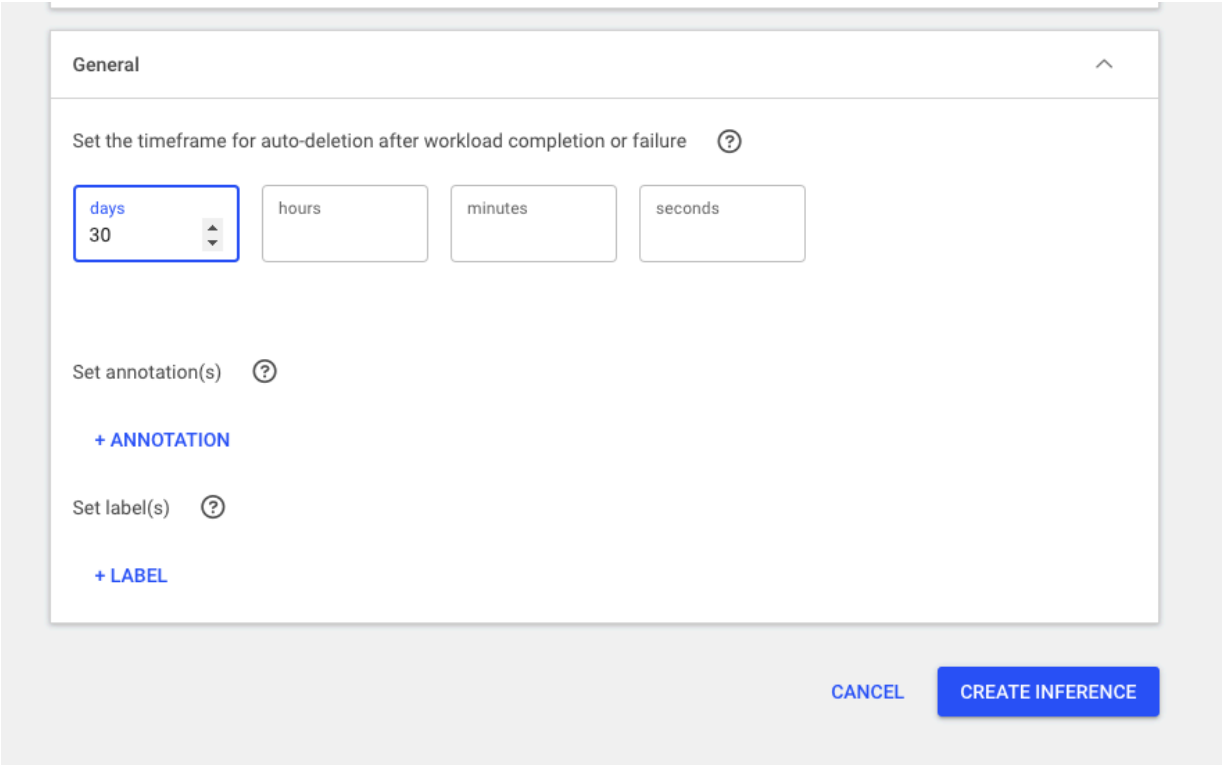
Click on Model store to expand it, and select the Data Source we created in the steps above.

Figure 12. Create NIM- Provide PVC



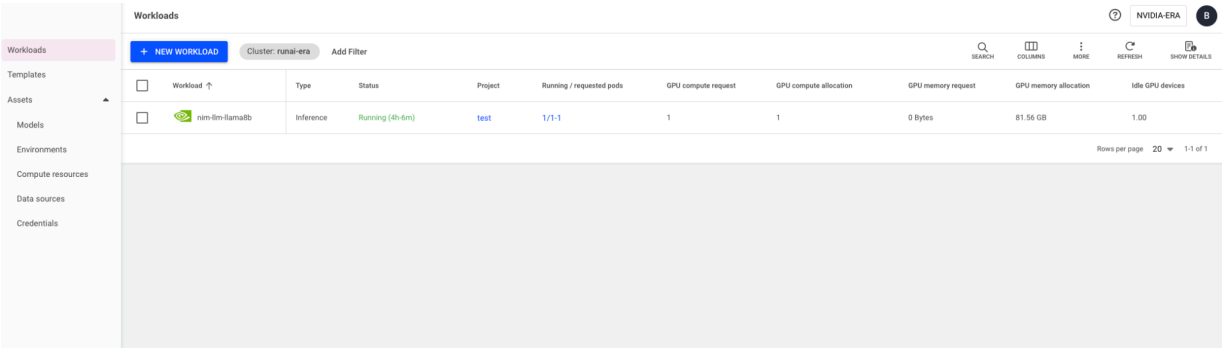
Expand the General section and select `days 30` box , click `Create Inference.`

Figure 13. Create NIM



You should see the inference service running in a few minutes.

Figure 14. Create NIM- Running status



Alternate YAML method

You can create an Inference request via YAML as well. Submit the below sample YAML, replace the NGC_API with your API key to Kubectl

```
apiVersion: run.ai/v2alpha1
kind: InferenceWorkload
metadata:
 name: llama-8b-single-gpu
 namespace: runai-test
spec:
 environment:
   items:
     NGC_API_KEY:
       value: nvapi- # Update this to your API key, no quotes
     NIM_CACHE_PATH:
       value: /opt/nim/.cache
 gpu:
   value: "0.5"
 image:
   value: nvcr.io/nim/meta/llama-3.1-8b-instruct
 minScale:
   value: 1
 maxScale:
   value: 2
 metric:
   value: concurrency #
 target:
   value: 1000  #
 ports:
   items:
     port1:
       value:
         container: 8000
         protocol: http
         serviceType: ServingPort
 pvcs:
   items:
     pvc:
       value:
         claimName: nfs-pvc-nim
         existingPvc: true
         path: /opt/nim/.cache
```

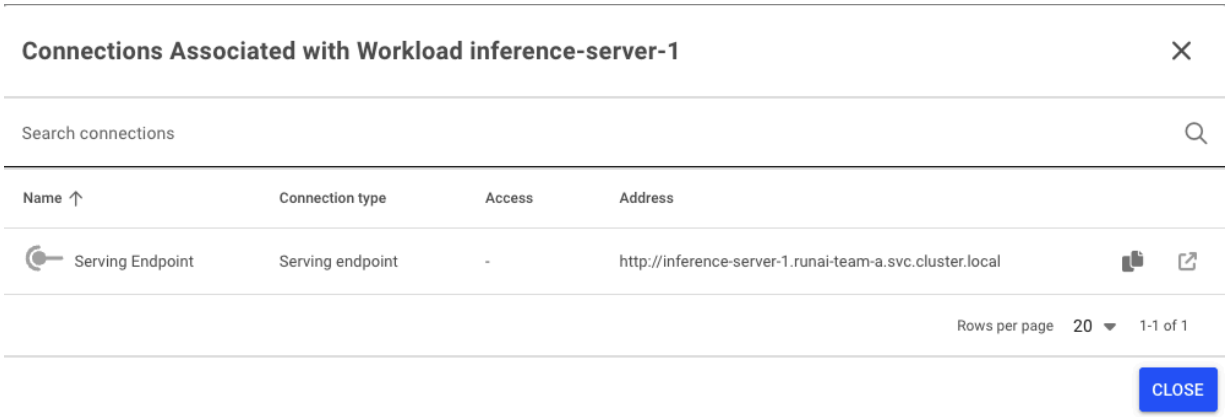
```
        readOnly: false
```

Query the Inference Server

Find the Inference Server Endpoint

- Under `Workloads`, select `Columns` on the top right. Add the column `Connections`.
- See the connections of the `inference-server-1` workload:

Figure 15 . NIM Serving Endpoint



Run:ai creates an External Name for the inference service, we can expose the service on an LB IP using the metal-LB already deployed on the cluster. Get the `Deployment` name of the inference service in Kubernetes by

kubectl get deployment -n <runai-namespace-project>

Then, run the following, change the name to the name that's on your cluster

```
kubectl expose deployment <deployment-name> --port=8000 --target-port=8000
--name=nim-lb-service-h100 --type=LoadBalancer -n <runai-namespace-project>
```

Check the Load Balancing IP of the Inference Service bu:
kubectl get svc -n <runai-namespace-project>

Validate everything is running my running curl or pointing a browser to
http://<LB-IP-Infernece-service>:8000/v1/models

You should see an output like below:


```
{
  "object": "list",
  "data": [
    {
      "id": "meta/llama3-8b-instruct",
      "object": "model",
      "created": 1741026705,
      "owned_by": "system",
      "root": "meta/llama3-8b-instruct",
      "parent": null,
      "permission": [
        {
          "id": "modelperm-777c1ee100c846b1ad8e0b4d530f4f1e",
          "object": "model_permission",
          "created": 1741026705,
          "allow_create_engine": false,
          "allow_sampling": true,
          "allow_logprobs": true,
          "allow_search_indices": false,
          "allow_view": true,
          "allow_fine_tuning": false,
          "organization": "*",
          "group": null,
          "is_blocking": false
        }
      ]
    }
  ]
}
```

Performance and Scale Methodology

Run:ai works with Kubernetes to add a custom scheduler that manages the allocation of GPU resources to the AI workloads. This scheduler can also allocate fractional GPUs to the Workloads. In order to benchmark this solution, we first established the Inference workload benchmarks using the Gen-AI perf tool without Run:ai scheduler. Once that was established, we ran the same benchmarking process on the Inference workload deployed with Run:ai, using both fractional and full GPUs. We ran this for two models, Meta Llama 3.1 8B and DeepSeek R1 Distill Llama 8B. Once the Performance benchmarks were established, we then scaled the GPUs, the NIM Pod instances and the workload on the NIM LLM using Gen-AI Perf.

