TECHNICAL WHITE PAPER
September 2025

Deploy Distributed LLM Inference with GPUDirect RDMA over InfiniBand in VMware Private AI

vmware® by Broadcom

Table of contents

1. Introduction

When deploying state-of-the-art reasoning large language models (LLMs) such as DeepSeek-R1 or Meta Llama-3.01-405B-Instruct, the memory and compute requirements often exceed the capacity of a single server with 8x H100 GPUs. In these cases, distributed inference becomes a necessity, allowing resources from multiple GPU-enabled nodes to be aggregated in service of a single model. Distributed inferencing introduces new complexities in distributed infrastructure management, interconnect performance optimization, and workload scheduling.

VMware Cloud Foundation® (VCF) is the industry's first private cloud platform to deliver public cloud scale and agility with on-premises security, resilience and performance, while lowering total cost of ownership. For distributed deployments, NVIDIA NVLink, NVIDIA NVSwitch and GPUDirect® RDMA are critical, as they allow high-bandwidth, low-latency communication between GPUs within and across nodes. VCF ensures network interconnects like InfiniBand (IB) and RDMA over Converged Ethernet (RoCEv2) can be leveraged effectively, reducing communication overhead that often limits distributed inference performance. With VCF, enterprises can enable production-grade distributed inference, ensuring that even the largest reasoning model can be deployed reliably while maintaining predictable performance characteristics.

This paper examines the various components of the technology stack that make distributed inference feasible via Dynamic DirectPath I/O on VCF, outlines the architectural considerations, and provides technical guidance required to effectively operate LLMs across multiple GPU nodes.

2. Leverage HGX servers for maximum performance

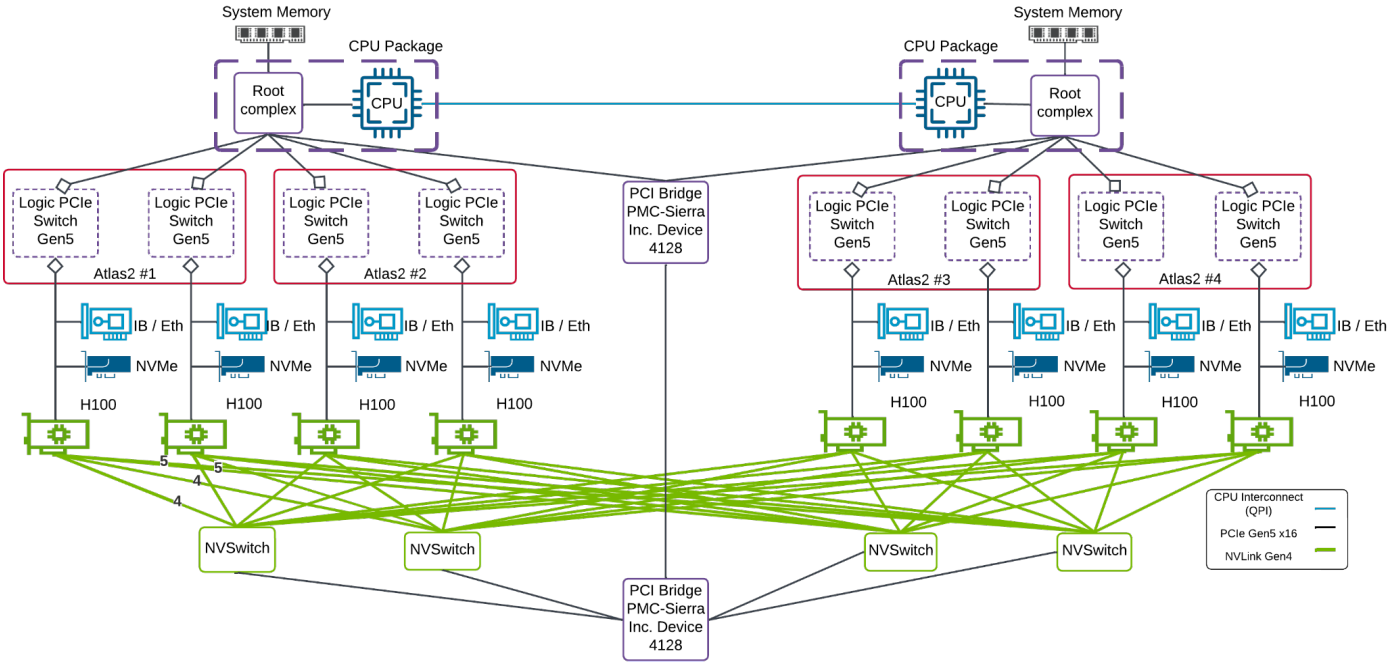
When preparing for distributed inference, it is critical to validate the detailed specifications of the HGX servers in use. Variations in interconnect topology (NVLink vs. PCIe), networking hardware (InfiniBand vs. Ethernet), and auxiliary components (cooling, power delivery, BIOS settings) can significantly impact distributed inference performance and scalability. Ensuring hardware alignment and consistency across nodes is a prerequisite for achieving predictable results in large-scale inference deployments. For architectural and deployment guidance, refer to our Reference Design for Inference for deploying VMware® Private AI Foundation with NVIDIA on NVIDIA HGX servers.

2.1 Typical 8x GPU HGX node

An 8x GPU HGX server, commonly used in VMware Private AI Certified platforms[1], typically contains the following components.

- 4x **Broadcom Atlas 2 (PEX89XXX) PCIe Switches** (synthetic mode enabled, each logically partitioned as 2)
- 8x **NVIDIA H100/H200 GPUs**
- 8x **NVIDIA ConnectX-7 IB HCAs** or **Ethernet NICs**

Figure 1. Topology diagram of a typical HGX server



Note: To achieve optimal **GPUDirect RDMA performance**, assign each GPU and its paired NIC under the same PCIe switch to a VM. A **1:1 GPU-to-NIC ratio** ensures every accelerator has a dedicated, high-bandwidth, low-latency network path. This design becomes critical once workloads exceed a single HGX host's max capacity, where collective operations (all-reduce, all-gather, etc.) dominate performance for LLM inference and training. Sharing NIC bandwidth across GPUs can introduce bottlenecks, but a 1:1 mapping allows parallel, oversubscription-free scaling across nodes.

[1] Refer to the Broadcom compatibility guide (BCG) for details: After clicking the link to the BCG, click **Select Desired Compatibility Guide** → **Platform & Compute** → **Systems / Servers** → under **Features,** select **VMware Private AI,** and then scroll up and click **View Results**.

2.2 Intra-node communication

Within a single HGX node, H100 GPUs communicate via **NVLink** and **NVSwitch**.

- **NVLink** provides point-to-point connectivity with up to 900 GB/s bidirectional bandwidth. Our HGX server utilizes **NV18**. As depicted in Figure 1, each H100 connects to the 2 NVSwitches within its NUMA node via 4 links each, and to the remote NVSwitch via 5 NVLinks, thus totaling 18 NVLinks per H100.
- **NVSwitch** enables all-to-all GPU communication with up to 7.2 TB/s bidirectional bandwidth.

Together, NVLink and NVSwitch orchestrate high-speed communication between the 8 GPUs in a node, enabling efficient inference with libraries such as NVIDIA Collective Communication Library (**NCCL**) and optimized runtimes like vLLM or SGLang or NVIDIA TensorRT-LLM.

2.3 Inter-node communication

While NVLink and NVSwitch deliver extremely fast communication within an HGX node, they are limited to **up to 8 GPUs per chassis**. Scaling inference across nodes requires additional interconnects.

Each HGX chassis typically includes:

- **Low-bandwidth Ethernet NICs** (management or VPC traffic)
- **High-bandwidth NICs** (for GPU-to-GPU communication across nodes)

For the high-bandwidth interconnect between HGX servers, customers can choose between:

- **InfiniBand (IB)**
- **RDMA over Converged Ethernet (RoCEv2)**

Both options enable the low-latency, high-throughput communication required for distributed inference. Our configuration utilized HGX servers equipped with **InfiniBand** HCAs. However, similar performance can be achieved with RoCE; for more information, refer to https://www.vmware.com/docs/paif-hgx-brcm-eth.

2.4 Distributed deployment example

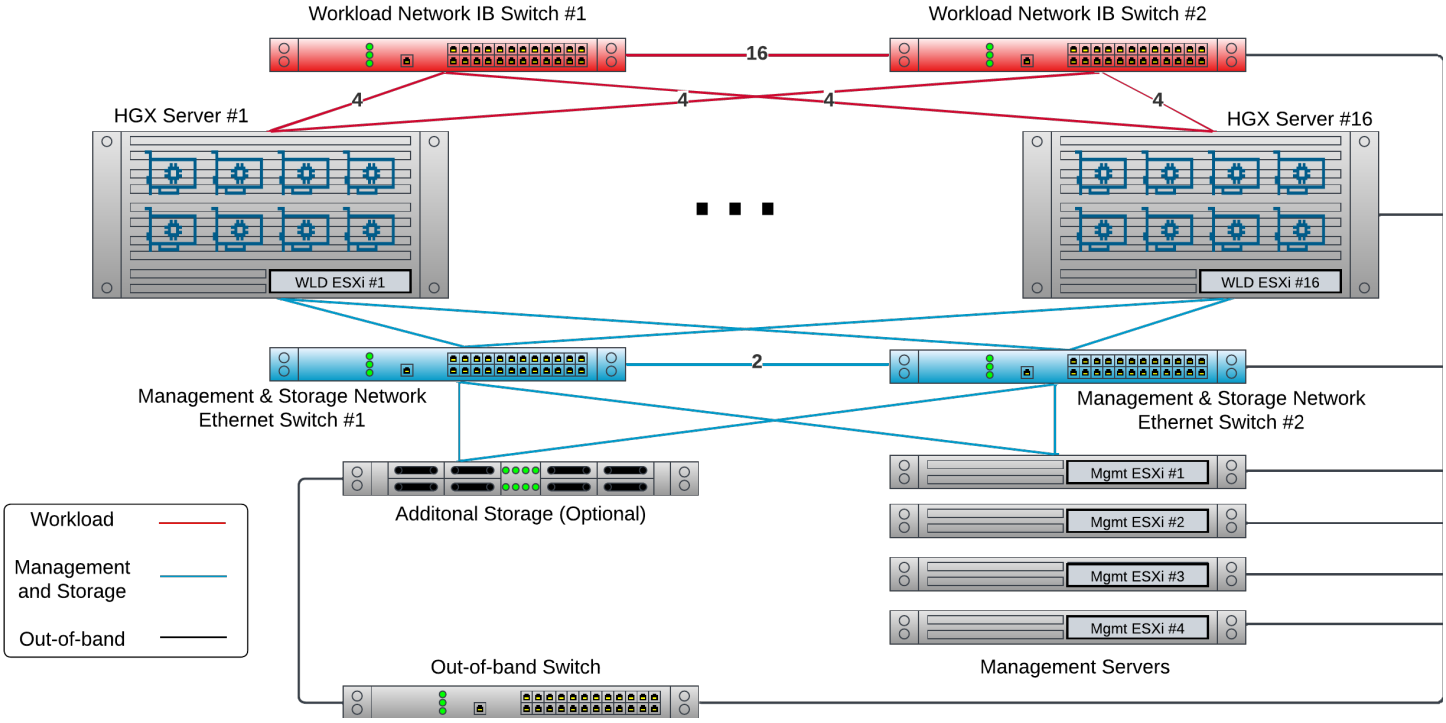
We recommend starting from a **minimum configuration** for a VCF Workload Domain (WLD) of:

- **4 HGX servers**
- **1 workload IB network switch**

Cost optimization: For the first WLD, not all 4 servers must be GPU-enabled. A valid configuration includes **2 HGX servers with GPUs** and **2 standard compute servers (CPU-only)**, which also follows a VCF consolidated architecture design pattern to support distributed LLM inference.

If the IB switch supports a higher port count than 32 per device (radix) or a multi-layer fabric is used, additional HGX servers can be integrated. Figure 2 depicts the InfiniBand connection topology for 16 HGX servers with 2 IB switches.

Figure 2. Physical architecture with 16 HGX servers (IB switch radix = 32)



2.5 Hardware used for deployment

Table 1 shows the hardware used for launching distributed LLM inference in VCF 5.2.1 and 9.0.

Table 1. Hardware components

Hardware	Details	Quantity
Server	Dell PowerEdge XE9680	2x
Processors	Intel Platinum 8470, 52 cores per CPU package	104 logical cores per CPU 2x per server
CPU RAM	32x DDR5 64GB DIMMs	2TB per server
GPUs	NVIDIA H100-80G-SXM	8x per server
NVSwitch	Gen4	4x per server
Storage	Dell (Samsung) Ent NVMe PM1733a RI 3.84TB, Triple-Level Cell (TLC), PCIe 4.0, NVMe 1.3	8x per server
Compute NIC	ConnectX-7 IB HCA	8x per server
Compute Fabric switch	NVIDIA Quantum QM9700 NDR 400Gbps InfiniBand	1x per cluster
Dual Port OSFP transceiver for switch	MMA4Z00-NS	32x per switch
Compute cables	NDR InfiniBand DAC, MCP4Y10	8x per server
Management & storage NIC	Intel Ethernet Controller E810-C 100 Gb/s	2x per server
Management switch	EdgeCore 100GbE	1x per cluster

3. GPUDirect RDMA in VCF

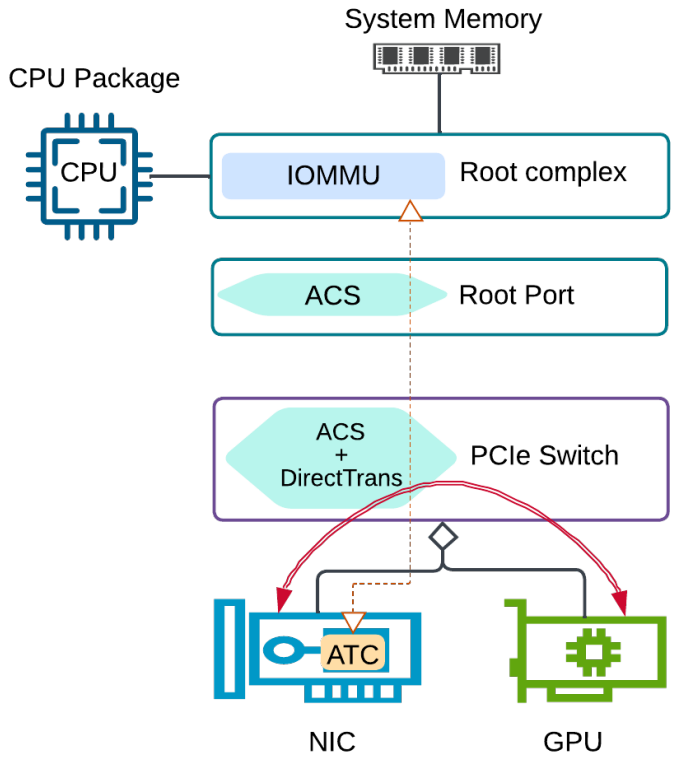
Enabling **GPUDirect RDMA** in VCF requires two key configurations:

1. **Access Control Services (ACS)** must be enabled in ESX.
2. **Address Translation Services (ATS)** must be enabled on the ConnectX-7 (CX-7) NICs.