Benchmarking NIM LLM

To benchmark the NIM service, we used NVIDIA's GenAI-Perf , a client-side tool, from within the Triton inference server running outside the cluster to generate load on the NIM service API. Gen-AI Perf is a tool developed by NVIDIA to measure the throughput and latency of generative AI models. It provides some key metrics to measure performance and benchmark.

Typically, in an LLM application, a user provides a query (prompt), the inference service queues the request, then processes it, and a response is generated. The model takes the user input prompt and breaks it down into tokens for efficient processing. As such, there are some common terms used to define the process

Tokens: The unit that LLM use to break a prompt into for processing

Input Sequence Length (ISL): The number of tokens entered by the user

Output Sequence Length (OSL): The number of tokens produced by the inference request at the generation phase.

Based on this, GenAI Perf can load an inference server with various ISL and OSL sequences to mimic Summarization, translation prompts, mimicking the scenarios a user might use the Inference server for, and provide various metrics. Here are some of the key metrics to monitor for an LLM Inference request:

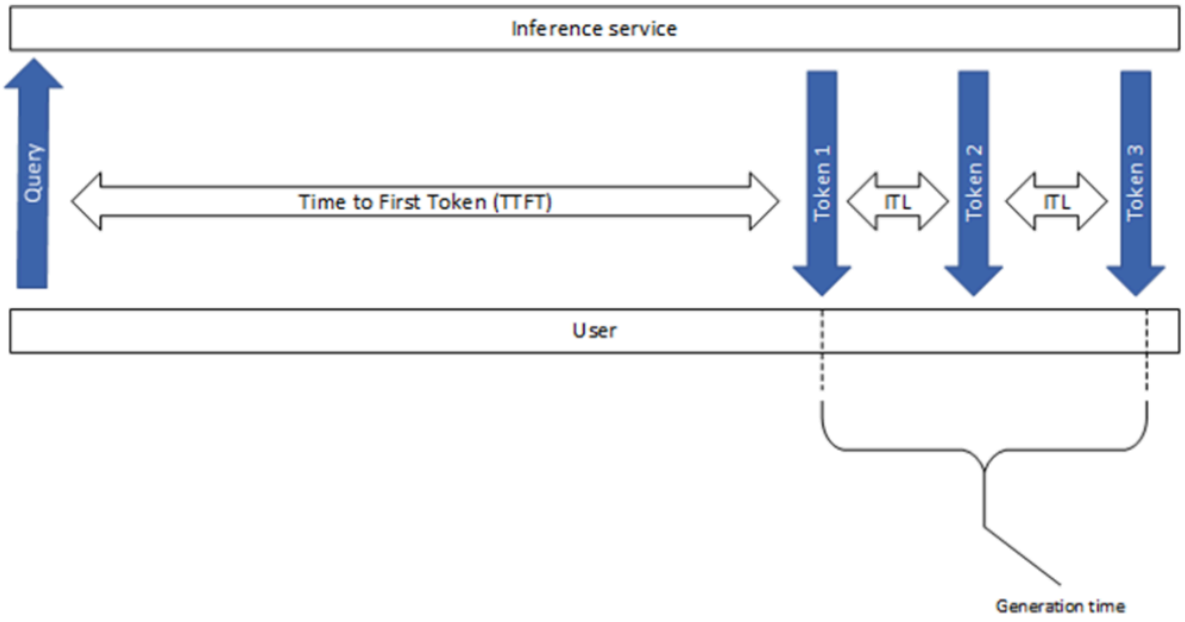
Time to First Token (TTFT): This metric calculates how long a user has to wait after entering a prompt and before the first token is received as a response from the inference service.

Output Tokens Per Second Throughput: This metric shows all the tokens generated by the Inference Service per second for every request.

Inter Token Latency (ITL): Average time taken between consecutive tokens generated

Concurrent Users (CCU): Total number of active users concurrently being served by the inference service

Figure 16. Performance Metrics to monitor for an LLM Inference



Scale Methodology for NIM LLM with Run:ai

Determining the appropriate scaling strategy for an LLM service depends on several key factors, including the expected number of concurrent users, acceptable peak-time performance, and organizational constraints such as budget and system tolerances. Two primary metrics drive this assessment: **user latency** and **throughput**.

- **User Latency** refers to the time a user waits for a request to be processed, typically measured as **Time to First Token (TTFT)** in LLM services.
- **Throughput** represents the total number of requests the system can handle within a given timeframe, regardless of the number of concurrent users.

To establish an optimal scaling approach, we evaluate how many concurrent users can issue queries simultaneously without a significant increase in latency or degradation in throughput. For this study, we set a performance threshold ensuring that user latency remains close to **1000ms** for all prompt requests. We then determine the maximum number of users a specific model can support, given the minimum number of GPUs needed to run for a particular characteristic, like chat, summary, translation, etc. For Meta Llama 8B, a single Pod needs at least one H100 NVL GPU. Scaling is achieved by increasing the number of pods running the **NIM LLM** within the Kubernetes cluster and analyzing the impact on user capacity as additional GPUs are consumed and as more LLM pods are scaled. With Run:ai, we are just increasing the number of replicas needed to run a LLM service, starting one replica per LLM service. A single replica is consuming 1 -2 GPUs based on the model's profile and Memory requirement.

This methodology enables precise scalability planning, ensuring that AI infrastructure can dynamically adjust to varying workload demands while maintaining an optimal balance between performance and resource utilization. With Run:ai, this can be automated. Run:ai lets the user determine at what point to scale up or scale down resources based on Throughput, Latency, or Concurrent users. This can be defined while creating the Inference Workload.

GenAI-Perf was used to load the NIM Service with multiple users, using a summarization characteristic, where ISL/OSL is set to 2000:200. Once GenAI-Perf runs a workload against the NIM service it provides metrics to check the throughput, Latency and the Time it took the model to output the first token (TTFT) etc.

Table 4. Models, Scale, and Use Cases tested

LLM	With Run:ai	GPU	GPU Scale Tested	Use case	Quantization tested
Meta Llama 3.1 8B Instruct	No	H100 NVL	From 1 Pod X 1 GPU to 64 Pods X 1	Summarization (2000:200)	fp8
DeepSeek R1 Distill Llama 8B	No	H100 NVL	From 1 Pod X 1 GPU to 64 Pods	Summarization (2000:200)	fp8

			X 1 GPU		
Meta Llama 3.1 8B Instruct	Yes	H100 NVL	From 1 Pod X 1 GPU to 64 Pods X 1 GPU	Summarization (2000:200)	fp8
DeepSeek R1 Distill Llama 8B	Yes	H100 NVL	From 1 Pod X 1 GPU to 64 Pods X 1 GPU	Summarization (2000:200)	fp8
Meta Llama 3.1 8B Instruct	Yes	H100 NVL	From 1 Pod X 0.5 GPU to 128 Pods X 0.5 GPU	Summarization (2000:200)	fp8
DeepSeek R1 Distill Llama 8B	Yes	H100 NVL	From 1 Pod X 0.5 GPU to 128 Pods X 0.5 GPU	Summarization (2000:200)	fp8
Mixed Workloads Meta Llama 3.1 8B & Deepseek R1 Distill Llama 8B	Yes	H100 NVL	From 1 Pod X 1 GPU to 128 Pods X 0.5 GPU (each LLM)	Summarization (2000:200)	fp8

Installing and Configuring Gen-AI Perf

Triton Inference Server can be installed as a Docker container or a Pod in a Kubernetes Cluster. You can get the server from NGC here. Triton Inference Server has the GenAI Perf tool in it.

To run on Docker,

```
docker pull nvcr.io/nvidia/tritonserver:24.12-trtllm-python-py3
```

To run on Kubernetes, you can deploy it using this YAML file, make sure the secret to NGC is also created in the cluster, as described in the NIM Service creation steps above.

```
Kubectl apply -f triton-server.yaml
```

```
apiVersion: v1
kind: Pod
metadata:
  name: triton
  labels:
     app: triton
spec:
  containers:
  - name: triton
     image: nvcr.io/nvidia/tritonserver:24.10-py3-sdk
     command: ["sleep", "infinity"]
     volumeMounts:
     - mountPath: "/mnt"
     name: config-volume
  volumes:
     - name: config-volume
     hostPath:
     path: /tmp
     type: Directory
```

Once you have the docker container or the Kubernetes pod, shell into it to run the genai-perf command.

The model name to run the GenAI Perf tool can be found by this command. Run this against the NIM LB Service, the IP in the command below is the IP of the NIM load balancer.

```
curl http://{ip}:{port}/v1/models


mrawat@pdxera-bcm01:~$ curl http://10.184.178.83:8000/v1/models


{"object":"list","data":[{"id":"mistralai/mixtral-8x7b-instruct-v0
.1","object":"model","created":1737586179,"owned_by":"system","roo
```

```
t":"mistralai/mixtral-8x7b-instruct-v0.1","parent":null,"max_model
_len":32768,"permission":[{"id":"modelperm-8d273b12ab3541c6807170b
0752429f0","object":"model_permission","created":1737586179,"allow
_create_engine":false,"allow_sampling":true,"allow_logprobs":true,
"allow_search_indices":false,"allow_view":true,"allow_fine_tuning"
:false,"organization":"*","group":null,"is_blocking":false}]}]}
```


More on GenAI-Perf can be found here , below is the command used to run load using GenAI Perf

```
     genai-perf profile \
          -m $MODEL \ #make sure put model name with quote ` `
          --endpoint v1/chat/completions \
          --endpoint-type chat \
          --service-kind openai \
          --streaming \
          -u http://<IP of Nim-Service>:8000 \
          --num-prompts 100 \
          --synthetic-input-tokens-mean $inputLength \
          --synthetic-input-tokens-stddev 50 \
          --concurrency $CONCURRENCY \
          --extra-inputs max_tokens:$OUTPUT_SEQUENCE_LENGTH \
          --extra-input ignore_eos:true \
          --profile-export-file
test_chat_concurrency${concurrency}_input${input_tokens}_output${ou
tput_tokens}.json
```

Inference Performance and Scale Results with Run:ai

Benchmarking without Run:ai

To determine the baseline performance of the two models we ran a single GPU benchmarking test with Gen-AI Perf, we create a NIM LLM service without Run:ai with a single pod and single GPU, incrementally increased the concurrent users loaded by Gen-AI Perf registering the TTFT and Throughput until we saw the TTFT coming close to 1000ms. We registered the concurrent users count at which this was achieved and then scaled the GPUs consumed by the NIM LLM Service by scaling the Pods incrementally by 16X, 32X, and 64X. We noted the Baseline and scale numbers. Based on the results, the no. of Concurrent users scaled linearly as we scaled the GPUs consumed by the NIM Workloads.

Below are the Baseline performance numbers recorded for each model for Single GPU

Table 5. Single GPU Performance Numbers without Run:ai

NIM LLM	GPU/Pod	Pods	CCU	Throughput (Output Tokens/Sec)	TTFT (ms)
Meta Llama 3.1 8B	1	1	141	3059	971
Deepseek R1 Distill Llama 8B	1	1	135	3117	976

Benchmarking with Run:ai using a full GPU

Once we had baseline performance numbers without Run:ai scheduling the workloads on the cluster, we created a NIM Inference workload using Run:ai, ran the same test on a single GPU managed by Run:ai and then scaled the workloads to 16X, 32X and 64X. This would

max out all the GPUs in the cluster. Gen-AI Perf tool was used to load as many concurrent users on the NIM service to reach a TTFT of close to 1000ms.

The results in terms of the total concurrent users and the Throughput matched or was close to that of the tests without Run:ai schedular. See below table:

Table 6. Single GPU Performance Numbers with Run:ai

NIM LLM	GPU/Pod	Pods	CCU	Throughput (Output Tokens/Sec)	TTFT (ms)
Meta Llama 3.1 8B	1	1	137	2977	987
Deepseek R1 Distill Llama 8B	1	1	141	3225	987

Benchmarking with Run:ai using a Fractional GPU

Run:ai supports allocating fractional GPUs to AI workloads, however when an AI workload runs and is requesting a fractional GPU, if there are no other workload competing for the fractional GPU, Run:ai can allocate the entire GPU to the workload dynamically, for the next set of tests we ran the same NIM using Run:ai, however, we allocated only half(0.5) the GPU of H100NVL to the NIM pod. We then ran the same Gen-AI perf tests on the NIM service and scaled the pods to 16x, 32X and 64X. Each Pod consumed 0.5 GPU and at 64X scale there was 1:1 mapping to pods and GPUs in the cluster.

Since there were no other competing workloads, the dynamic GPU allocation in Run:ai gave the pods the entire GPU, and hence the perf results for the fraction (0.5) GPU are comparable to the NIM running with a full GPU.

Table 7. Single GPU Performance Numbers with Run:ai fractional (0.5) GPU

NIM LLM	GPU/Pod	Pods	CCU	Throughput (Output Tokens/Sec)	TTFT (ms)
Meta Llama 3.1 8B	0.5	1	137	3086	995
Deepseek R1 Distill Llama 8B	0.5	1	141	3237	978

Performance of NIM LLM at Scale with Run:ai

The Below charts show how the NIM LLM scales with and without Run:ai (both fractional and full GPUs). The concurrent users and throughput scales linearly. At Max scale (64 GPUs) there was a slight decrease in performance of the NIM LLM for the 0.5 fractional GPU, it was around ~10% drop in total concurrent users supported and ~20% drop in throughput.

Below are the Perf numbers for Meta Llama 3.1 8 B. We ran the same test on Deepseek R1 Distill Llama 8B; the results were similar.

Figure 17. Concurrent Users scale for Meta Lama 3.18B

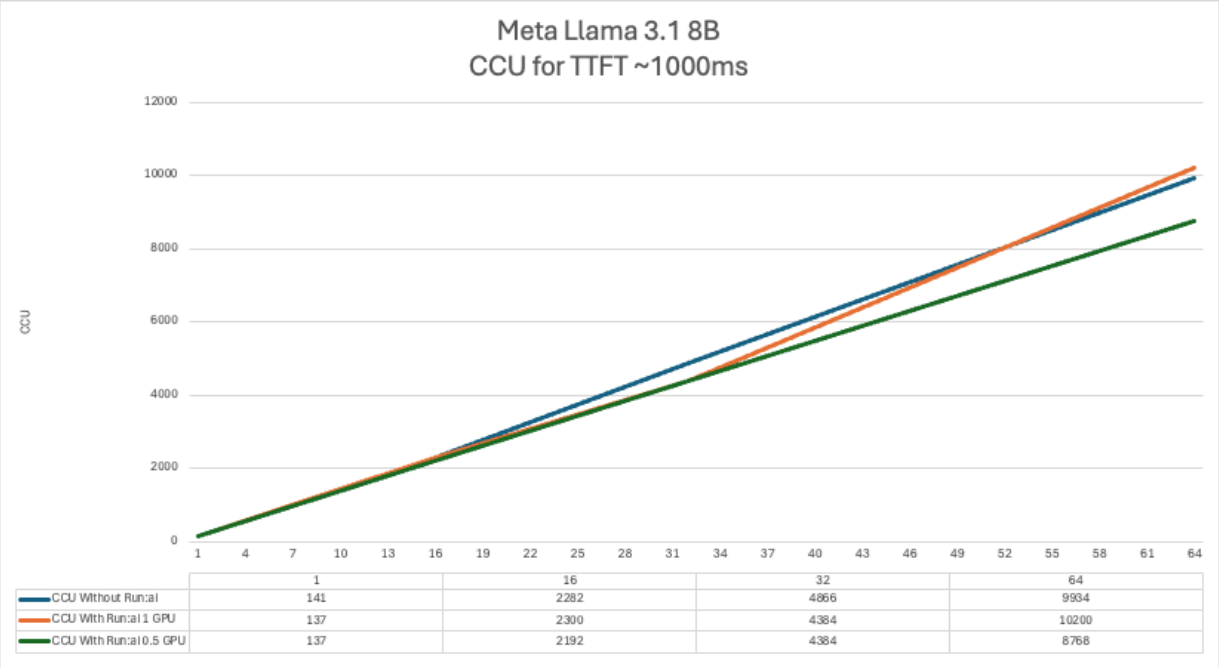
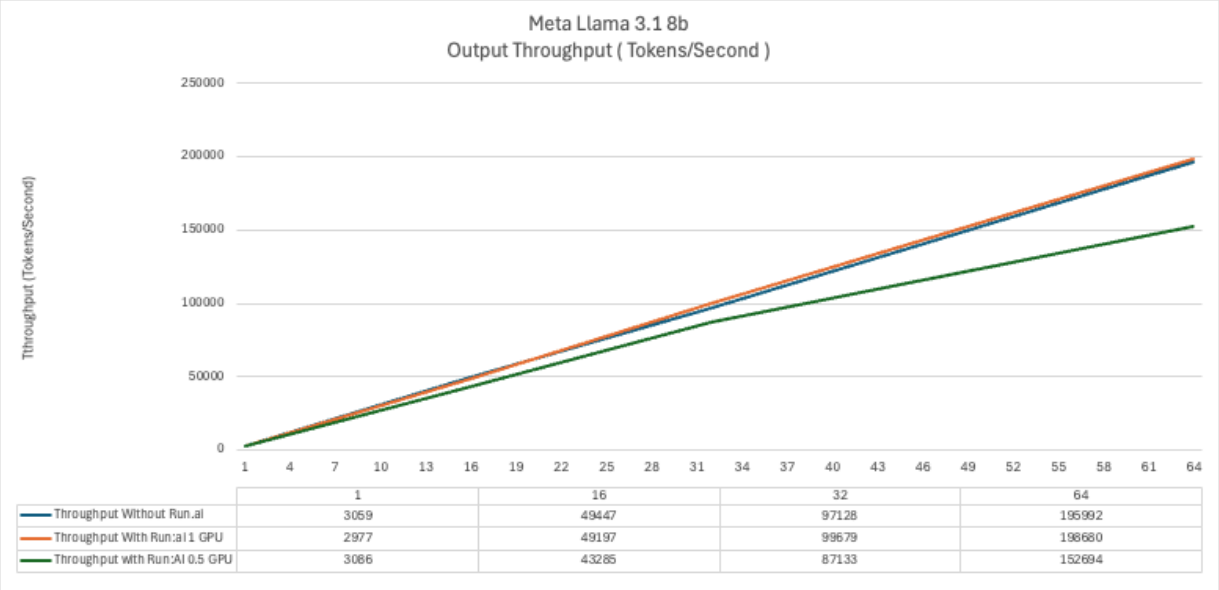