Within virtualization, the **Input-Output Memory Management Unit (IOMMU)**, implemented via Intel VT-d or AMD I/O Virtualization (IOV) provides each PCIe device with a unique translated virtual address space (IOVA). Meanwhile, ACS is typically configured to redirect all peer-to-peer (P2P) requests and completions to the **Root Complex** for security enforcement. For two devices (e.g., GPU and NIC) to exchange data directly over PCIe, the IOMMU must establish mappings so that each device can issue transactions into the other's PCIe address space.

While ACS offers fine-grained control over PCIe transactions, it can block direct device-to-device communication by forcing all traffic through the root complex—reducing bandwidth and increasing latency.

Figure 3. Illustration of how GPUDirect RDMA works in VCF

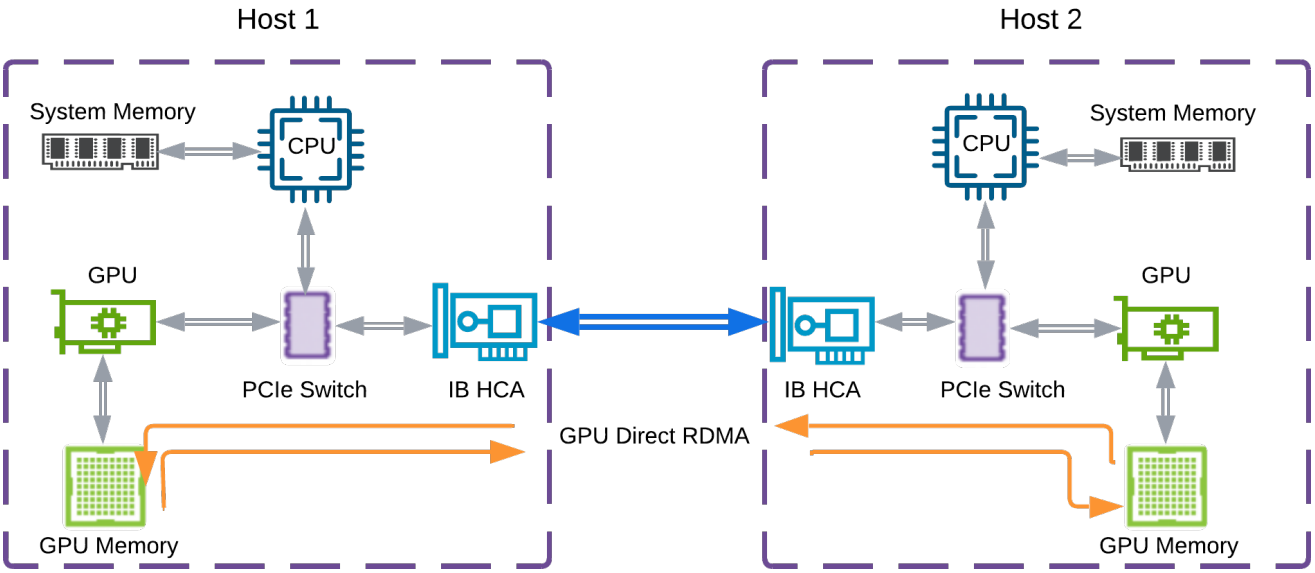


Address Translation Services (ATS), facilitates direct DMA transactions between PCIe endpoints, even when ACS and/or IOMMU are active. As depicted in Figure 3, ATS achieves this by caching translation results in its **Address Translation Cache (ATC)**. This caching allows devices to bypass the root complex and continue direct GPU-to-NIC communication.

Note: VMware has collaborated with hardware vendors to fully enable ATS in ESX, a key PCIe feature that lets devices cache and share virtual-to-physical address translations directly. With ATS function in the hardware, VMware supports not only **GPUDirect RDMA** for GPU-to-NIC transfers but also **GPUDirect Storage (GDS)** and general **PCIe peer-to-peer (P2P)** communication between compliant devices. This ensures efficient, low-latency data movement in virtualized environments, delivering near bare-metal performance for AI, HPC, and data-intensive workloads.

Without ATS, ACS forwarding can cut available bidirectional bandwidth in half or more. With ATS enabled and Direct Translation active on PCIe switches, most traffic flows directly between the GPU and NIC again, restoring the low-latency path required for **GPUDirect RDMA**. As a result, with ACS and ATS properly configured in Figure 4, GPUDirect RDMA can be achieved across VCF hosts.

Figure 4. Conceptual view of GPUDirect RDMA across two hosts



4. Determine the number of HGX servers required for LLM inference

A key question when planning inference at scale is: **What is the minimum number of GPU servers required to serve a given LLM?** The answer depends on both the model architecture and hardware constraints.

One critical factor is the model's `num_attention_heads` parameter (found in the `config.json` in each model's Huggingface repo). In multi-GPU inference, attention heads must be distributed evenly across GPUs. Since each HGX server typically contains **8x H100 GPUs**, the total number of GPUs across all servers must **divide evenly** into the number of attention heads. For example, if a model defines **64 attention heads**, it can be evenly distributed across 16 GPUs (2 HGX servers), but not across 10 or 12 GPUs.

Another determining factor is **context length**. Models with extremely long context windows (e.g., up to 10M tokens) require additional GPU memory and bandwidth, so additional GPU memory and bandwidth are necessary. To estimate memory consumption, use the calculator provided in the LLM Inference Sizing and Performance Guidance. Even if the attention heads divide evenly, a single HGX server might not have enough memory capacity to serve the full context length, forcing scale-out across multiple servers.

4.1 Example of model requirements

Table 2 lists how these constraints apply to a range of popular LLMs. With the exception of Llama-3.1-405B, all of the listed models are Mix of Expert (MoE) LLMs. The table includes each model's parameter size, the number of attention heads, context length, and the **minimum number of HGX servers (8x H100 GPUs per server)** required to serve them at full context length. For MoE models, an additional column is provided to show the number of **Active parameters**.

Table 2. Minimum HGX server requirements for LLM serving

Models	Total parameters	Active parameters	num_attention_heads	Full context length	Minimum H100-80G required for full context length	Minimum HGX servers required for full context length
gpt-oss-120b	117B	5.1B	64	128K	8	1
Llama-3.1-405B-Instruct	405B	N/A	128	128K	16	2
DeepSeek-R1	671B	37B	128	128K	16	2
Mixtral-8x22B-Instruct-v0.1	141B	39B	48	64K	8	1
Qwen3-235B-A22B-Thinking-2507	235B	22B	64	256K	16	2
Kimi-K2-Instruct	1T	32B	64	128K	32	4
Llama-4-Maverick-17B-128E-Instruct	400B	17B	40	1M	10	2 (only use 10 out of 16 GPUs)
Llama-4-Scout-17B-16E-Instruct	109B	17B	40	10M	5	1 (only use 5 out of 8 GPUs)

Note that:

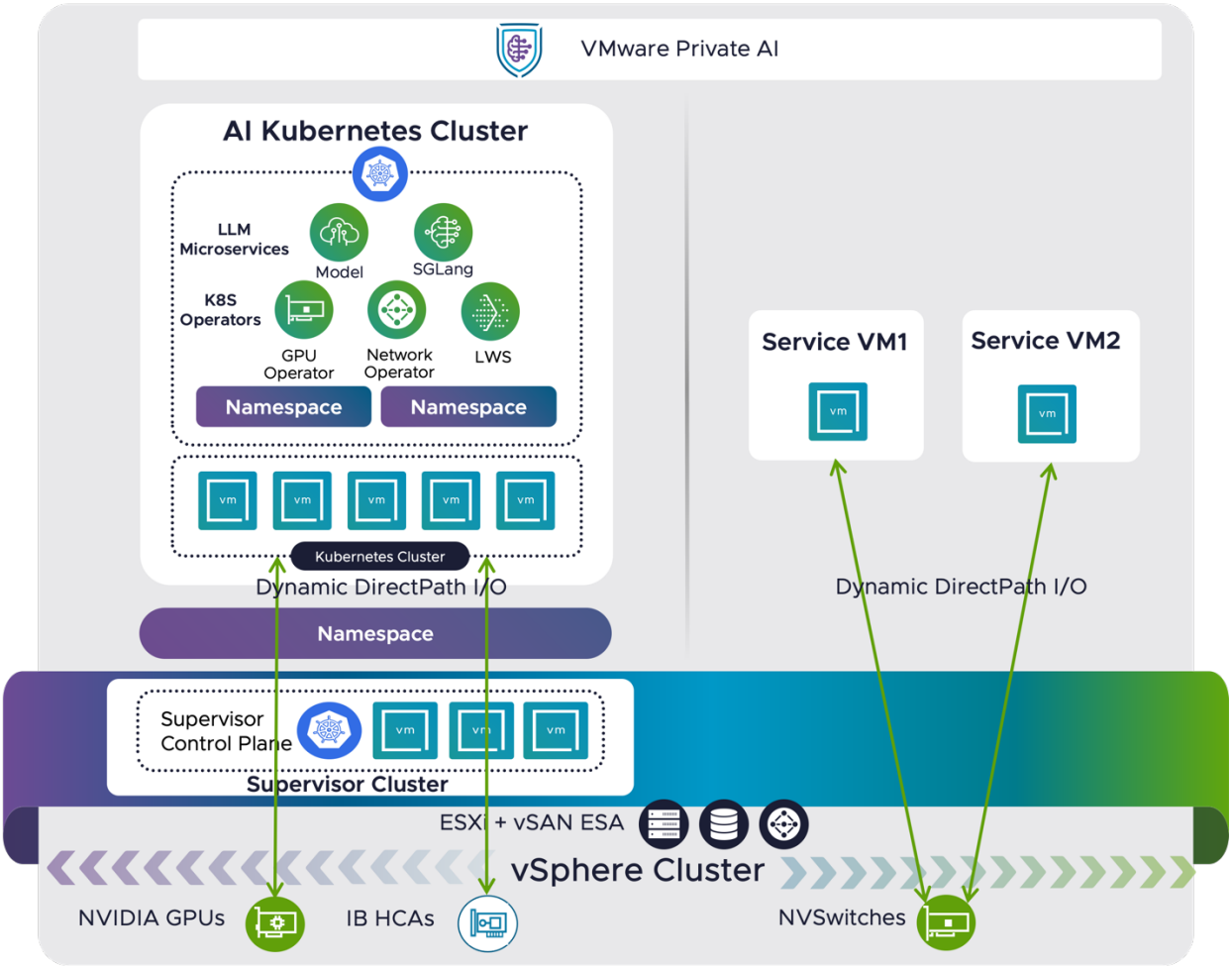
- **Single-node fit:** Some models, like **gpt-oss-120b**, can be fully served on 8x H100 GPUs in a single HGX server, even at maximum context length.
- **Distributed (multi-nodes) requirement:** Larger models such as **Llama-3.1-405B** and **DeepSeek-R1** require at least **2 HGX servers**, primarily due to the model weights demand.
- **Scaling beyond 2 nodes:** Ultra-large or long-context models (e.g., **Kimi-K2**) require 4 HGX servers. The high context length (up to **10M tokens**) and non-divisible attention head counts make them impossible to serve on fewer nodes.

Distributing Llama-4 across 16 H100 GPUs in two HGX servers is not applicable because SGLang's `self.total_num_heads % attn_tp_size == 0` assertion (40 % 16 != 0) is violated. This uneven division of Llama-4's 40 attention heads by the 16-GPU attention tensor parallelism size prevents proper model partitioning and loading, causing inference deployment errors. Solutions involve architectural modification, such as using 10 out of 16 GPUs for Llama-4-maverick and the remaining 6 for other LLMs. As a result, we showcase how to deploy DeepSeek-R1-0528, Llama-3.1-405, and Qwen3-235B-A22B-Thinking-2507 on multi-nodes in VCF. For our initial deployment of distributed LLM inference, we chose SGLang. A similar approach is demonstrated in the NVIDIA doc Example: Helm chart for DeepSeek R1 using an SGLang Backend. In future revisions, we plan to evaluate additional inference engines.

5. Architecture overview

Figure 5 shows an overview of the architecture used to deploy distributed LLM inference with GPUDirect RDMA by IB in VMware Private AI on HGX servers.

Figure 5. Solution architecture of deploying distributed LLM inference by GDR with IB in VCF



The design incorporates several key components:

- **AI Kubernetes Cluster:** Hosts AI microservices like SGLang, Leader-Worker-Set (LWS), along with GPU and network operators.
- **Supervisor Cluster:** Provides namespace management, workload orchestration, security, governance, and policy enforcement.
- **Service VMs:** Each ESX host runs a dedicated lightweight VM (2 vCPUs and 4 GB memory) that always remain powered on for high availability.
  - The service VMs are connected to 4x NVSwitches in shared-passthrough mode.
  - They run the NVIDIA Fabric Manager, ensuring NVSwitch interconnects remain fully operational.
  - They subdivide the HGX system into GPU partitions without relying on vGPU.
- **Dynamic DirectPath I/O:** Ensures the GPUs and NICs are directly accessible to the Service VMs and Workload VKS nodes.

To ensure system stability and preserve fabric connectivity, Service VMs must be powered on before deploying any GPU-attached VKS workload nodes. Workload VKS nodes must also follow an explicit GPU–NIC pairing policy based on PCIe device identifiers (SBDF: Segment–Bus–Device–Function) exposed by ESX. In an 8x H100 HGX system with 8x IB HCAs, this policy can be aligned with a customized NCCL topology file for the VMClass, providing deterministic GPU–NIC assignments and simplifying distributed scaling.

This layered design ensures optimal performance for **distributed inference and training**, while maintaining operational consistency through VMware's enterprise control plane.

5.1 Software used for deployment

Table 3 lists the software components used in this deployment. We have validated GPUDirect RDMA and distributed NCCL performance on both VCF 5.2.1 and VCF 9.

Table 3. VMware software components used in deployment

VCF 9 components

Software	Version	Notes
vSphere	ESX 9.0.0.0.24755229	Hypervisor platform for running VMs and Kubernetes
vCenter	9.0, 24755230	Management for ESX clusters
NSX	9.0.0.0.24733063	Network virtualization and security platform; creates VPC networking
Kubernetes Service	3.3.1-embedded	Kubernetes Service for orchestration
Supervisor	v1.30.5+vmware.4-fips-vsc9.0.0.0-24686447	Built-in Kubernetes control plane (vSphere Supervisor)
VKS nodes	v1.32.0---vmware.6-fips-vkr.2 ubuntu 22.04.5	OS for Kubernetes worker nodes

VCF 5.2.1 components

Software	Version	Notes
vSphere	ESX 8.0.3, 24280767	Hypervisor platform for running VMs and Kubernetes.
vCenter	8.0U3b, 24305161	Management for ESX clusters
Tanzu Kubernetes Grid Service	3.3.0	Tanzu Kubernetes Grid for orchestration
Supervisor	v1.29.7+vmware.1-fips-vsc0.1.10-24224934	Built-in Kubernetes control plane (vSphere Supervisor)
TKG nodes	v1.32.0---vmware.6-fips-vkr.2 ubuntu 22.04.5	OS for Kubernetes worker nodes

Software used within VKS or TKG cluster

Software	Version	Notes
GPU Operator	v25.3.0 or NVIDIA AI Enterprise 6.3	Automatic management of GPU drivers
Network-operator	v25.4.0	Configures InfiniBand and RoCE networking
NVIDIA Driver	570.148.08	GPU Driver (via GPU operator)
NVIDIA-Fabric Manager	570.86.15	Manages GPU–NVSwitch communication (via GPU operator)
CUDA	12.6	GPU compute runtime
OFED Driver	DOCA 25.04-0.6.1.0-2	IB driver (via Network Operator)
NCCL	2.27.3	Multi-GPU and distributed communication library
SGLang	0.5.0rc2	LLM Inference engine
LeaderWorkerSet	v0.6.2	PodGroup deployment API
MLNX_OS	3.12.4002	IB Switch OS
GenAI-Perf	0.0.15.post1	Inference stress test benchmark tool

6. Deployment workflow

The following workflow (steps 1~8) presents the complete process for deploying distributed LLM inference with GPUDirect RDMA over InfiniBand in VMware Private AI. To avoid redundancy, the main sections of the following chapters focus only on advanced configurations and important settings specific to distributed inference. Common configuration steps are not repeated here but are instead consolidated in the appendices for reference.

1. **Hardware and firmware preparation:** *(Ch. 7)*
   - Validate HGX server specifications for consistent performance.
   - Update BIOS and firmware, enabling Access Control Services (ACS) and above 4G decoding. Determine whether to enable or disable Sub-NUMA Cluster (SNC).
   - Set the system power profile to Performance Per Watt (OS).
2. **ESX configuration for GPUDirect RDMA enablement:** *(Ch. 8)*
   - Install NMST and MFT for ConnectX-7 NICs using vSphere Life Cycle Manager (LCM). *(App. B.)*
   - Verify and configure ESX kernel settings for ACS (`disableACSCheck=FALSE`, `atsSupport=TRUE`, `enableACSDTP2P=TRUE` for ESX 9). Reboot ESX if changes are made.
   - Enable ATS on all ConnectX-7 NICs using `mlxconfig`. A host reboot is required.
   - Change GPU's reset type to **flr** in the `/etc/vmware/passthru.map` file in ESX.
   - Configure GPUs, NVSwitches, and CX-7 NICs as passthrough devices with correct hardware labels in vSphere Client.
3. **Service VM deployment:** *(Ch. 9)*
   - Deploy Service VMs using the DLVM template via pre-configured content library for Private AI Service (minimum 2 vCPUs and 4GB RAM).
   - Configure static IP and attach 4x NVSwitch devices via Dynamic DirectPath I/O.
   - Install, enable, and start the NVIDIA Fabric Manager service.
   - Clone the validated Service VM to all other ESX hosts. Service VMs must be powered on before VKS nodes.
4. **VKS setup:** *(Ch. 10)*
   - Review VKS architecture, port groups, and IP planning. *(App. C.1)*
   - Enable Workload Management. *(App. C.2)*
   - Deploy Local Consumption Interface (LCI). *(App. C.3)*
   - Create a vSphere Namespace for LLM deployments. *(App. C.4)*
   - Create custom VMClasses with passthrough devices (8 GPUs, 8 NICs) and configure advanced settings for P2P communication and MMIO size. *(Sec. 10.1)*
   - Manage VM Classes in the Namespace, assigning the custom GPU-enabled VMClass. *(App. C.5)*
   - Deploy a TKG Cluster (via LCI UI or YAML) with GPU-enabled worker nodes and persistent storage. *(Sec. 10.3)*

5. **Operator installation:**
   - Install the NVIDIA Network Operator first, then the NVIDIA GPU Operator (with `driver.rdma.enabled=true`). ***(Sec. 10.2 & App. E.1~5)***
   - Deploy NICClusterPolicy CRD (Custom Resource Definition). ***(App. E.6)***
6. **Storage and model download:** ***(Sec. 10.4)***
   - Create PersistentVolumeClaims (PVCs) for LLM model weights.
   - Deploy temporary pods to mount PVCs and download LLM models (e.g., Llama-3.1-405B) using `huggingface-cli`.
7. **LLM deployment with SGLang:**
   - Deploy Leader-Worker Set (LWS). ***(Sec. 10.5)***
   - Deploy the desired LLM (e.g., Llama-3.1-405B, DeepSeek-R1) using SGLang with VM customized NCCL_TOPO_FILE in Appendix I. ***(Sec. 10.6.1~10.6.2)***
   - Test inference API functionality and review launch parameters. ***(Sec. 10.6.3 ~ 10.6.4)***
8. **Performance verification:** ***(Ch. 11)***
   - Verify RDMA and GPUDirect RDMA performance via InfiniBand across pods on different HGX nodes. ***(App. F & G)***
   - Verify NCCL performance. ***(App. H)***
   - Benchmark LLM performance using tools like GenAI-Perf Stress Test. ***(Sec. 11.1~11.2)***

7. Recommended BIOS and firmware settings

- Keep the **BIOS and other firmware updated** to the latest version provided by your OEM server vendor. Refer to **Appendix A** about how to upgrade the firmware of the Atlas2 PCIe Switch.
- Enable **ACS:** For Dell Servers, enable **Virtualization Technology** in the iDRAC. Find similar settings from your specific server vendor.
- Enable **SRIOV:** For Dell servers, enable **Global SRIOV**. Although SRIOV was not enabled on CX-7 HCAs in this document, enabling it will not negatively impact DirectPath I/O performance and is beneficial for other and future workload testing (e.g., vGPU, other IB or Ethernet NICs that require SRIOV).
- Enable **above 4G decoding** (also known as **memory mapped I/O above 4GB** or **PCI 64-bit resource handling above 4G**). This setting is typically found under the Advanced, Processor Configuration, or Memory Configuration sections in the BIOS setup.
- **Performance Per Watt (OS)** is the recommended system power profile setting. HPE servers have a similar profile setting. Once this is set, choose **High Performance** for ESX power management.
- Sub-NUMA Cluster (SNC): Enable or disable this setting based on whether your workload heavily utilizes the last level cache (LLC). If LLC is intensive, disable; otherwise, enable.
- I/O Snoop HoldOff Response (If exists): Set to **2K Cycles**.

8. ESX settings

8.1 Install NMST & MFT for ConnectX-7

We recommend you follow the “Steps to configure InfiniBand with vSphere 8.x” section in the InfiniBand Configuration on VMware vSphere 8 to use LCM to install MFT and NMST. You can also refer to Appendix B for the command-line instructions.

8.2 Sanity check the ACS-related settings

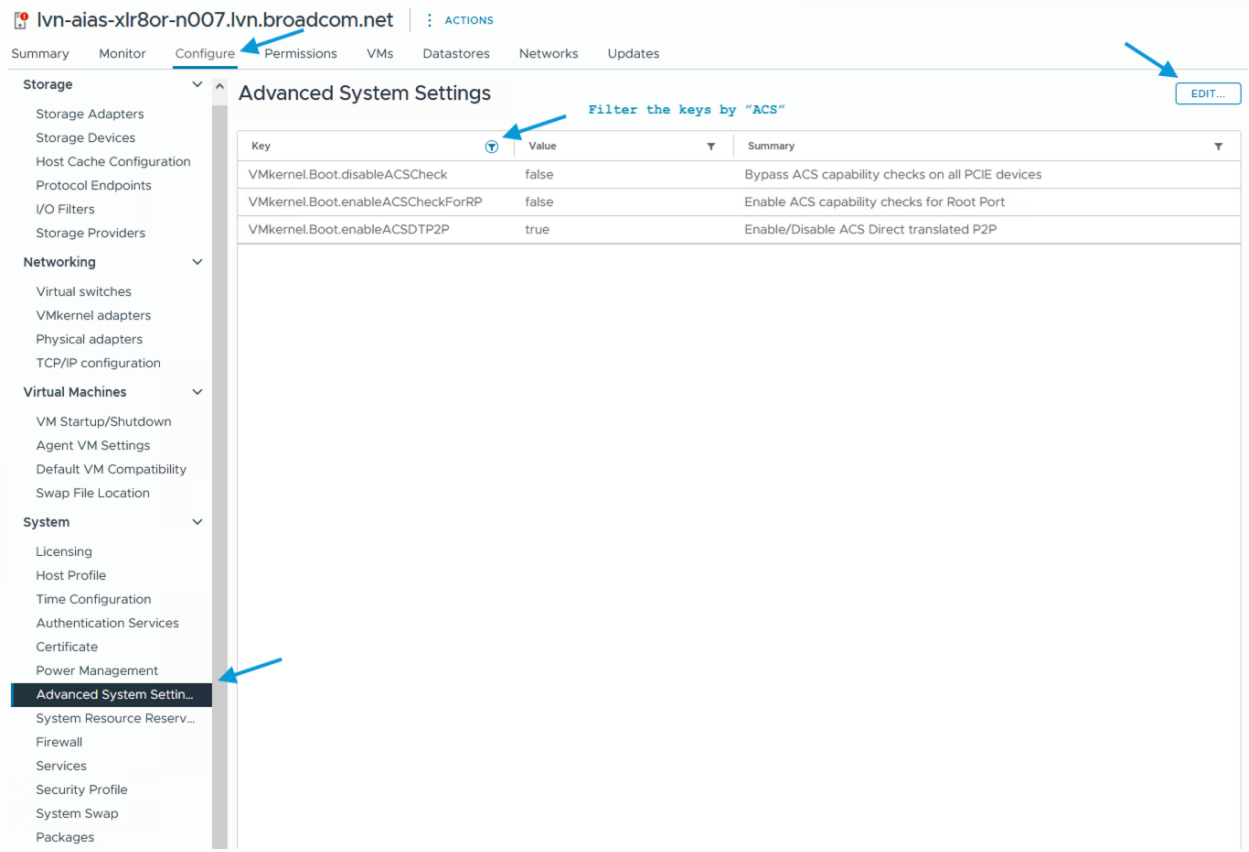
In virtual environments, ESX enables PCIe ACS by default for security. To verify the ACS-related variables on ESX, check the following values.

Example 1. Check ACS-related settings by command lines in ESX

```
# 1. disableACSCheck
[esx] esxcli system settings kernel list -o disableACSCheck
Name             Type  Configured  Runtime  Default  Description
---------------  ----  ----------  -------  -------  -----------
disableACSCheck  Bool  FALSE       FALSE    FALSE    Bypass ACS capability checks on all PCIE devices

# 2. atsSupport
[esx] esxcli system settings kernel list -o atsSupport
Name        Type  Configured  Runtime  Default  Description
----------  ----  ----------  -------  -------  -----------
atsSupport  Bool  TRUE        TRUE     TRUE     Enable Support for PCIe ATS.


# enableACSDTP2P (Available only in ESX 9)
[esx] esxcli system settings kernel list -o enableACSDTP2P
Name            Type  Configured  Runtime  Default  Description
--------------  ----  ----------  -------  -------  -----------
enableACSDTP2P  Bool  TRUE        TRUE     TRUE     Enable ACS Direct Translated P2P.

# If any of these values deviate from their defaults, set them as follows:
[ESX] esxcli system settings kernel set -s enableACSDTP2P -v true
```

Alternatively, you can use vCenter to check and configure these settings in Figure 6:

Figure 6. ACS-related setting in vSphere UI



1. Migrate or power off all running VMs and place the host into maintenance mode.
2. Navigate to the host's **Configure** tab.
3. In the middle pane, go to **System** and click on **Advanced System Settings**.
4. Click **Edit**.
5. Filter the keys by "ACS" and "atsSupport."
6. Review their current values.
7. If changes are required, remember that these VMkernel settings will require a host reboot to take effect.

8.3 Enable ATS on all CX-7

1. Identify the CX-7 devices.

Run the following command to list CX-7 PCI devices. In this example, device `dc:00.0` maps to `mt4129_pciconf7`:

```
# Check the ATS status of CX-7 in ESX
[ESX] /opt/mellanox/bin/mst status -vv
PCI devices:
------------
DEVICE_TYPE             MST                           PCI       RDMA            NET             NUMA
ConnectX7(rev:0)        mt4129_pciconf0               1a:00.0
ConnectX7(rev:0)        mt4129_pciconf1               3c:00.0
ConnectX7(rev:0)        mt4129_pciconf2               4d:00.0
ConnectX7(rev:0)        mt4129_pciconf3               5e:00.0
ConnectX7(rev:0)        mt4129_pciconf4               9c:00.0
ConnectX7(rev:0)        mt4129_pciconf5               bc:00.0
ConnectX7(rev:0)        mt4129_pciconf6               cc:00.0
ConnectX7(rev:0)        mt4129_pciconf7               dc:00.0
```

2. Check the current ATS status.

Use the following command to verify whether ATS is enabled on one of the devices (e.g., mt4129_pciconf7):

```
# Check ATS values
[ESX]/opt/mellanox/bin/mlxconfig -d mt4129_pciconf7 query | grep -i ATS_ENABLED
        ATS_ENABLED                                 True(1)
```

If the output shows `True(1)`, ATS is already enabled on the NIC.

3. Enable ATS if it is disabled.

If ATS is not enabled, run:

```
# Enable ATS
/opt/mellanox/bin/mlxconfig -d mt4129_pciconf7 -y set ATS_ENABLED=true
```

4. Enable ATS on All CX-7 Interfaces

To enable ATS across all 8 ConnectX-7 devices in the host by a single loop:

```
# Loop to enable ATS on all CX-7 IB interfaces
for i in 0 1 2 3 4 5 6 7; do /opt/mellanox/bin/mlxconfig -d mt4129_pciconf$i -y set ATS_ENABLED=true; done
```

5. Reboot the ESX host (required).

A reboot of the ESX host is required for the ATS configuration changes to take effect.

As a final note, enabling GPUDirect RDMA in VCF requires careful attention to system configuration. To ensure stability and avoid potential issues such as host failures (PSOD) or application errors (Segfault), it is recommended to:

- Keep hardware firmware (e.g., Atlas2 PCIe switch) up to date.
- Configure Access Control Services (ACS) properly on ESX.
- Enable Address Translation Services (ATS) on ConnectX-7 NICs.

Following these steps helps create a reliable foundation for GPUDirect RDMA functionality and minimizes the risk of runtime errors in production environments.

8.4 Change the GPU reset type to flr

To change the GPU reset method to **flr**:

1. Edit the `/etc/vmware/passthru.map` file in ESX.
2. Find the entry for NVIDIA and change the `resetMethod` from **bridge** to **flr**.
3. Reboot the ESX host for the change to take effect.