Figure 18. Throughput scale for Meta Lama 3.18B



	1	16	32	64
Throughput Without Run.ai	3059	49447	97128	195992
Throughput With Run:ai 1 GPU	2977	49197	99679	198680
Throughput with Run:AI 0.5 GPU	3086	43285	87133	152694

Simultaneous Multiple NIMs with Run:ai

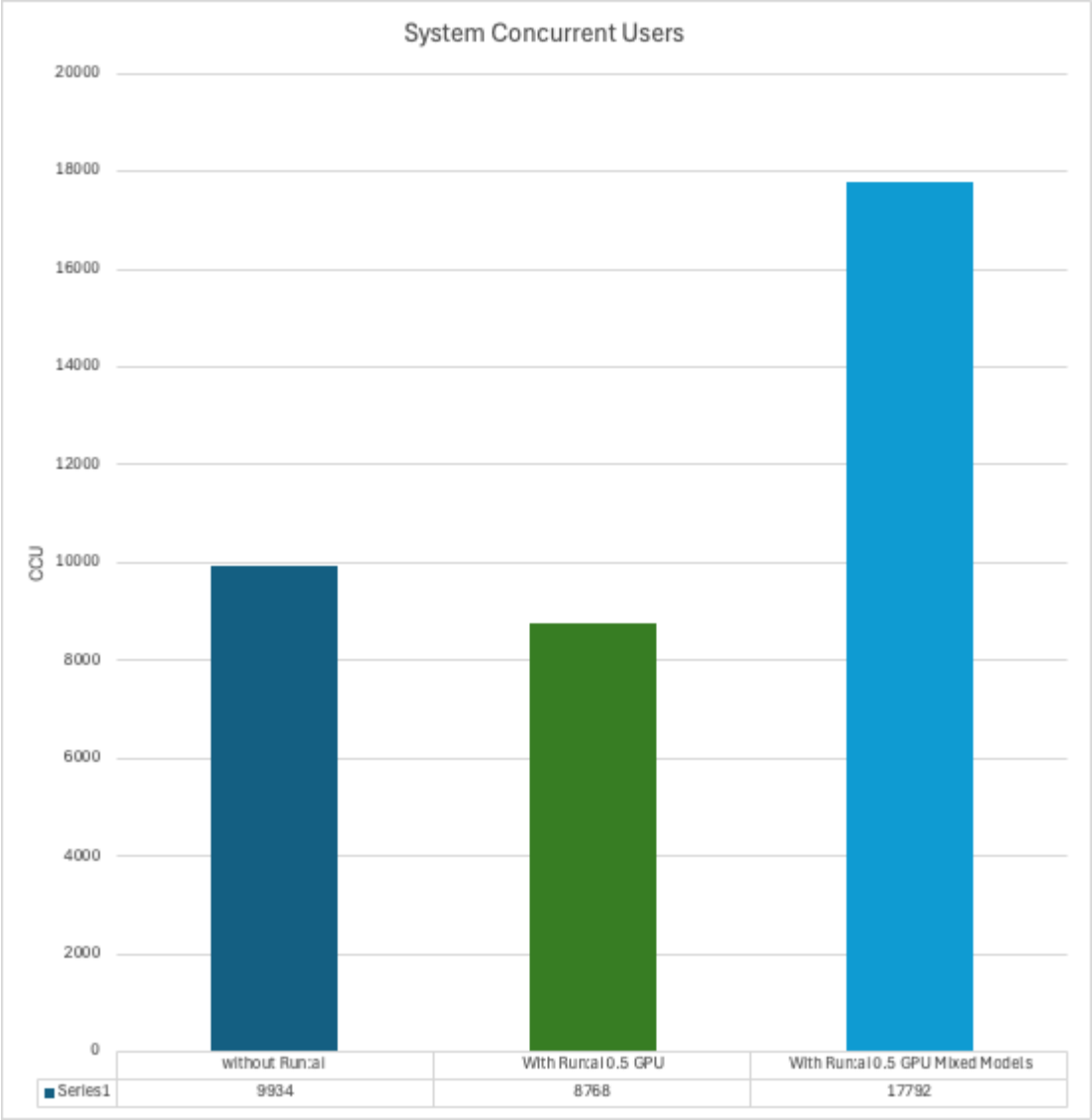
So far, we have run individual NIMs with different models on the cluster. Since Run:ai supports running AI workloads on fractional GPUs, we load both the NIMs, Meta Llama 3.1 8B and Deepseek R1 Distill Lama 8B simultaneously using Run:ai and gave 0.5 /half GPU to each. The cluster was then targeted by Gen-AI Perf to run workloads with the same number. In the CCU, we registered for individual runs of the respective models. The individual NIM workload was incrementally scaled along with the NIM Service pods to 16X, 32X, and 64X to completely utilize all the 64 GPUs in the cluster. Each Pod consumed 0.5 GPU and at 64X scale there was 2:1 mapping of pods and GPUs in the cluster.

Both the Inference services scaled linearly with Run:ai , both the Concurrent users and Throughput scaled Linearly

With Run:ai's scheduler we were able to load twice as many users on the overall system, during peak workload where 128 pods (64 pods for each NIM service) were requesting 0.5 GPU each, maxing out the 64 GPUs, there was 3X drop in TTFT and around 0.8X drop in the total Throughput the cluster cloud handle.