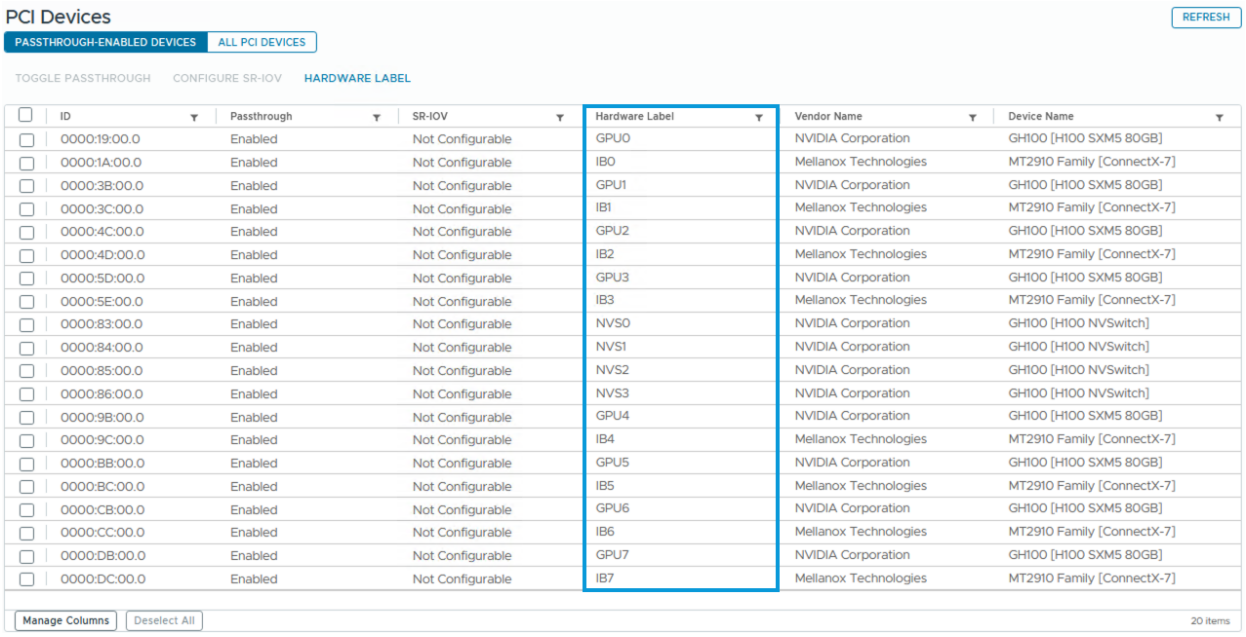
```
# Original content in /etc/vmware/passthru.map in ESX
...
# NVIDIA (FLR issue on Ampere and Hopper GPUs)
10de  ffff  bridge false

# Change to the following
...
# NVIDIA (FLR issue on Ampere and Hopper GPUs)
10de  ffff  flr false
```

8.5 Passthrough GPU, NVSwitch, and CX-7 with hardware labels

Configure all GPUs, CX-7 HCAs, and NVSwitches across the ESX hosts in passthrough mode.

After enabling passthrough, label each device correctly according to its unique SBDF identifier. Figure 7 illustrates an example of HGX servers equipped with 8 GPUs and 8 NICs. For configurations involving HGX servers with 4 GPUs and 4 NICs, please contact Broadcom VCF support or consult an upcoming technical paper for labeling and mapping instructions.

Figure 7. Setting hardware labels for Passthrough GPU, CX-7, and NVSwitch



ID	Passthrough	SR-IOV	Hardware Label	Vendor Name	Device Name
0000:19:00.0	Enabled	Not Configurable	GPU0	NVIDIA Corporation	GH100 [H100 SXM5 80GB]
0000:1A:00.0	Enabled	Not Configurable	IB0	Mellanox Technologies	MT2910 Family [ConnectX-7]
0000:3B:00.0	Enabled	Not Configurable	GPU1	NVIDIA Corporation	GH100 [H100 SXM5 80GB]
0000:3C:00.0	Enabled	Not Configurable	IB1	Mellanox Technologies	MT2910 Family [ConnectX-7]
0000:4C:00.0	Enabled	Not Configurable	GPU2	NVIDIA Corporation	GH100 [H100 SXM5 80GB]
0000:4D:00.0	Enabled	Not Configurable	IB2	Mellanox Technologies	MT2910 Family [ConnectX-7]
0000:5D:00.0	Enabled	Not Configurable	GPU3	NVIDIA Corporation	GH100 [H100 SXM5 80GB]
0000:5E:00.0	Enabled	Not Configurable	IB3	Mellanox Technologies	MT2910 Family [ConnectX-7]
0000:83:00.0	Enabled	Not Configurable	NVS0	NVIDIA Corporation	GH100 [H100 NVSwitch]
0000:84:00.0	Enabled	Not Configurable	NVS1	NVIDIA Corporation	GH100 [H100 NVSwitch]
0000:85:00.0	Enabled	Not Configurable	NVS2	NVIDIA Corporation	GH100 [H100 NVSwitch]
0000:86:00.0	Enabled	Not Configurable	NVS3	NVIDIA Corporation	GH100 [H100 NVSwitch]
0000:9B:00.0	Enabled	Not Configurable	GPU4	NVIDIA Corporation	GH100 [H100 SXM5 80GB]
0000:9C:00.0	Enabled	Not Configurable	IB4	Mellanox Technologies	MT2910 Family [ConnectX-7]
0000:BB:00.0	Enabled	Not Configurable	GPU5	NVIDIA Corporation	GH100 [H100 SXM5 80GB]
0000:BC:00.0	Enabled	Not Configurable	IB5	Mellanox Technologies	MT2910 Family [ConnectX-7]
0000:CB:00.0	Enabled	Not Configurable	GPU6	NVIDIA Corporation	GH100 [H100 SXM5 80GB]
0000:CC:00.0	Enabled	Not Configurable	IB6	Mellanox Technologies	MT2910 Family [ConnectX-7]
0000:DB:00.0	Enabled	Not Configurable	GPU7	NVIDIA Corporation	GH100 [H100 SXM5 80GB]
0000:DC:00.0	Enabled	Not Configurable	IB7	Mellanox Technologies	MT2910 Family [ConnectX-7]

To assign hardware labels in the vSphere Client, follow these steps:

1. In the VMware Host Client inventory, click **Manage**.
2. Open the **Hardware** tab and select **PCI Devices**.
3. Choose the desired device from the list and ensure passthrough is enabled.
4. Click **Hardware Label**.
5. In the **Edit Hardware Label** dialog box, update the label and click **Save** to apply the changes.

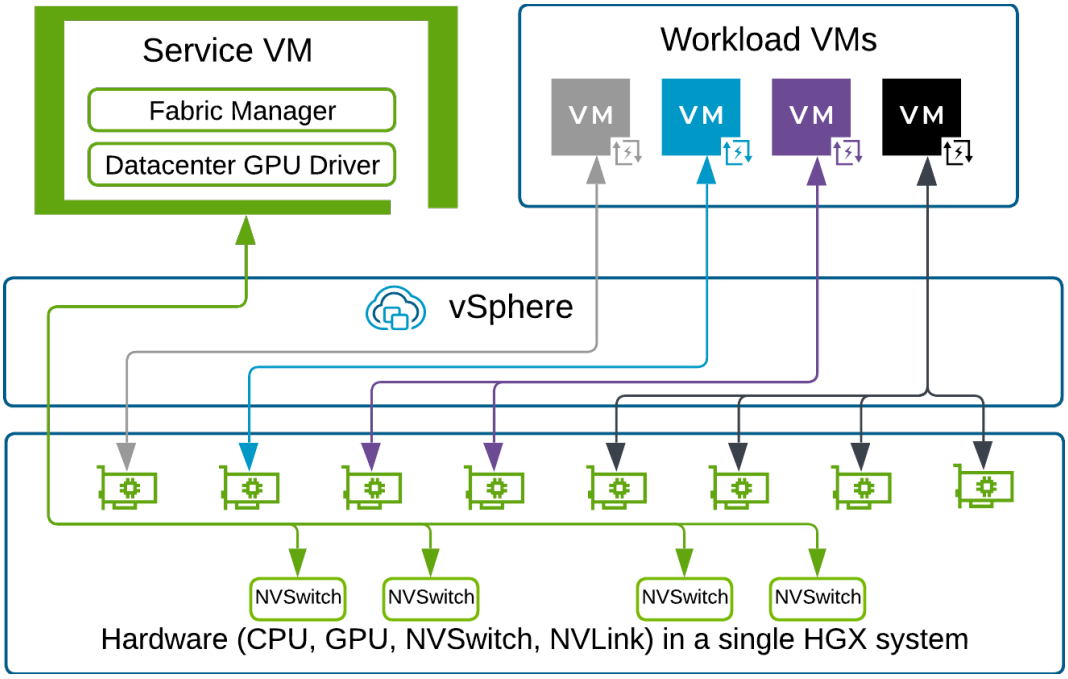
Note: For 8x GPU and 8x NIC HGX servers, the SBDF (each device's PCIe ID) is likely to match the configuration in Figure 8. For other server configurations, such as 4-GPU HGX servers, please refer to an upcoming technical paper on Shared-Passthru or contact VMware Support for guidance on specifying GPU and NIC IDs.

The goal of this deployment was to evaluate the performance of each HGX server when configured with DirectPath I/O (passthrough), assigning all devices (GPUs and InfiniBand HCAs) to a single VKS worker node within an ESX host. Although it is possible to distribute subsets of devices across multiple concurrent VMs in a single HGX server using a shared-nvswitch-passthrough design, our objective here was to passthrough all GPUs and CX-7 HCAs to a single VKS worker node.

9. Deploy Service VMs

This deployment leverages the shared-NVSwitch-passthrough design, which subdivides an HGX system by using Dynamic DirectPath I/O without requiring vGPU. The Service VM must remain powered on at all times to ensure high availability for the workload VMs. Workload VMs use fixed passthrough to bind a specific set of GPUs (2, 4, or 8), ensuring proper utilize the NVSwitch and alignment with the GPU partitions discovered by Fabric Manager. In this deployment guide, we assign all GPU–NIC pairs within a physical server to a single workload VM (VKS node), but you can also define multiple VMClasses to subdivide the system into smaller groups of GPU–NIC pairs.

Figure 8. Service VM in the shared-NVSwitch-passthrough design



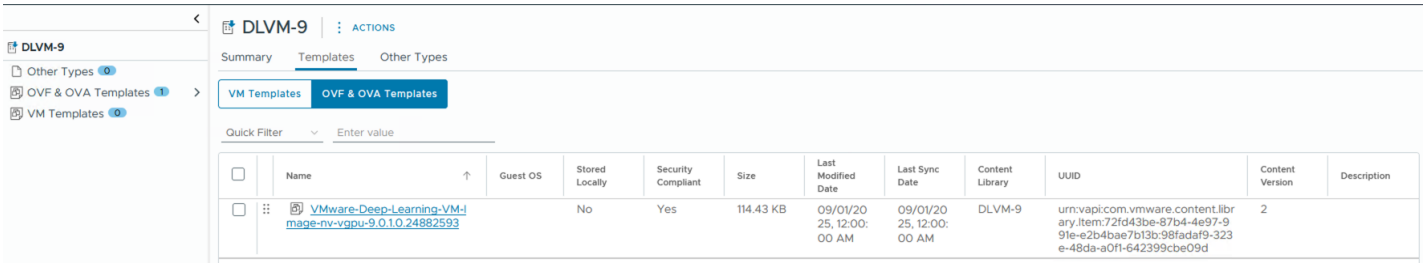
Service VMs can be deployed using the **Deep Learning VM (DLVM) template** available in **Private AI Service**. These VMs act as management nodes to support **NVSwitch fabric services** for multi-GPU systems. If you are not using DLVM, make sure your VM hardware version is above 20.

9.1 Create a DLVM content library

Follow the VMware Deep Learning VM Image Release Notes to download the DLVM template from `https://dl.broadcom.com/<download_token>/PROD/COMP/DLVM/9.0.0.0/lib.json`. The `<download_token>` can be retrieved from the Broadcom Support Portal.

Next, follow this procedure to create a content library.

Figure 9. Import DLVM template to content library

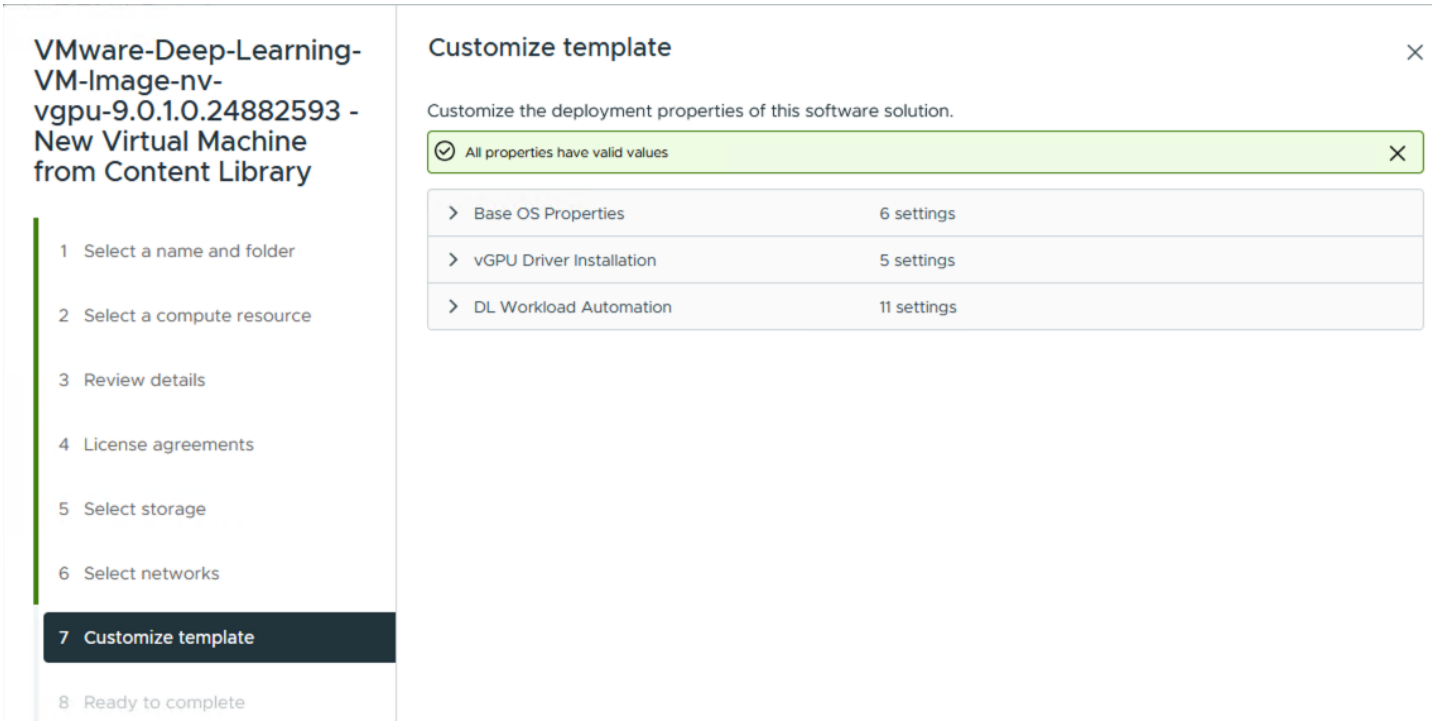


9.2 Customize the DLVM template

When deploying a DLVM via **DirectPath I/O** as shown in Figure 10, the template customization wizard (Step 7) presents three main categories of configuration values:

1. **Base OS Properties** (required)
2. **vGPU Driver Installation** (not used in this guide)
3. **DL Workload Automation** (not used in this guide)

Figure 10. DLVM template deployment UI

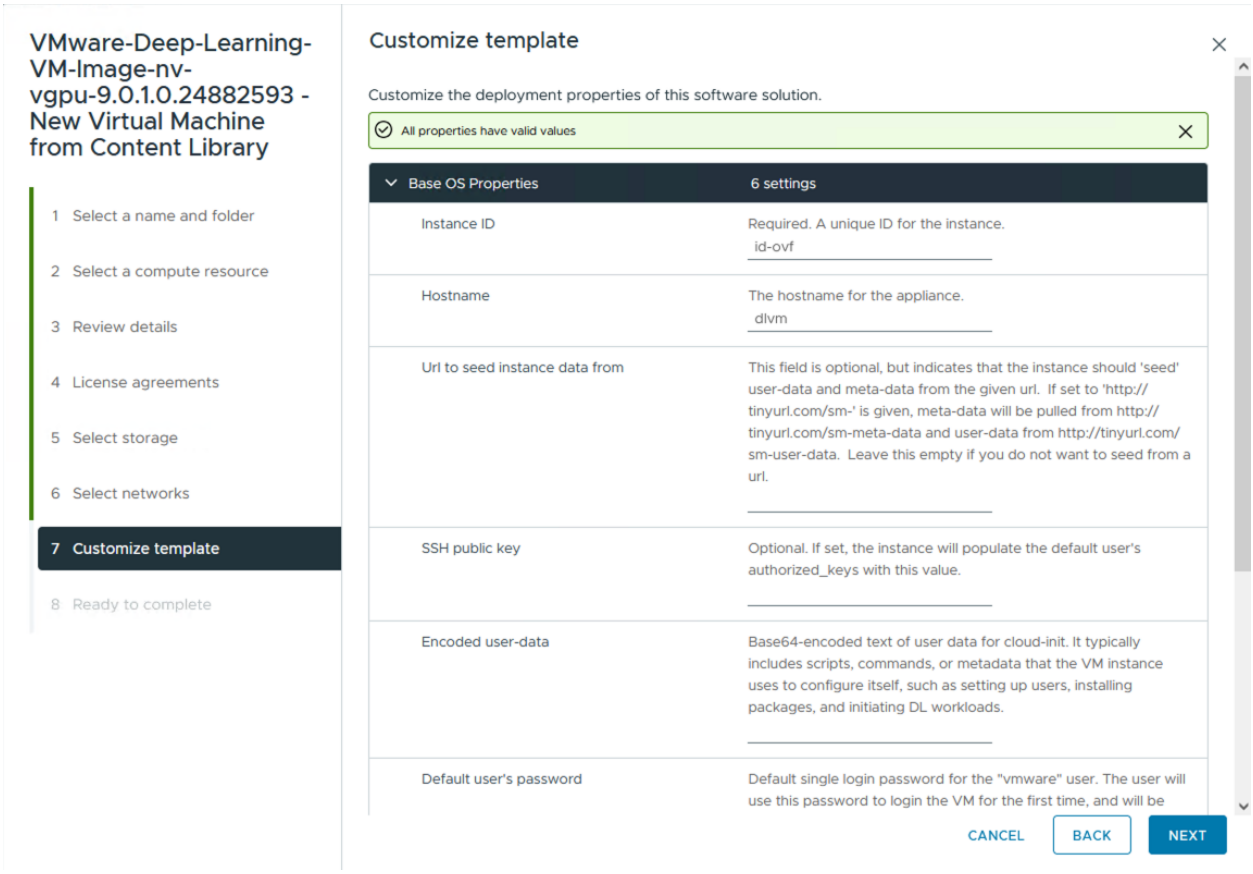


Since this deployment does not use **vGPU stacks** and workloads will be managed separately, you only need to configure values in the **Base OS Properties** section.

Note: Ignore the 5 settings under **vGPU Driver Installation** and the 11 settings under **DL Workload Automation** in Figure 11.

Within **Base OS Properties**, you can set the **hostname**, **user password**, and **Encoded user-data** as shown in Figure 11. The **Encoded user-data** allows you to provide a static IP configuration for the DLVM, which is discussed in the next section.

FIgure 11. Base OS properties



9.2.1 Assign a static IP address

The following `cloud-config.yaml` can be used to configure a static IP for the DLVM (ref: Assign a Static IP Address).

Example 2. cloud-config.yaml for setting static IP

```
#cloud-config.yaml
<instructions_for_your_DL_workload>

manage_etc_hosts: true

write_files:
  - path: /etc/netplan/50-cloud-init.yaml
    permissions: '0600'
    content: |
      network:
        version: 2
        renderer: networkd
        ethernets:
          ens33:
            dhcp4: false # disable DHCP4
            addresses: [x.x.x.x/x]  # Set the static IP address and mask
            routes:
                - to: default
                  via: x.x.x.x # Configure gateway
            nameservers:
              addresses: [x.x.x.x, x.x.x.x] # Provide the DNS server address. Separate multiple DNS server addresses with commas.

runcmd:
  - netplan apply
```

For environments that require a proxy server to gain access to the internet, refer to Configure a Deep Learning VM with a Proxy Server.

After your `cloud-config.yaml` file is prepared, encode it to Base64:

```
base64 -i cloud_init.yaml > encoded_output.txt
```

Finally, copy the Base64-encoded content from `encoded_output.txt` into the **Encoded user-data** field during deployment.

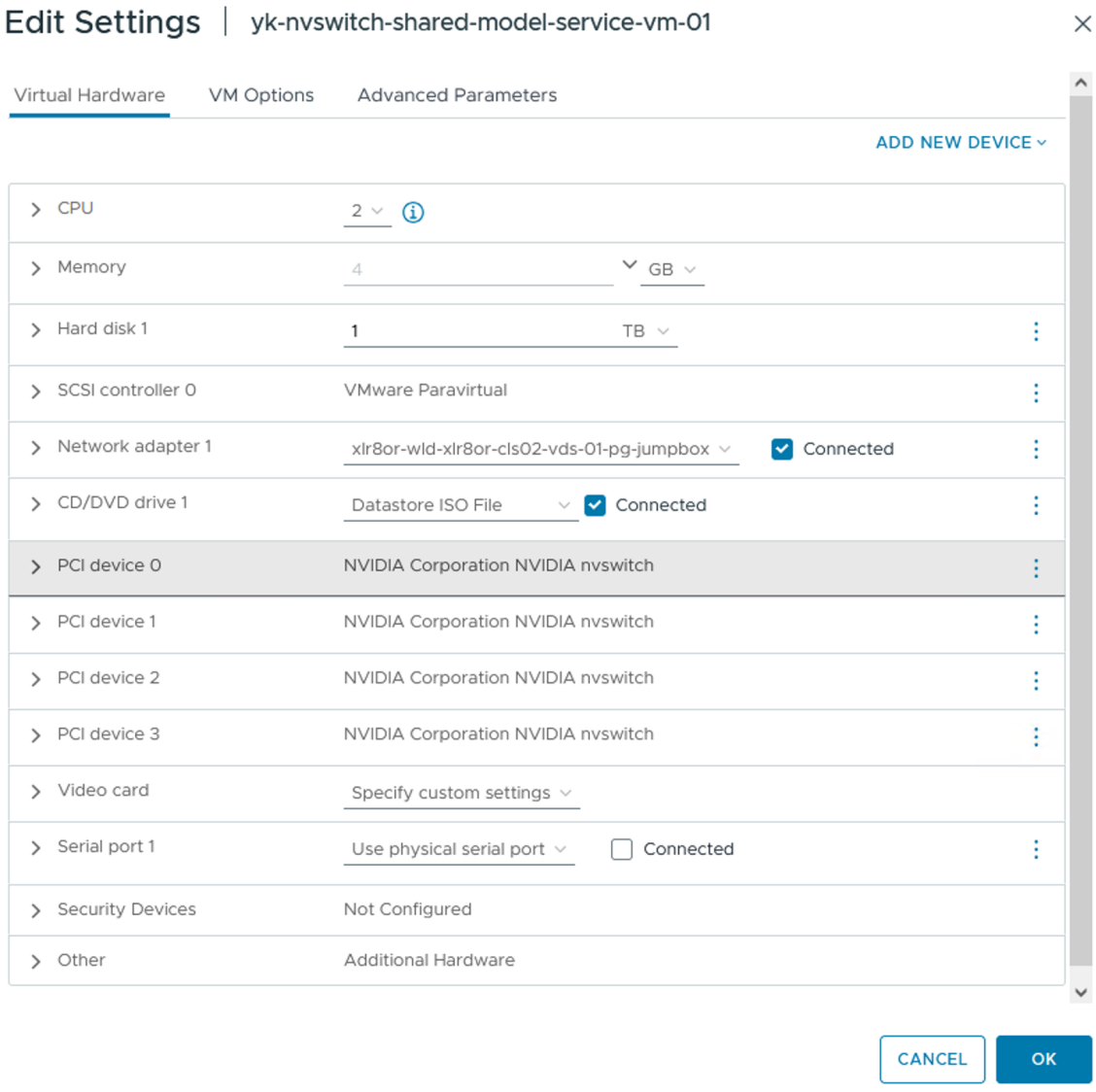
9.3 Configure Service VM

Minimum configuration:

- 2 vCPUs
- 4GB RAM
- 4x NVSwitch devices (attached via Dynamic DirectPath I/O or DirectPath I/O)

Note: With vSphere 8 and above, memory is automatically reserved when a PCI device is added.

Figure 12. Service VM configuration



9.4 Deploy Fabric Manager (FM)

The **NVIDIA Fabric Manager (FM)** package is required for servers with NVSwitch-connected GPUs. It installs the core components and registers the `nvidia-fabricmanager` daemon as a system service. This service manages NVLink peer-to-peer GPU communication.

Example 3. Deploy Fabric Manager

```
# 1. Install Fabric Manager

# Verify Fabric Manager package info
apt info nvidia-fabricmanager-575 -a

# Install matching Fabric Manager version for your GPU driver
sudo apt install nvidia-fabricmanager-575=575.57.08-1
sudo apt install nvidia-fabricmanager-dev-575=575.57.08-1

# 2. Enable and start service
sudo systemctl enable nvidia-fabricmanager
sudo systemctl start nvidia-fabricmanager
sudo systemctl status nvidia-fabricmanager

# 3. Validate service state with "Active" as shown
sudo systemctl status nvidia-fabricmanager
nvidia-fabricmanager.service - NVIDIA fabric manager service
Loaded: loaded (/lib/systemd/system/nvidia-fabricmanager.service; enabled; vendor preset: enabled)
Active: active (running) since Thu 2025-06-26 16:16:48 UTC; 2 months 2 days ago

# 4. Troubleshooting

# 4.1 If status shows as masked, unmask it:
sudo systemctl unmask nvidia-fabricmanager.service

# 4.2 If needed, remove conflicting packages
sudo apt purge nvidia-fabricmanager-570
sudo apt purge nvidia-fabricmanager-dev-570
```

9.5 Clone the Service VM to each ESX host

After the Service VM has been validated, clone it to each additional ESX host in the vSphere cluster. Then change the static IP on each cloned VM. This ensures that every host has a dedicated Service VM running the Fabric Manager for NVSwitch operations.

10. Deploy distributed LLM inference in VKS

Follow Appendix C to complete the prerequisites for deploying VKS.

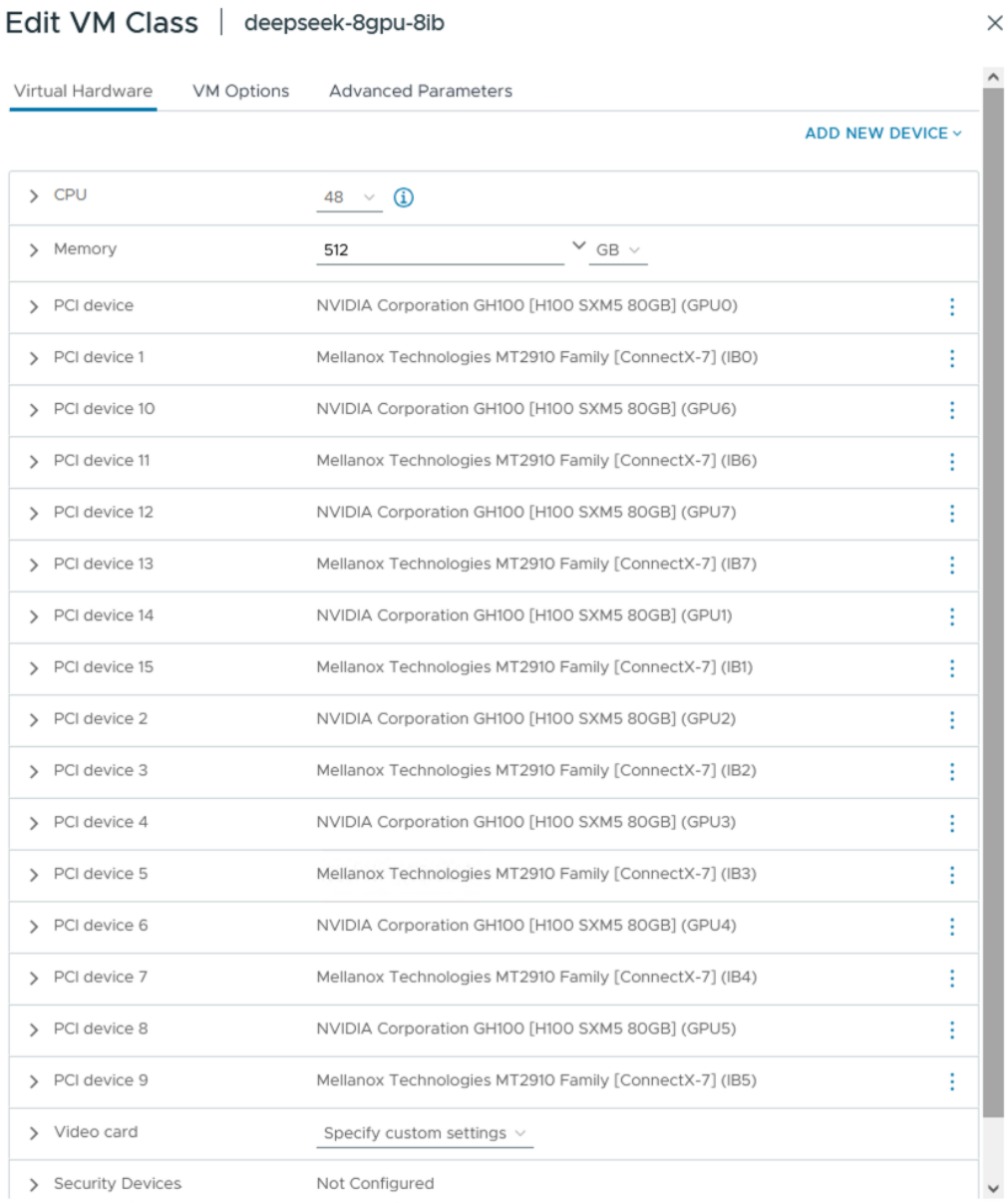
10.1 Create VMClass

VMClasses in VKS define the compute and hardware profile of a VM that can be requested through the Kubernetes service. For GPU-accelerated workloads, you must define a custom VMClass with passthrough devices.

1. **Add the PCI devices:**

   Add the 8 GPUs, 8 NICs, and 4 NVSwitches to the VMClass in the following sequence: GPU0, NIC0, GPU1, NIC1, …, GPU7, NIC7, NVS0, NVS1, NVS2, NVS3.