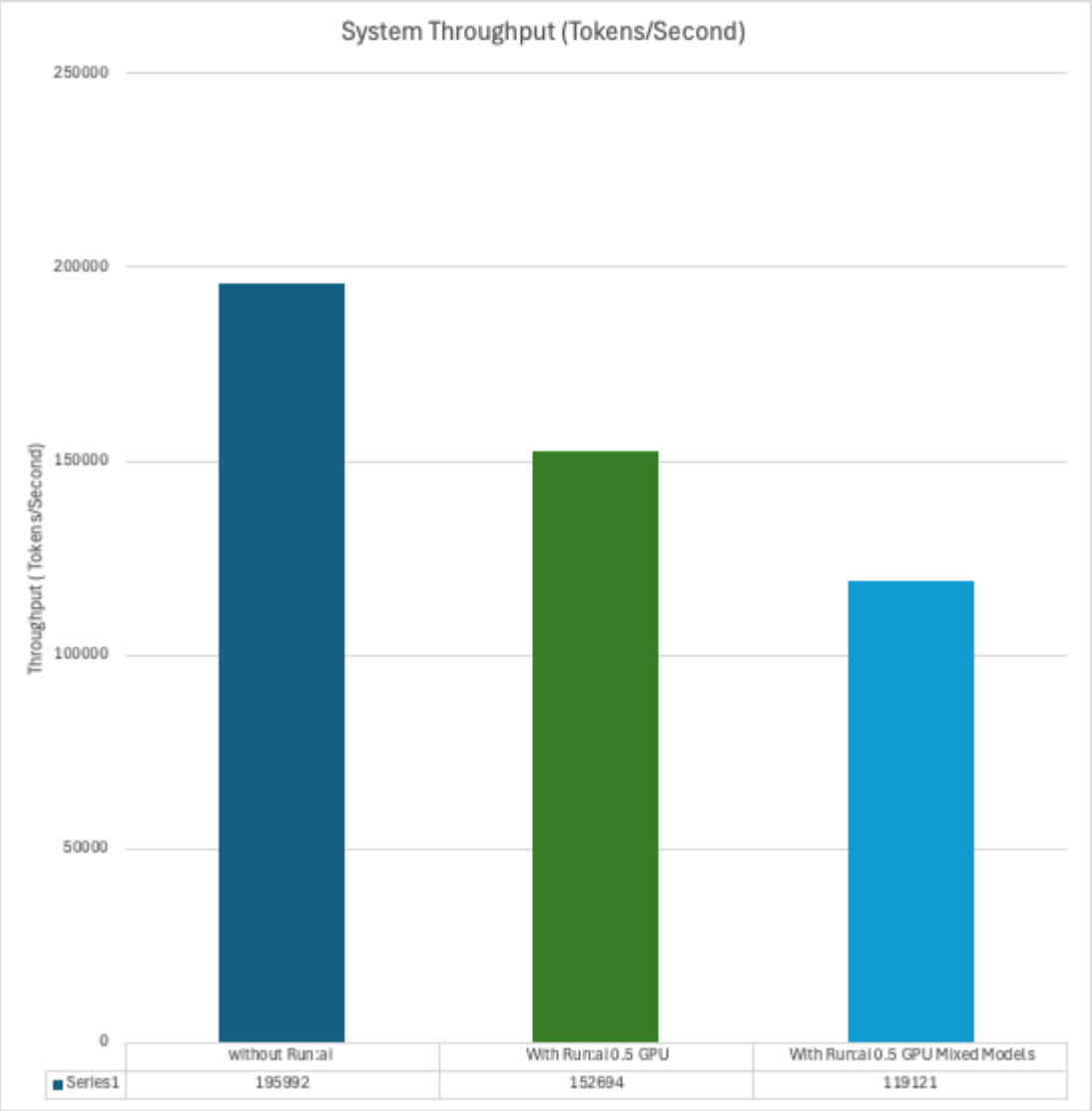
The overall users we could load on the cluster were 2X more than what we could load with individual models when all the GPUs were consumed.

Figure 19. Concurrent User on Meta Llama 3.1 8B across the system



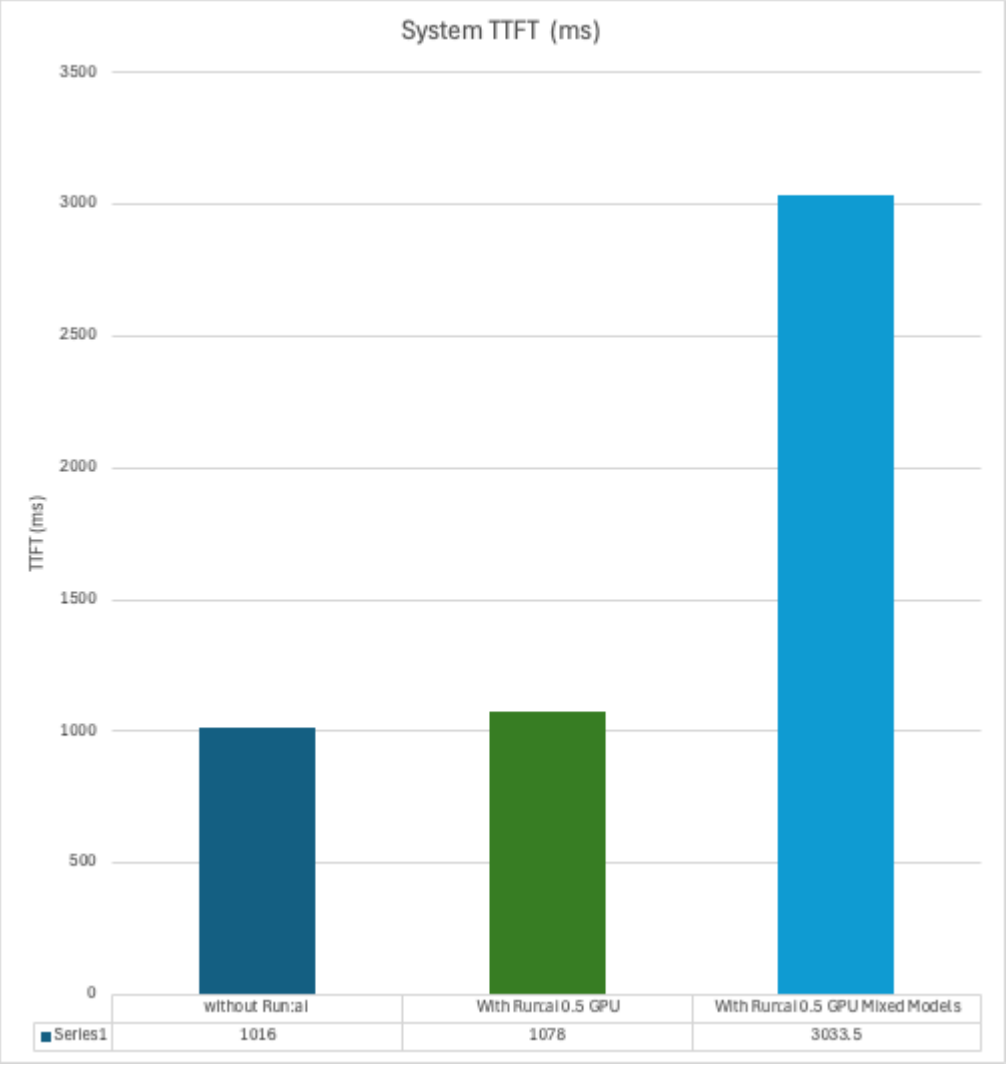
The System Throughput, or the total tokens processed by the cluster at max capacity, dropped by 0.4X compared to the total system throughput when individual NIMs were run at max GPU utilization.

Figure 20. Throughput on Meta Llama 3.1 8B across the system



The overall User Latency, or TTFT, increased by 3X at the max GPU utilization; however, comparatively, the total no. of users that could be loaded and could run Inference on multiple models doubled.

Figure 21. TTFT on Meta Llama 3.1 8B across the system



Sizing Guidelines

The above scaling methodology provides a framework on how to scale NIM LLMs on a system based on user response wait times under common input/output sequence length. Organizations can use this framework to run tests on their stack with workload characteristics that are more specific to their users and determine what CCU values they are getting for a single GPU, Fractional GPU and running multiple models simultaneously. Based on the results and expected peak load they can determine how many GPUs will be needed and correspondingly they can size the servers needed to hold that many GPUs. Organizations can also use this framework to determine scale and sizing for other LLM NIMs.

For example, For this version of the paper, below table would determine the concurrent users that run simultaneously using Run:ai on a **single NVIDIA H100 NVL GPU, precision of fp8, characterization of 2000:200 on the** 2-4-3-200 **Enteprise RA based cluster** and yet maintain a **TTFT of close to 1000ms.**

Table 8 . Sizing GPUs for NIM Workloads with Run:ai

Model	CCU	Throughput (Output Tokens/Sec)	TTFT (ms)
Meta Llama 3.1 8B	137	2977	987
Deepseek R1 Distill Llama 8B	141	3225	987
Mixed (Both the above models)	278	6235	982

Summary

NVIDIA Run:ai can be used to dynamically allocate GPUs to AI workloads in a Kubernetes Cluster. Run:ai can automatically scale up or scale down workload pods in a cluster, thereby releasing the amount of GPUs consumed. These free GPUs can then be assigned to other users/projects/workloads to drive the overall utilization of existing GPUs and datacenter resources. Run:ai can help scale NIMs based on concurrent users, user latency, and throughput. Below are the observations and conclusions for running NIM workloads at scale on a 16-node cluster with Run:ai

- The Run:ai scheduler does not add any additional performance overhead to Inference workloads when running on full GPUs. The performance data for NIMs running with and without Run:ai are comparative
- NIM scheduled on Fractional (0.5) GPUs
    - Scaled Linearly
    - Performance of CCU, TTFT, and Throughput was comparable to non-Run:ai tasks til 32X Scale. This is because if there is no other competing workload on that GPU, Run:ai will dynamically expand the GPU to full for the workload.
    - At peak scale, consuming all the 64 GPUs in a cluster, the no. of concurrent users and throughput recorded were slightly lower, the Throughput dropped by ~20%, and the Concurrent Users dropped by ~10%
- Enterprise IT can use Run:ai to load more than one NIM for the same number of GPUs and get a variety of models to run, as well as twice as many users, decreasing the overall TCO. However, at full scale 64X, the TTFT can drop by 3X and the Throughput by 0.4X.