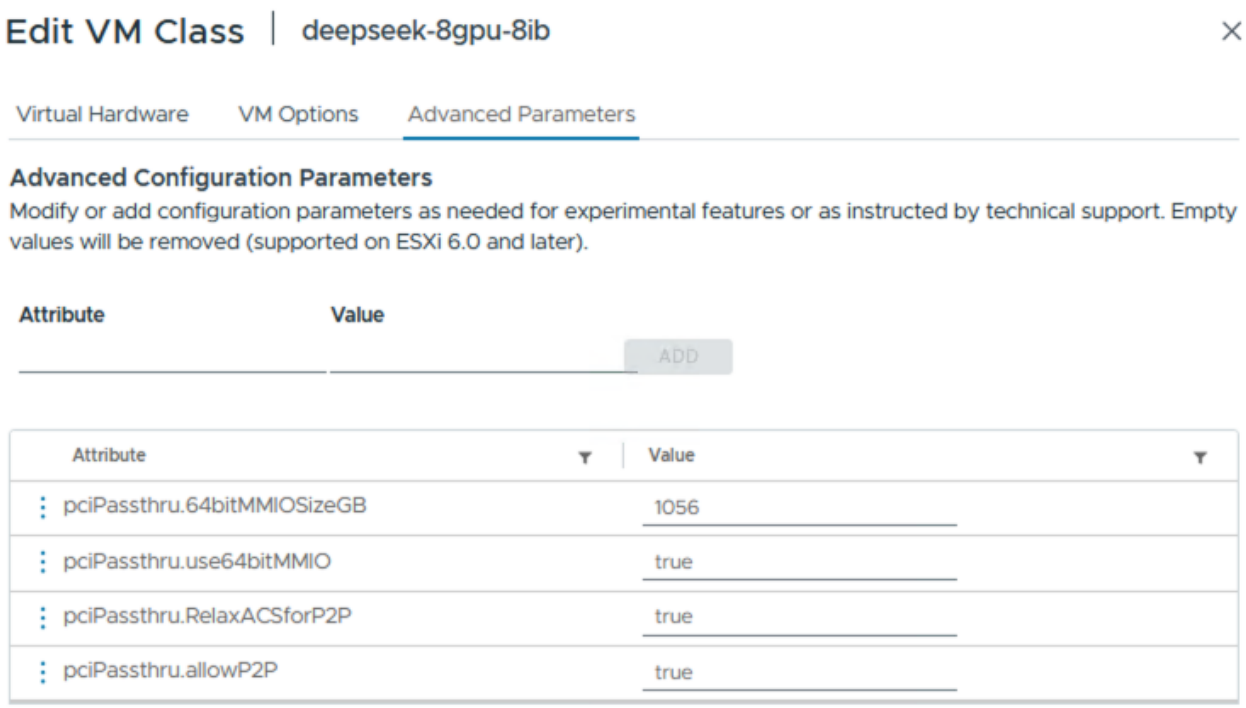
Figure 13. VMClass for an 8-GPU and 8-IB HCA



Configure the VM Class advanced settings:

To ensure the VM can power on and leverage GPU-to-NIC peer-to-peer (P2P) communication, configure the following advanced settings as shown in Figure 14.

Figure 14. Advanced parameters of VMClass for 8-GPU and 8-IB HCA



Relaxing ACS settings for P2P between PCIe devices:

- `pciPassthru.allowP2P = true`
- `pciPassthru.relaxACSforP2P = true`

Required settings for VM power-on:

- `pciPassthru.use64bitMMIO = TRUE`
- `pciPassthru.64bitMMIOSizeGB = 1056`

Without these settings, the VM with passthrough GPUs will not power on.

MMIO size calculation and considerations

Each passthrough NVIDIA H100 (or H200) GPU requires **128 GB of MMIO space**. NICs typically require less, but a minimum of **32 GB** must be reserved.

For a VM with 8 GPUs and 8 NICs, the required MMIO size is:

```
(128 GB * 8 GPUs) + (32 GB for NICs) = 1056 GB
```

Important: Do not set the MMIO size excessively higher than necessary. A very large MMIO allocation can **increase the VM boot time by a few seconds** because the hypervisor needs additional time to allocate and configure the virtualized MMIO address space for all passthrough devices.

10.2 Deploy a VKS or TKG Cluster

A VKS cluster in VCF 9 or a Tanzu Kubernetes Grid (TKG) cluster in VCF 5.2.1 can be deployed in two main ways.

1. **(Recommended) Using the Local Consumption Interface (LCI) UI to deploy the TKG.**

   LCI streamlines the provisioning of Kubernetes clusters directly from the vSphere environment through a simplified, self-service user experience. For detailed steps, refer to Appendix D.

2. **Using a declarative YAML manifest.**

   Alternatively, you can define the cluster specifications in a YAML manifest and apply it directly. This method gives you full control over network CIDRs, storage classes, VM classes, and worker node pool configurations.

   Example 4 shows a YAML file that provisions a cluster named `deepseek-test-cluster` with GPU-enabled worker nodes and custom storage volumes.

Example 4. Declarative YAML to deploy TKG cluster

```
apiVersion: cluster.x-k8s.io/v1beta1
kind: Cluster
metadata:
  name: deepseek-test-cluster
  namespace: deepseek-test
  labels:
    tkg-cluster-selector: deepseek-test-cluster
spec:
  clusterNetwork:
    pods:
      cidrBlocks:
        - 192.168.156.0/20
    services:
      cidrBlocks:
        - 10.96.0.0/12
    serviceDomain: cluster.local
  topology:
    class: builtin-generic-v3.3.0
    version: v1.32.0---vmware.6-fips-vkr.2
    variables:
      - name: vmClass
        value: guaranteed-large
      - name: storageClass
        value: xlr8or-cls02-k8s-vsan
    controlPlane:
      replicas: 1
      metadata:
        annotations:
          run.tanzu.vmware.com/resolve-os-image: os-name=ubuntu
    workers:
      machineDeployments:
        - class: node-pool
```

```
    name: gpu
    replicas: 2
    metadata:
      annotations:
        run.tanzu.vmware.com/resolve-os-image: os-name=ubuntu
    variables:
      overrides:
        - name: vmClass
          value: deepseek-8gpu-8ib
        - name: volumes
          value:
            - name: containerd
              mountPath: /var/lib/containerd
              storageClass: xlr8or-cls02-k8s-vsan
              capacity: 500Gi
```

In this manifest:

- The **control plane** is deployed with 1 replica.
- The **worker node pool** (`gpu`) includes 2 replicas using a custom VM class (`deepseek-8gpu-8ib`).
- A **dedicated containerd volume** (500GiB) is attached to worker nodes for container runtime storage.
- Networking is defined with pod CIDR `192.168.156.0/20` and service CIDR `10.96.0.0/12`.

10.3 Deploy NVIDIA network, GPU operator, and NicClusterPolicy

To run high-performance GPU workloads with RDMA and InfiniBand in Kubernetes, you need to install both the **NVIDIA Network Operator** and the **NVIDIA GPU Operator**. The recommended order is to deploy the Network Operator first, and then the GPU Operator because GPU features such as GPUDirect RDMA depend on the networking stack.

In addition, you must deploy a NicClusterPolicy Custom Resource Definition (CRD) in the VKS cluster (i.e., the `deepseek-test-cluster` created in the previous section). NicClusterPolicy CRD is used by the NVIDIA Network Operator to define and manage the desired cluster-wide configuration for NVIDIA networking components. It provides a declarative API that tells the operator how to set up networking features across nodes with NVIDIA-compatible hardware. This custom resource sets up the Mellanox DOCA or OFED drivers and the RDMA Shared Device Plugin, enabling Kubernetes pods to use InfiniBand devices efficiently.

Refer to Appendix E for the detailed deployment procedure.

10.4 Deploy PVC and download model weights

To prepare the cluster for large AI models, we first need to create **PersistentVolumeClaims (PVCs)**.

Each PVC provides **4 TiB of storage** and uses the **ReadWriteOnce (RWO)** access mode. With RWO, a **block volume is typically attached to a single pod at a time**, which helps ensure reliable, exclusive access for downloading and storing large AI models. This is the default behavior of the **vSAN CSI driver** for volumes formatted with **ext4** or **XFS**.

Note: vSAN "file shares" (essentially NFS) can support **ReadWriteMany (RWM)** or **ReadOnlyMany (ROX)**, allowing multiple pods to access the same volume concurrently. If concurrent access is needed, an NFS-based volume would be an alternative.

Using RWO works well in this deployment to safely ingest LLMs, such as:

- DeepSeek-R1-0528 – 642 GB
- Llama-3.1-405B – 2.3 TB
- Qwen3-235B-A22B-thinking-2507 – 438 GB

Provisioning **4 TiB per PVC** provides ample space for these models, additional files, and future expansion.

Disclaimer: In our experiments, using a 100 GbE NIC to back vSAN ESA and loading DeepSeek-R1 and Qwen3 across two pods took approximately 7 minutes, while Llama-3.1-405B required about 12 minutes. Actual load times may vary based on your specific management and storage network configuration.

10.4.1 Create PVCs

PVC-01 (ib-nas01)

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: ib-nas01
  namespace: deepseek
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 4Ti
  storageClassName: xlr8or-cls02-k8s-vsan
```

Similarly, create `create-pvc02` for `ib-nas02`.

```
# Apply the PVC manifests:
k apply -f create-pvc01.yaml
k apply -f create-pvc02.yaml

# Expected output
k get pvc
```

NAME	STATUS	VOLUME	CAPACITY	ACCESS MODES	STORAGECLASS	VOLUMEATTRIBUTESCLASS	AGE
ib-nas01	Bound	pvc-42bb7ba0-70db-4906-8f54-ddfad1d606b5	4Ti	RWO	xlr8or-cls02-k8s-vsan	<unset>	43d
ib-nas02	Bound	pvc-e89d100b-31fa-46d2-adeb-08fc370d9ddd	4Ti	RWO	xlr8or-cls02-k8s-vsan	<unset>	43d

10.4.2 Create test pods for model download

To download models into the PVCs, we deploy **temporary pods** that mount the volumes. These pods use a lightweight Python image with `huggingface_hub` installed.

Example of a pod manifest for PVC 1:

```
apiVersion: v1
kind: Pod
metadata:
  name: test-pvc-pod-1
  namespace: deepseek
spec:
  containers:
  - name: test-pvc-pod-1
    image: python:3.9.23-slim-bookworm
    imagePullPolicy: IfNotPresent
    command:
    - /bin/sh
    - -c
    - |
      apt-get update && \
      apt-get install -y git && \
      pip install --no-cache-dir huggingface_hub && \
      sleep infinity
    volumeMounts:
    - name: ib-nas-volume
      mountPath: /root/.cache/huggingface
      readOnly: false
  volumes:
  - name: ib-nas-volume
    persistentVolumeClaim:
      claimName: ib-nas01
      readOnly: false
```

Similarly, create `test-pvc-pod-2` for `ib-nas02`.

```
# Apply the pods
k apply -f pvc/test-pvc-pod01.yaml
k apply -f pvc/test-pvc-pod02.yaml

# Interactively login two pods
kubectl exec -it test-pvc-pod-1 -- bash
kubectl exec -it test-pvc-pod-2 -- bash

huggingface-cli login
# Paste your hf_key
```

```
# Download each models
hf download meta-llama/Llama-3.1-405B-Instruct
hf download deepseek-ai/DeepSeek-R1-0528
hf download Qwen/Qwen3-235B-A22B-Instruct-2507

# Check downloaded models' size
~/.cache/huggingface/hub# du -h -d 1 .
438G    ./models--Qwen--Qwen3-235B-A22B-Thinking-2507
2.3T    ./models--meta-llama--Llama-3.1-405B-Instruct
642G    ./models--deepseek-ai--DeepSeek-R1-0528
```

10.4.3 Copy customized NCCL topology file

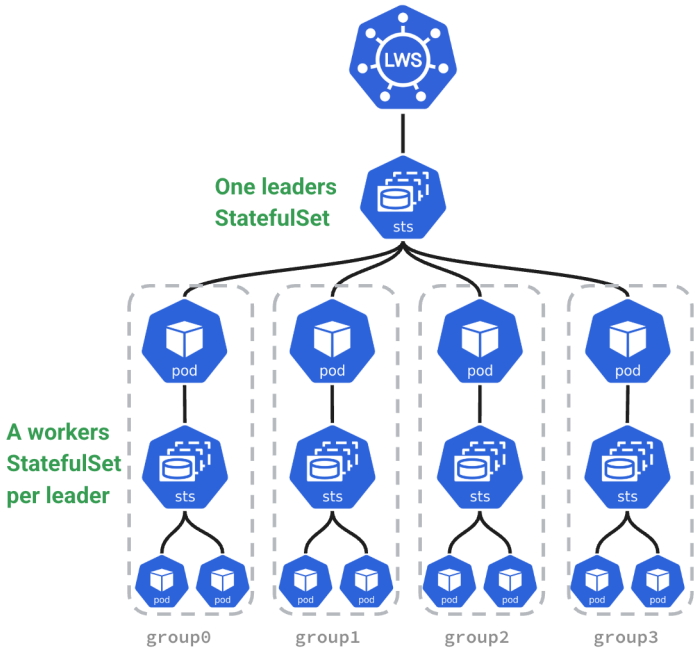
For NCCL functionality and optimal performance within a virtual environment, a customized virtual topology tailored to the previously defined VM Class is essential. Without it, NCCL cannot fully detect the underlying infrastructure, leading to functional issues (such as SGLang failing to launch distributively) or low performance in virtual environments. The virtual topology acts as a map, giving NCCL hints on how to find the fastest and nearest path between the virtualized devices and network pathways, thereby enabling the high-performance, inter-GPU communication essential for distributed deep learning workloads. The customized NCCL topology file `vm_topo_8h100_8ib_mod.xml` (In Appendix G) should be copied into the pods to optimize multi-GPU communication:

```
# Copy vm_topo_8h100_8ib_mod.xml file
kubectl cp vm_topo_8h100_8ib_mod.xml test-pvc-pod-
1:/root/.cache/huggingface/vm_topo_8h100_8ib_mod.xml -n deepseek

kubectl cp vm_topo_8h100_8ib_mod.xml test-pvc-pod-
2:/root/.cache/huggingface/vm_topo_8h100_8ib_mod.xml -n deepseek
```

10.5 Deploy Leader-Worker Set

The Leader-Worker Set (LWS) is a Kubernetes-native API designed to streamline the deployment and management of distributed AI/ML inference workloads, particularly those involving large language models (LLMs) that span multiple nodes and GPUs. For more information, refer to https://lws.sigs.k8s.io/docs/installation/.

Figure 15. LWS architecture




```
# Create namespace
kubectl create ns lws-system
kubectl label --overwrite ns  lws-system pod-security.kubernetes.io/enforce=privileged

# Deploy LWS
VERSION=v0.6.2
kubectl apply --server-side -f https://github.com/kubernetes-
sigs/lws/releases/download/$VERSION/manifests.yaml

# Verify the deployment
k get pod -n lws-system
NAME                                      READY   STATUS    RESTARTS   AGE
lws-controller-manager-77cd846d69-hcbrx   1/1     Running   0          133m
lws-controller-manager-77cd846d69-q6htl   1/1     Running   0          133m
```

10.6 Deploy LLM with SGLang

We recommend referring to the following appendices before proceeding:

- **Appendix F:** Verify RDMA performance via IB on two pods across two HGX nodes.
- **Appendix G:** Verify GDR performance via IB on two pods across two HGX nodes.
- **Appendix H:** Verify NCCL performance on two pods in VKS.

Failure to meet the performance expectations in any of the above areas will impact distributed LLM inference on multi-nodes.

We use **SGLang** to deploy LLMs on multi-GPU HGX nodes within a **VKS** cluster. SGLang leverages **Leader-Worker Sets (LWS)** for distributed inference, enabling efficient model parallelism across GPUs and nodes.

10.6.1 Deploy DeepSeek-R1 on 2 HGX nodes in VKS

The following `LeaderWorkerSet` YAML demonstrates deploying **DeepSeek-R1-0528** on 2 HGX nodes:

- **Leader**: Coordinates the worker nodes and manages distributed computation.
- **Workers**: Execute inference tasks, holding model weights and KV caches.
- **GPU & RDMA resources**: Each container requests 8 GPUs and multiple RDMA shared devices to ensure high-throughput, low-latency communication.
- **Memory configuration**: `--mem-fraction-static=0.85` ensures sufficient memory for model weights, KV cache, and temporary activations.
- The launch uses Bfloat16 (BF16) models for model accuracy performance, not quantized models.
- Use the customized virtual NCCL topology file (`vm_topo_8h100_8ib_mod.xml`) to ensure NCCL correctly detects the GPU-NIC-NVSwitch-PCI_Switch topology in VCF, preventing SGLang segfaults during launch.

Example 5. YAML to deploy DeepSeek-R1-0528

```
apiVersion: leaderworkerset.x-k8s.io/v1
kind: LeaderWorkerSet
metadata:
  name: sglang
spec:
  replicas: 1
  leaderWorkerTemplate:
    size: 2
    restartPolicy: RecreateGroupOnPodRestart
    leaderTemplate:
      metadata:
        labels:
          role: leader
      spec:
        containers:
          - name: sglang-leader
            image: lmsysorg/sglang:v0.5.0rc2-cu126
            securityContext:
              allowPrivilegeEscalation: true
```

```
      capabilities:
        add: ["NET_ADMIN", "IPC_LOCK"]
      privileged: true
    env:
      - name: PYTORCH_CUDA_ALLOC_CONF
        value: expandable_segments:True
      - name: GLOO_SOCKET_IFNAME
        value: eth0
      - name: NCCL_IB_HCA
        value: "mlx5_0,mlx5_1,mlx5_2,mlx5_3,mlx5_4,mlx5_5,mlx5_6,mlx5_7"
      - name: NCCL_P2P_LEVEL
        value: "NVL"
      - name: NCCL_IB_GID_INDEX
        value: "0"
      - name: NCCL_NVLS_ENABLE
        value: "1"
      - name: NCCL_IB_CUDA_SUPPORT
        value: "1"
      - name: NCCL_IB_DISABLE
        value: "0"
      - name: NCCL_SOCKET_IFNAME
        value: "eth0"
      - name: NCCL_DEBUG
        value: "VERSION"
      - name: NCCL_NET_GDR_LEVEL
        value: "1"
      - name: NCCL_TOPO_FILE
        value: "/root/.cache/huggingface/vm_topo_8h100_8ib_mod.xml"
      - name: LWS_WORKER_INDEX
        valueFrom:
          fieldRef:
            fieldPath: metadata.labels['leaderworkerset.sigs.k8s.io/worker-index']
    command:
      - python3
      - -m
      - sglang.launch_server
      - '--model-path'
      - 'deepseek-ai/DeepSeek-R1-0528'
      - --attention-backend
      - fa3
      - --mem-fraction-static
      - "0.85"
      - --tp
      - "16"
      - --dist-init-addr
      - $(LWS_LEADER_ADDRESS):20000
      - --nnodes
      - $(LWS_GROUP_SIZE)
```

```
    - --node-rank
    - $(LWS_WORKER_INDEX)
    - --trust-remote-code
    - --enable-multimodal
    - --host
    - "0.0.0.0"
    - --port
    - "8000"
    - --load-balance-method
    - round_robin
  resources:
    requests:
      rdma/rdma_shared_devices_a: 1
      rdma/rdma_shared_devices_b: 1
      rdma/rdma_shared_devices_c: 1
      rdma/rdma_shared_devices_d: 1
      rdma/rdma_shared_devices_e: 1
      rdma/rdma_shared_devices_f: 1
      rdma/rdma_shared_devices_g: 1
      rdma/rdma_shared_devices_h: 1
      nvidia.com/gpu: "8"
    limits:
      rdma/rdma_shared_devices_a: 1
      rdma/rdma_shared_devices_b: 1
      rdma/rdma_shared_devices_c: 1
      rdma/rdma_shared_devices_d: 1
      rdma/rdma_shared_devices_e: 1
      rdma/rdma_shared_devices_f: 1
      rdma/rdma_shared_devices_g: 1
      rdma/rdma_shared_devices_h: 1
      nvidia.com/gpu: "8"
  ports:
    - containerPort: 8000
  readinessProbe:
    httpGet:
      path: /health
      port: 8000
    initialDelaySeconds: 1500
    periodSeconds: 30
    timeoutSeconds: 10
    failureThreshold: 5
  livenessProbe:
    httpGet:
      path: /health
      port: 8000
    initialDelaySeconds: 1800
    periodSeconds: 300
    timeoutSeconds: 30
```

```
            failureThreshold: 3
          volumeMounts:
            - mountPath: /dev/shm
              name: dshm
            - name: model01
              mountPath: /root/.cache/huggingface
      imagePullSecrets:
      - name: dockerhub-broadcom
      - name: aips-prod-broadcom
      - name: aips-dev-broadcom
      volumes:
        - name: dshm
          emptyDir:
            medium: Memory
            sizeLimit: 16Gi
        - name: model01
          persistentVolumeClaim:
            claimName: ib-nas01
  workerTemplate:
    metadata:
      labels:
        role: worker
    spec:
      containers:
        - name: sglang-worker
          image: dockerhub.artifactory.vcfd.broadcom.net/lmsysorg/sglang:v0.5.0rc2-cu126
          securityContext:
            allowPrivilegeEscalation: true
            capabilities:
              add: ["NET_ADMIN", "IPC_LOCK"]
            privileged: true
          env:
            - name: PYTORCH_CUDA_ALLOC_CONF
              value: expandable_segments:True
            - name: GLOO_SOCKET_IFNAME
              value: eth0
            - name: NCCL_IB_HCA
              value: "mlx5_0,mlx5_1,mlx5_2,mlx5_3,mlx5_4,mlx5_5,mlx5_6,mlx5_7"
            - name: NCCL_P2P_LEVEL
              value: "NVL"
            - name: NCCL_NVLS_ENABLE
              value: "1"
            - name: NCCL_IB_GID_INDEX
              value: "0"
            - name: NCCL_IB_CUDA_SUPPORT
              value: "1"
            - name: NCCL_IB_DISABLE
              value: "0"
```

```
    - name: NCCL_SOCKET_IFNAME
      value: "eth0"
    - name: NCCL_DEBUG
      value: "VERSION"
    - name: NCCL_NET_GDR_LEVEL
      value: "1"
    - name: NCCL_TOPO_FILE
      value: "/root/.cache/huggingface/vm_topo_8h100_8ib_mod.xml"
    - name: LWS_WORKER_INDEX
      valueFrom:
        fieldRef:
          fieldPath: metadata.labels['leaderworkerset.sigs.k8s.io/worker-index']
  command:
    - python3
    - -m
    - sglang.launch_server
    - '--model-path'
    - 'deepseek-ai/DeepSeek-R1-0528'
    - --attention-backend
    - fa3
    - --mem-fraction-static
    - "0.85"
    - --tp
    - "16"
    - --dist-init-addr
    - $(LWS_LEADER_ADDRESS):20000
    - --nnodes
    - $(LWS_GROUP_SIZE)
    - --node-rank
    - $(LWS_WORKER_INDEX)
    - --trust-remote-code
    - --enable-multimodal
  ports:
    - containerPort: 8000
  livenessProbe:
    httpGet:
      path: /health
      port: 8000
    initialDelaySeconds: 1800
    periodSeconds: 300
    timeoutSeconds: 30
    failureThreshold: 3
  resources:
    requests:
      rdma/rdma_shared_devices_a: 1
      rdma/rdma_shared_devices_b: 1
      rdma/rdma_shared_devices_c: 1
      rdma/rdma_shared_devices_d: 1
```

```
                rdma/rdma_shared_devices_e: 1
                rdma/rdma_shared_devices_f: 1
                rdma/rdma_shared_devices_g: 1
                rdma/rdma_shared_devices_h: 1
                nvidia.com/gpu: "8"
              limits:
                rdma/rdma_shared_devices_a: 1
                rdma/rdma_shared_devices_b: 1
                rdma/rdma_shared_devices_c: 1
                rdma/rdma_shared_devices_d: 1
                rdma/rdma_shared_devices_e: 1
                rdma/rdma_shared_devices_f: 1
                rdma/rdma_shared_devices_g: 1
                rdma/rdma_shared_devices_h: 1
                nvidia.com/gpu: "8"
            volumeMounts:
              - mountPath: /dev/shm
                name: dshm
              - name: model02
                mountPath: /root/.cache/huggingface
        imagePullSecrets:
        - name: dockerhub-broadcom
        - name: aips-prod-broadcom
        - name: aips-dev-broadcom
        volumes:
          - name: dshm
            emptyDir:
              medium: Memory
              sizeLimit: 16Gi
          - name: model02
            persistentVolumeClaim:
              claimName: ib-nas02
---
apiVersion: v1
kind: Service
metadata:
  name: sglang-leader
spec:
  type: LoadBalancer
  selector:
    leaderworkerset.sigs.k8s.io/name: sglang
    role: leader
  ports:
    - protocol: TCP
      port: 8000
      targetPort: 8000
```

After deployment, the logs in Example 6 confirm that the DeepSeek-R1 service successfully initializes distributed training with Gloo/NCCL and loads model weights from safetensor checkpoints. This ensures both multi-node coordination and correct weight sharding across GPUs.

Example 6. DeepSeek-R1 launching log

```
[deepseek-test-cluster|deepseek] root@head:/home/ml# kubectl logs sglang-0 --follow
...
[2025-08-13 14:29:45] server_args=ServerArgs(model_path='deepseek-ai/DeepSeek-R1', tokenizer_path='deepseek-
ai/DeepSeek-R1',
...

[2025-08-13 14:29:55 TP0] Init torch distributed begin.
...
[Gloo] Rank 0 is connected to 15 peer ranks. Expected number of connected peer ranks is : 15
[Gloo] Rank 1 is connected to 15 peer ranks. Expected number of connected peer ranks is : 15
[Gloo] Rank 2 is connected to 15 peer ranks. Expected number of connected peer ranks is : 15
[Gloo] Rank 3 is connected to 15 peer ranks. Expected number of connected peer ranks is : 15
[Gloo] Rank 4 is connected to 15 peer ranks. Expected number of connected peer ranks is : 15
[Gloo] Rank 5 is connected to 15 peer ranks. Expected number of connected peer ranks is : 15
[Gloo] Rank 6 is connected to 15 peer ranks. Expected number of connected peer ranks is : 15
[Gloo] Rank 7 is connected to 15 peer ranks. Expected number of connected peer ranks is : 15
[Gloo] Rank 0 is connected to 15 peer ranks. Expected number of connected peer ranks is : 15
[Gloo] Rank 1 is connected to 15 peer ranks. Expected number of connected peer ranks is : 15
[Gloo] Rank 2 is connected to 15 peer ranks. Expected number of connected peer ranks is : 15
[Gloo] Rank 3 is connected to 15 peer ranks. Expected number of connected peer ranks is : 15
[Gloo] Rank 4 is connected to 15 peer ranks. Expected number of connected peer ranks is : 15
[2025-08-13 14:29:58 TP0] sglang is using nccl==2.27.6
[Gloo] Rank 5 is connected to 15 peer ranks. Expected number of connected peer ranks is : 15
[Gloo] Rank 6 is connected to 15 peer ranks. Expected number of connected peer ranks is : 15
[Gloo] Rank 7 is connected to 15 peer ranks. Expected number of connected peer ranks is : 15
<<cut for brevity>>
[2025-08-13 14:30:00 TP0] Init torch distributed ends. mem usage=1.75 GB
[2025-08-13 14:30:01 TP0] Load weight begin. avail mem=76.88 GB
<<cut for brevity>>
[2025-08-13 14:30:03 TP5] Using model weights format ['*.safetensors']
Loading safetensors checkpoint shards:   0% Completed | 0/163 [00:00<?, ?it/s]
Loading safetensors checkpoint shards:   2% Completed | 3/163 [00:00<00:06, 23.62it/s]
<<cut for brevity>>
Loading safetensors checkpoint shards: 100% Completed | 163/163 [02:27<00:00,  1.04s/it]
Loading safetensors checkpoint shards: 100% Completed | 163/163 [02:27<00:00,  1.10it/s]
<<cut for brevity>>
[2025-08-29 16:47:13 TP0] Load weight end. type=DeepseekV3ForCausalLM, dtype=torch.bfloat16, avail mem=36.55 GB, mem
usage=40.33 GB.
[2025-08-29 16:47:14 TP3] KV Cache is allocated. #tokens: 381804, KV size: 24.99 GB
[2025-08-29 16:47:14 TP5] KV Cache is allocated. #tokens: 381804, KV size: 24.99 GB
[2025-08-29 16:47:14 TP0] KV Cache is allocated. #tokens: 381804, KV size: 24.99 GB
[2025-08-29 16:47:14 TP0] Memory pool end. avail mem=10.30 GB
[2025-08-29 16:47:14 TP1] KV Cache is allocated. #tokens: 381804, KV size: 24.99 GB
[2025-08-29 16:47:14 TP2] KV Cache is allocated. #tokens: 381804, KV size: 24.99 GB
[2025-08-29 16:47:14 TP7] KV Cache is allocated. #tokens: 381804, KV size: 24.99 GB
[2025-08-29 16:47:14 TP4] KV Cache is allocated. #tokens: 381804, KV size: 24.99 GB
[2025-08-29 16:47:14 TP6] KV Cache is allocated. #tokens: 381804, KV size: 24.99 GB
[2025-08-29 16:47:14 TP0] Capture cuda graph begin. This can take up to several minutes. avail mem=10.20 GB
```

```
[2025-08-29 16:47:14 TP0] Capture cuda graph bs [1, 2, 4, 8, 16, 24, 32, 40, 48, 56, 64, 72, 80, 88, 96, 104, 112, 120,
128, 136, 144, 152, 160]
...
Capturing batches (bs=1 avail_mem=8.89 GB): 100%|██████████| 23/23 [01:54<00:00,  4.96s/it]
[2025-08-29 16:49:09 TP0] Capture cuda graph end. Time elapsed: 114.91 s. mem usage=1.34 GB. avail mem=8.86 GB.
[2025-08-29 16:49:10 TP0] max_total_num_tokens=381804, chunked_prefill_size=8192, max_prefill_tokens=16384,
max_running_requests=2048, context_len=163840, available_gpu_mem=8.86 GB
[2025-08-29 16:49:10] INFO:     Started server process [1]
[2025-08-29 16:49:10] INFO:     Waiting for application startup.
[2025-08-29 16:49:10] INFO:     Application startup complete.
[2025-08-29 16:49:10] INFO:     Uvicorn running on http://0.0.0.0:8000 (Press CTRL+C to quit)
```

The logs above illustrate the sequential stages of the inference server's startup:

- **Distributed Initialization**