Appendix A

Installing Pre-reqs for RunAI

Get the Run.ai SaaS Login

The run.ai portal where the control pane is required a login to be created by NVIDIA. Please work with Account/SA teams to get an org craved out for this.

Installing Nginx

```
helm repo add nginx-stable https://helm.nginx.com/stable
helm repo update
helm install nginx-ingress nginx-stable/nginx-ingress --set rbac.create=true
```

Installing Prometheus

Note: The default Prometheus enabled during creating the k8s cluster via BCM does not enable the prometheus operator in the monitoring namespace and that's what Run:ai expects it to be at.

```
helm repo add prometheus-community
https://prometheus-community.github.io/helm-charts
helm repo update
helm install prometheus prometheus-community/kube-prometheus-stack -n monitoring
--create-namespace --set grafana.enabled=false
```

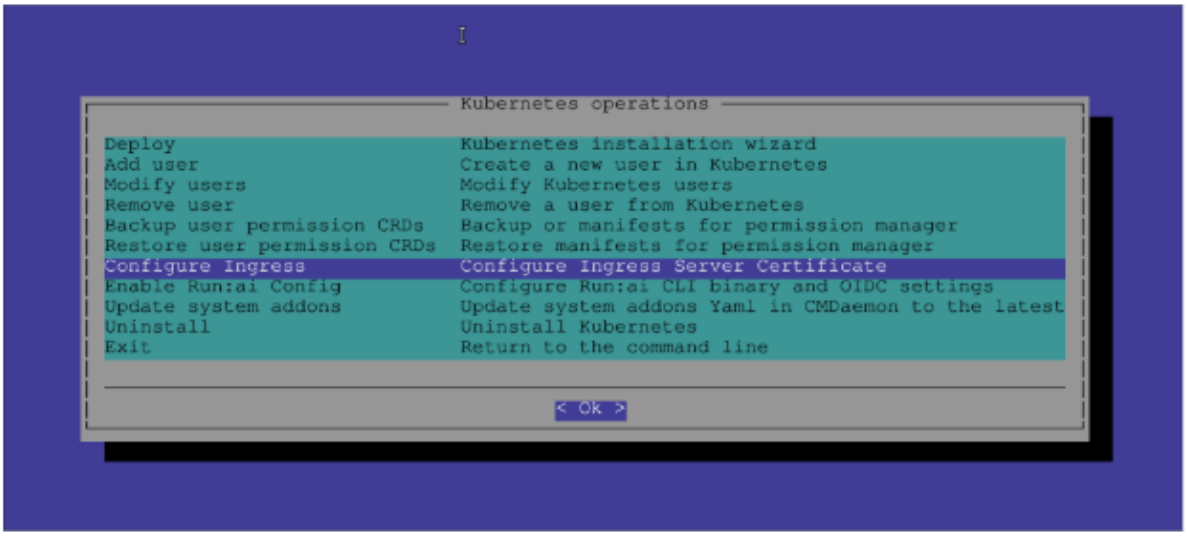
Create Certs and Private Keys

Make sure to create it with the url that is going to be the cluster URL, replace era.nvidia.com with the URL you will be using

```
# Generate a Private Key (2048-bit RSA)
openssl genpkey -algorithm RSA -out era.nvidia.com.key -pkeyopt
rsa_keygen_bits:2048

# Generate a Self-Signed Certificate (Valid for 1 year)
openssl req -x509 -new -key era.nvidia.com.key -out era.nvidia.com.crt -days 365
-subj "/CN=era.nvidia.com"
```

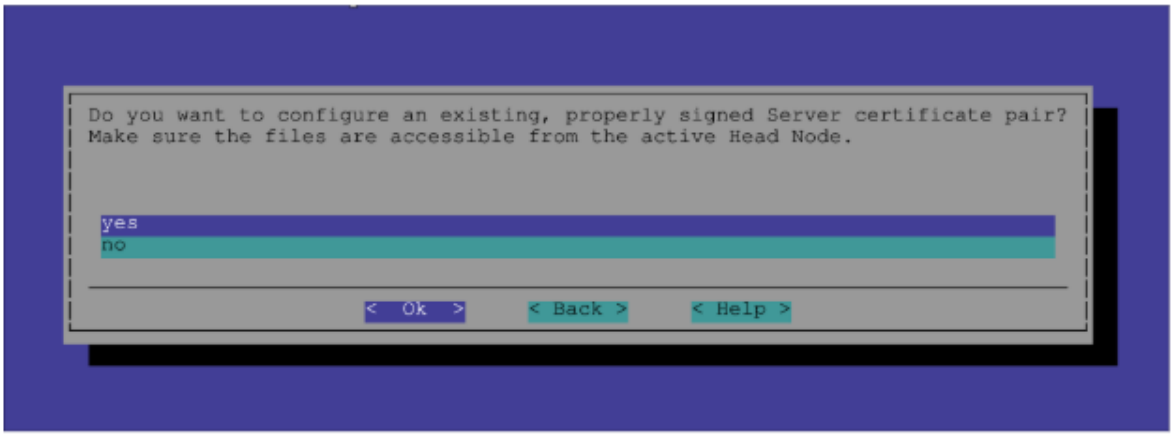
Update BCM Ingress with CA Certificates

Run the `cm-kubernetes-setup` CLI command on the BCMe head node, select `Configure Ingress`

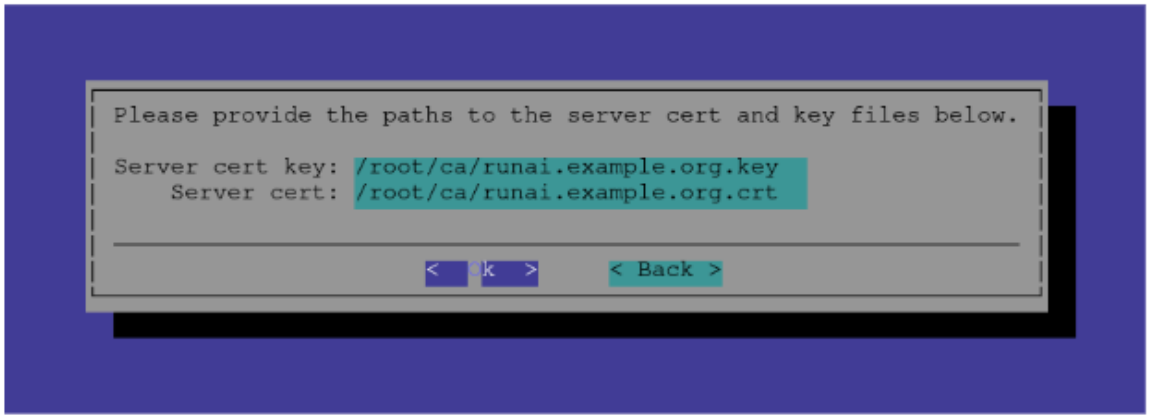


Select the cluster you will use for Run:ai and select "yes", since we do want to configure an existing, properly signed certificate pair

Press `ENTER` to proceed



Set the path to the key and Certs in the next screen, click ok



Expose Ingress Controller to use Public IP from the Metal LB pool

The ingress so far is on private Cluster network, we will expose it to public network so that we can use the Run:ai UI through Ingress's public IP.

```
kubectl patch svc ingress-nginx-controller  -n ingress-nginx  -p '{"spec": {"type": "LoadBalancer"}}'
```

Update DNS Server

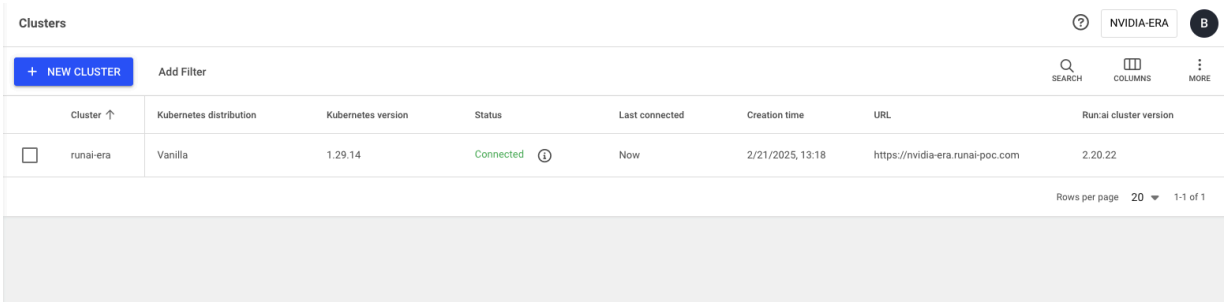
Configure your DNS to map the IP of the Ingress Controller to the Service IP the Ingress Controller gets, the FQDN should point to the IP, so if you plan to use for e.g, `runai.nvidia.local` than the DNS and the Certificates should map the DNS `runai.nvidia.local` to the IP of the Nginx External IP, you you can check the External IP the Ingress gets by running

```
kubectl get svc -A
```

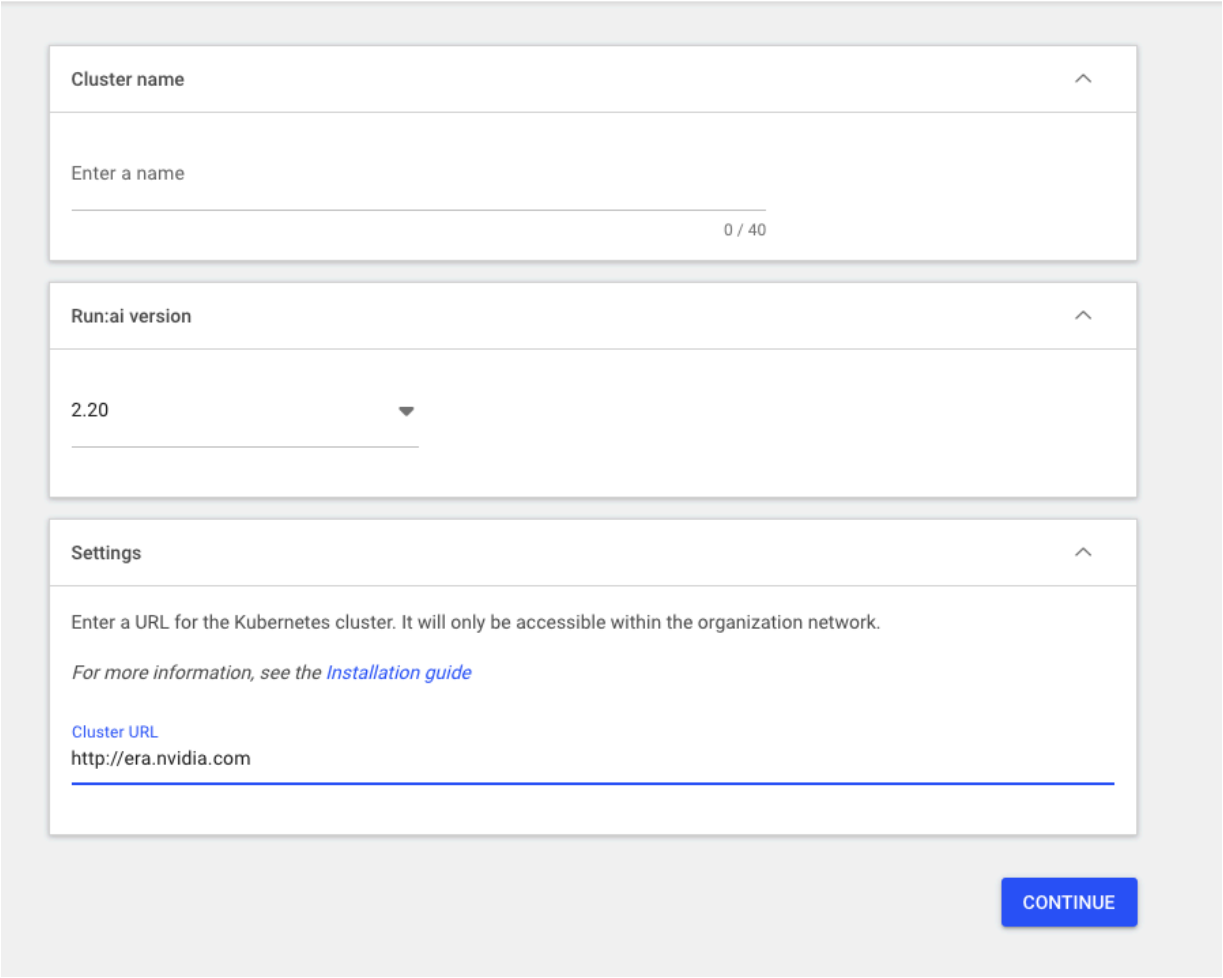
Configure Run:ai

Configure the addition of a cluster to Run.ai SaaS

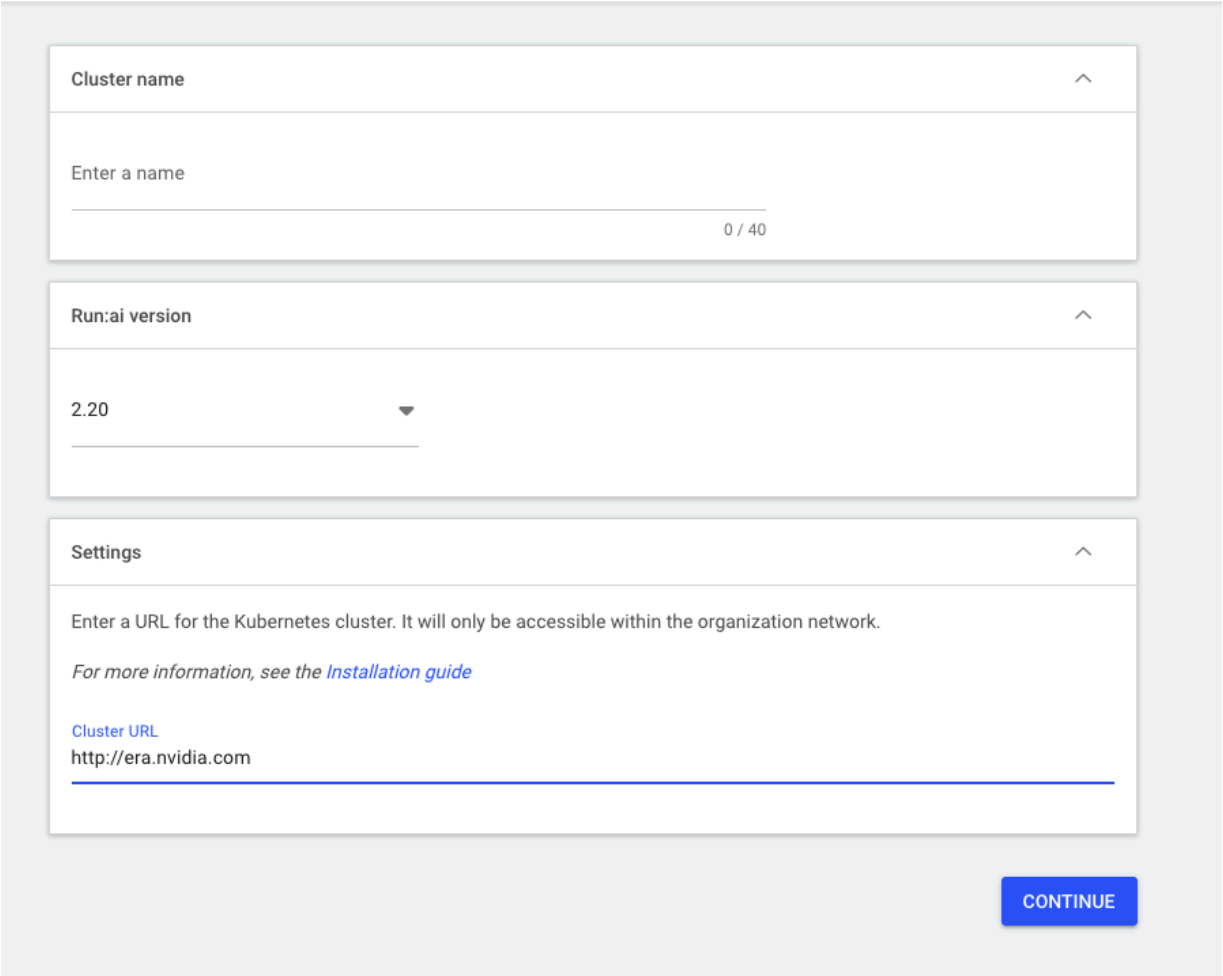
Login to the run.ai SaaS instance provided to you by NVIDIA, click on Resource, and click `New Cluster` button.



Give the cluster a name, let the default version, and enter the URL of the Kubernetes Cluster (Note: This should be the same URL that the certs were created in the steps above, and the certificates)



Capture the instructions and run the next command.



Install Run:ai Cluster

```
kubectl create ns runai
kubectl create secret tls runai-cluster-domain-tls-secret -n runai --cert
/root/ca/era.nvidia.com.crt --key /root/ca/era.nvidia.com.key
 helm repo add runai https://runai.jfrog.io/artifactory/api/helm/run-ai-charts
--force-update
helm repo update
helm upgrade -i runai-cluster runai/runai-cluster -n runai --set
controlPlane.url=nvidia-era.run.ai --set
controlPlane.clientSecret=JpgdoQajWYHdCoN2YgQuolXFD5wnaL1D --set
cluster.uid=1100804e-0bd4-4c11-9f15-0d0b8231ccca --set
cluster.url=http://era.nvidia.com --version="2.20.20" --create-namespace
```

Wait for a few minutes till the cluster syncs with Control-Plane

Install Knative for Inference Workloads

```
kubectl apply -f
https://github.com/knative/serving/releases/download/knative-v1.17.0/serving-crds.
yaml
kubectl apply -f
https://github.com/knative/serving/releases/download/knative-v1.17.0/serving-core.
yaml
```

Configure Knative to use with Run: ai

```
kubectl patch configmap/config-autoscaler \
  --namespace knative-serving \
  --type merge \
  --patch '{"data":{"enable-scale-to-zero":"true"}}'

kubectl patch configmap/config-features \
  --namespace knative-serving \
  --type merge \
  --patch
'{"data":{"kubernetes.podspec-schedulername":"enabled","kubernetes.podspec-affinit
y":"enabled","kubernetes.podspec-tolerations":"enabled","kubernetes.podspec-volume
s-emptydir":"enabled","kubernetes.podspec-securitycontext":"enabled","kubernetes.c
ontainerspec-addcapabilities":"enabled","kubernetes.podspec-persistent-volume-clai
m":"enabled","kubernetes.podspec-persistent-volume-write":"enabled","multi-contain
er":"enabled","kubernetes.podspec-init-containers":"enabled"}}'
```

Configure HPA for Autoscaling by Knative

```
kubectl apply -f
https://github.com/knative/serving/releases/download/knative-v1.17.0/serving-hpa.y
aml
```

Update Knative timeout

For larger Inference Workloads, we need to increase the timeout value knative uses, else when a large model is deployed , the model will take a long time to download and run, knative

will think the workload isn't processing and timeout. In order to avoid this increase the time knative uses to wait for workloads to come up. The below will change it to 30 minutes.

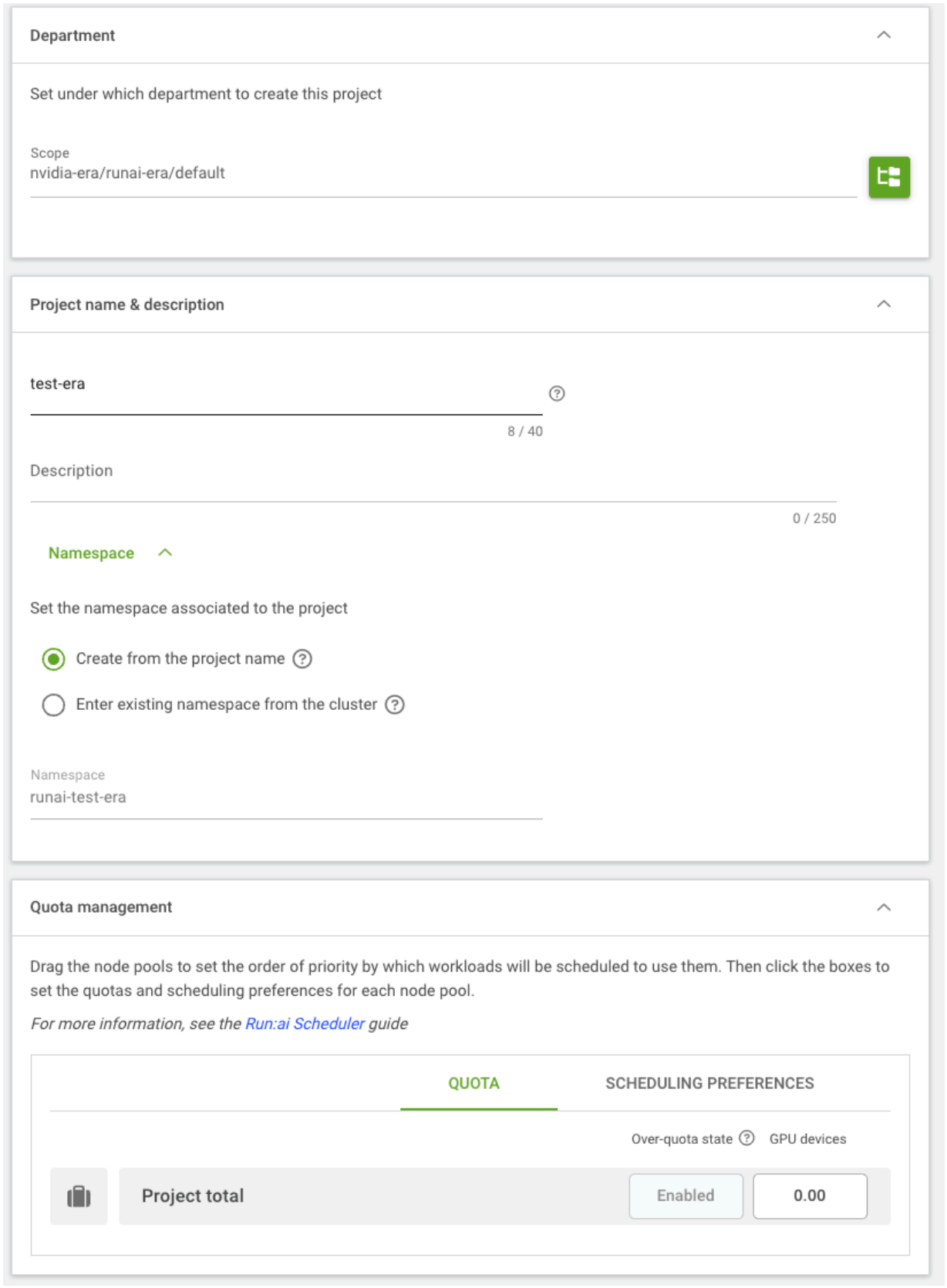
```
kubectl patch ConfigMap config-deployment -n knative-serving --type='merge' -p
'{"data": {"progress-deadline": "1800s"}}'
```

Create a Project in Run:ai

Once the Cluster is connected to the Run:ai control plane, we need to create a Project in Run:ai , a project is a logical separation of resources, once a project gets created Run:ai adds its custom schedular in the namespace ont he Run:ai cluster defined during the creation of the project, this project is where all the workloads that are submitted will be scheduled by Run:ai.

Go to Run:ai UI → Click on `Organization` from the Left Hand Menu → Select “ +New Project”

Select the scope for the project, this could be org wide, enter the Name for the Project and the namespace the Kubernetes Cluster to be where Run:ai deploys its schedular,you can select an existing namespace in the cluster or create a new one, provide the GPU quota, this would be the max no. of GPUs to allocate from the pool to that project.


Department
Set under which department to create this project
Scope
nvidia-era/runai-era/default
Project name & description
test-era
8 / 40
Description
0 / 250
Namespace
Set the namespace associated to the project
Create from the project name
Enter existing namespace from the cluster
Namespace
runai-test-era
Quota management
Drag the node pools to set the order of priority by which workloads will be scheduled to use them. Then click the boxes to set the quotas and scheduling preferences for each node pool.
For more information, see the Run:ai Scheduler guide
QUOTA
SCHEDULING PREFERENCES
Over-quota state
GPU devices
Project total
Enabled
0.00

Change the Placement Strategy

Run:ai will try to pack as many workload on GPUs in a single worker node before it moves to another node, it will try to pack as much as possible, In order to test NIM scaling we need to change this default behavior to `Spread`, where new workload will be spread across all the nodes and GPUs in the cluster.

In the Run:ai UI→ Go to Resources on the LHS menu → Select Node Pools→ Click on the Node pool used , select Edit and change the Radio Button on `Placement Strategy` to `Spread` for both GPU and CPU and Click `Save` .

Add Users in Run:ai

Login to Run:ai User Interface →On the LHS Menu →Select `Access` →Select `Users`→Click `"New User"`