  ```
  [2025-08-13 14:29:55 TP0] Init torch distributed begin.
  [Gloo] Rank 0 is connected to 15 peer ranks. Expected number of connected peer ranks is : 15
  ...
  [2025-08-13 14:30:00 TP0] Init torch distributed ends. mem usage=1.75 GB
  ```

  Each rank confirms connectivity to all 15 peers, validating that the distributed backend (Gloo) is fully synchronized across 16 nodes. Memory usage is logged at this stage, showing minimal overhead before model loading.

- **Model checkpoint loading**

  ```
  [2025-08-13 14:30:03 TP5] Using model weights format ['*.safetensors']
  Loading safetensors checkpoint shards: 100% Completed | 163/163 [02:27<00:00, 1.10it/s]
  [2025-08-29 16:47:13 TP0] Load weight end. type=DeepseekV3ForCausalLM, dtype=torch.bfloat16, avail mem=36.55
  GB, mem usage=40.33 GB.
  ```

  The system detects the `safetensors` format and loads all 163 shards in ~2.5 minutes. Once complete, the model (`DeepseekV3ForCausalLM`) is ready in bfloat16 precision with ~40 GB GPU memory usage.

- **KV cache allocation**

  ```
  [2025-08-29 16:47:14 TP0] KV Cache is allocated. #tokens: 381804, KV size: 24.99 GB
  ```

  Multiple threads allocate **KV cache**, which stores attention keys and values for efficient inference on long sequences. The cache size (~25 GB) matches the configured token capacity (381,804).

- **CUDA graph capture**

  ```
  [2025-08-29 16:47:14 TP0] Capture cuda graph begin. ...
  Capturing batches (bs=1 avail_mem=8.89 GB): 100%|██████████| 23/23 [01:54<00:00, 4.96s/it]
  [2025-08-29 16:49:09 TP0] Capture cuda graph end. Time elapsed: 114.91 s. mem usage=1.34 GB. avail mem=8.86
  GB.
  ```

  CUDA graphs are pre-captured for multiple batch sizes (1–160), which reduces kernel launch overhead and improves inference latency. This step takes ~2 minutes and optimizes the GPU for stable throughput under varying workloads. The `avail_mem` is 8.89 GB in the end, which aligns with SGLang's recommended good setting (discussed in Section 10.6.3).

- **Server startup**

  ```
  [2025-08-29 16:49:10] INFO: Application startup complete.
  [2025-08-29 16:49:10] INFO: Uvicorn running on http://0.0.0.0:8000
  ```

  Finally, the inference server (Uvicorn) starts, confirming readiness to accept API requests.

10.6.2 Deploy Llama-3.1-405B-Instruct or Qwen3-235B-A22B-thinking on 2 HGX nodes in VKS

To deploy additional models using the same YAML structure, update the `--model-path` parameter. Examples include:

- **Llama-3.1-405B-Instruct**: 'meta-llama/Llama-3.1-405B-Instruct'
- **Qwen3-235B-A22B-Thinking-2507**: 'Qwen/Qwen3-235B-A22B-Thinking-2507'

This approach allows you to leverage the existing distributed setup with optimized GPU and RDMA resources. Additionally, the `--mem-fraction-static` parameter also needs to be changed, as discussed below.

10.6.3 Launch parameters and GPU memory discussion

This discussion is based on the SGLang documentation regarding hyperparameter tuning, specifically: https://docs.sglang.ai/advanced_features/hyperparameter_tuning.html

During inference, SGLang manages GPU memory according to the following formula:

Total memory usage = model weights + KV cache pool + CUDA graph buffers + temporary activations

1. **Model weights:** Core parameters of the model.
2. **KV cache:** Key-value cache used during attention for long-sequence inference.
3. **CUDA graph buffers:** Pre-allocated memory for recording GPU operations. These buffers enhance latency and throughput, particularly for small-batch or high-QPS inference, by reducing kernel launch overhead. SGLang suggests checking the `available_gpu_mem` value in the logs; an optimal setting is typically between 5–8 GB.
4. **Temporary activations:** These are the intermediate outputs generated by each layer, held in memory temporarily during the token generation process.

Memory Allocation Control (`--mem-fraction-static`): This parameter balances memory between model weights, KV cache, CUDA graph buffers, and activations:

- **Too high:** Leaves insufficient memory for activations and CUDA graph buffers.
- **Too low:** Underutilizes GPU memory, potentially reducing performance.

Example: On an 80 GB GPU with 60 GB model weights and `--mem-fraction-static=0.9`:

- Memory for KV cache: 80 × 0.9 − 60 = 12 GB
- Memory for intermediate activations: 80 × (1 − 0.9) = 8 GB

For the three models, we set the mem-fraction-static as below

Table 4. Model launch parameters

Model	--mem-fraction-static
DeepSeek-R1-0528	0.85
Llama-3.1-405B	0.92
Qwen3-235B-A22B-Thinking-2507	0.8

For more model weights and KV cache, refer to LLM Inference Sizing and Performance Guidance.

CUDA graph buffers:

- **Purpose**: CUDA graphs are a feature in NVIDIA CUDA that lets you "record" a fixed sequence of GPU operations (kernels, memory copies, etc.) once and then "replay" them many times without the CPU needing to relaunch each kernel.
- **Why**: This cuts down kernel launch overhead and improves latency/throughput. This is especially useful in high QPS inference or small-batch scenarios.
- **Memory use**:
  - To use CUDA graphs, the framework pre-allocates a fixed set of GPU buffers to hold **all intermediate tensors** for that sequence.
  - These buffers must be large enough for the **largest** request the graph will handle, so the memory is held even if smaller requests run.
  - Think of it as **reserved space for the replay template**.

Activations:

- Activations are intermediate outputs of each layer, needed temporarily even during inference.
- Unlike encoded prompts (token IDs or embeddings), activations require significantly more memory.
- Memory usage depends on:
  - Batch size
  - Sequence length (prompt or generation step)
  - Model hidden size (vector width)
  - Number of layers processed before discarding activations
- In prefill (processing the prompt), activations scale with prompt length × batch size × hidden size. In decode (generating one token at a time), activations are smaller because sequence length is 1.

10.6.4 Test inference API functionality

Example 7. Test inference API functionality

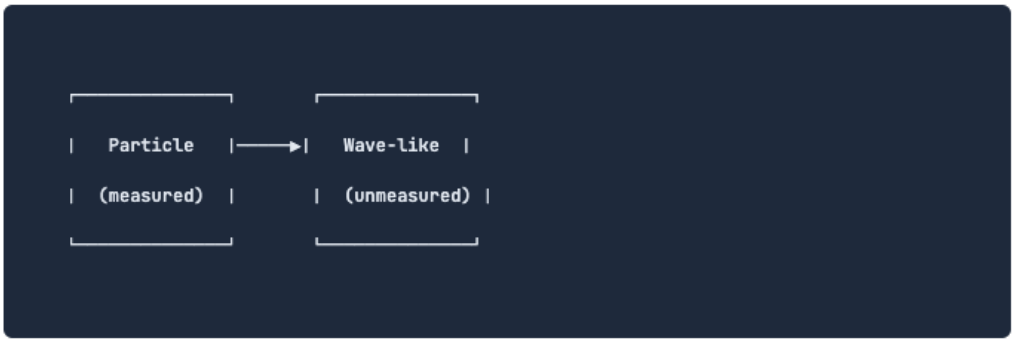
```
# 1. Check the service
kubectl get service sglang-leader -o wide

# Example output:
NAME            TYPE           CLUSTER-IP     EXTERNAL-IP      PORT(S)          AGE   SELECTOR
sglang-leader   LoadBalancer   10.105.76.23   10.191.83.196    8000:32286/TCP   33m
leaderworkerset.sigs.k8s.io/name=sglang,role=leader
# Note the External IP (10.191.83.196) and port (8000) for API queries.

# 2. Query the v1/chat/completions API
curl -s http://10.191.83.196:8000/v1/chat/completions \
  -H "Content-Type: application/json" \
  -d '{
    "model": "deepseek-ai/DeepSeek-R1",
    "messages": [
      {
        "role": "user",
        "content": "Explain quantum mechanics clearly and concisely. Using chain of thought if necessary.
You can use table or graph to help you explain. You can use markdown to format your response."
      }
    ],
    "temperature": 0.7,
    "max_tokens": 8000,
    "stream": false
  }' | jq .
```

Below is a portion of the model response, formatted for readability:

Figure 16. DeepSeek-R1-0528's response

Key Principles (Chain of Thought)

1. **Wave-Particle Duality**
- Particles (e.g., electrons) behave as **waves** (spread-out probabilities) or **particles** (localized) depending on observation.
- Example: *Double-slit experiment*: Electrons create an interference pattern (wave-like) unless measured (particle-like).



2. **Superposition**
- A system exists in **multiple states simultaneously** until measured.
- Example: A quantum bit (qubit) can be in state $|0\rangle$, $|1\rangle$, or both ($\alpha|0\rangle + \beta|1\rangle$).

3. **Uncertainty Principle (Heisenberg)**
- **Position** and **momentum** cannot both be precisely known:
$\Delta x \cdot \Delta p \geq \hbar/2$ ($\hbar$ = reduced Planck's constant).

4. **Entanglement**
- Particles become correlated; measuring one *instantly* affects the other, regardless of distance.
- Basis for quantum teleportation and cryptography.

5. **Measurement Collapse**
- Observing a quantum system forces it into a definite state (collapses the wavefunction).

You can see the model's structured response, which can include markdown, tables, or graphs as requested.

11. Performance

This section details how to use and validate the GenAI-Perf benchmark to assess virtual infrastructure performance requirements or establish a baseline. GenAI-Perf is a convenient command-line tool for measuring the throughput and latency of generative AI models served through an inference server. The instructions provided here are adapted from the step-by-step documentation on Using GenAI-Perf to Benchmark. We use GenAI-Perf to benchmark two LLMs across two pods on 2 ESX hosts. Please note that the examples are for reference only and do not guarantee specific performance outcomes.

11.1 Launch GenAI-Perf stress test

We launch GenAI-Perf in a VM that resides on the same network as the SGLang-leader's external IP.

As shown in Example 8, we first launch a *Triton inference server* container without GPU support. Within this CPU-only container, the GenAI-Perf parameters outlined in Table 5 are used to run a warm-up load test on the SGLang backend. For additional model input parameters, refer to this link. In this setup, the GenAI-Perf tool inside the CPU-only Triton container sends prompt requests to the GPU-enabled SGLang backend, thereby validating its functionality in Figure 24.

Table 5. GenAI-perf launch parameters

Parameters	Value
Input Sequence Length	200
Output Sequence Length	50
Output Sequence Std	10
Concurrency	5

Note: With concurrency=N, GenAI-Perf maintains N active inference requests during profiling. For example, with a concurrency of 4, it sustains 4 simultaneous requests, issuing a new request as each one completes.

For understanding additional metrics and parameters to run GenAI-Perf, consult the Metrics and Parameters and Best Practices page.

Example 8. Setting up GenAI-Perf and warm up load test

```
# Launch a triton inference server container with only CPU
export RELEASE="24.10"
docker run -it --rm --runtime=nvidia \
        --net=host \
        -v $(pwd):/workspace/host \
        nvcr.io/nvidia/tritonserver:${RELEASE}-py3-sdk \
        /bin/bash

# Update genai-perf
pip install -U genai-perf

# Log in with your Huggingface credential for accessing llama-3 tokenizer
pip install huggingface_hub
huggingface-cli login

# Run GenAI-Perf within triton inference server container for a warm up test: ISL=200, OSL=50
export INPUT_SEQUENCE_LENGTH=200
export INPUT_SEQUENCE_STD=10
export OUTPUT_SEQUENCE_LENGTH=50
export CONCURRENCY=5

genai-perf profile \
    -m deepseek-ai/DeepSeek-R1-0528 \
    --endpoint-type chat \
    --streaming \
    -u http://10.191.83.196:8000 \
    --synthetic-input-tokens-mean $INPUT_SEQUENCE_LENGTH \
    --synthetic-input-tokens-stddev $INPUT_SEQUENCE_STD \
    --concurrency $CONCURRENCY \
    --output-tokens-mean $OUTPUT_SEQUENCE_LENGTH \
    --extra-inputs max_tokens:$OUTPUT_SEQUENCE_LENGTH \
    --extra-inputs min_tokens:$OUTPUT_SEQUENCE_LENGTH \
    --extra-inputs ignore_eos:true \
    --measurement-interval 30000
```

Figure 17. Sample output of warm up test

NVIDIA GenAI-Perf | LLM Metrics

Statistic	avg	min	max	p99	p90	p75
Time To First Token (ms)	350.36	167.09	1,265.93	1,202.82	396.57	352.33
Time To Second Token (ms)	39.79	17.04	229.01	198.35	146.18	42.16
Request Latency (ms)	1,282.68	1,218.47	2,172.47	2,126.74	1,377.95	1,267.30
Inter Token Latency (ms)	19.03	18.18	23.02	22.25	21.57	18.95
Output Token Throughput Per User (tokens/sec/user)	52.71	43.45	55.02	54.77	54.38	54.11
Output Sequence Length (tokens)	50.00	50.00	50.00	50.00	50.00	50.00
Input Sequence Length (tokens)	200.39	175.00	227.00	224.00	213.00	207.00
Output Token Throughput (tokens/sec)	194.89	N/A	N/A	N/A	N/A	N/A
Request Throughput (per sec)	3.90	N/A	N/A	N/A	N/A	N/A
Request Count (count)	420.00	N/A	N/A	N/A	N/A	N/A

11.2 Benchmarking

We assessed the performance of DeepSeek-R1-0528 and Llama-3.1-405B in four typical workload scenarios, using a `MEASURE_INTERVAL=100k`. Please note that the performance data presented here may vary based on different conditions. We are continuously working to enhance this performance by tuning different parameters in future releases.

The scenarios tested were:

1. **Short Translation (200 input / 200 output tokens)** – lightweight translation task with short sequences.
2. **Medium Translation (1,000 / 1,000)** – translation with longer input and output sequences, stressing context handling.
3. **Generation (500 / 2,000)** – text generation with relatively small prompts and long outputs.
4. **Summarization (5,000 / 500)** – large input document summarization with modest output length.

Plot description:

- The x-axis represents latency (TTFT: Time to First Token).
- The y-axis represents throughput (tokens per second).
- Points are connected by lines, with the concurrency level indicated (1, 2, ..., 125).

Interpreting the graph:

- For optimal performance, select the latency that best suits each use case. This point will be closest to the top-left corner of your desired TTFT. To accommodate a larger user base, scale horizontally by adding more copies of this LLM.
- Higher points indicate higher throughput.
- Points further to the left indicate lower latency.

11.2.1 DeepSeek-R1-0528 Performance

The enhanced **DeepSeek-R1-0528** model, an improved version of DeepSeek-R1 with superior reasoning capabilities, demonstrated stronger benchmark performance. Under the same test conditions, it achieved a peak throughput of approximately **3,500 tokens/second** at **concurrency=125** for Tasks 1~3, and around 2,200 tokens/second for Task 4. An outlier was observed at concurrency=100, which will be further investigated with the community.

Figure 18. Short translation (200/200) for DeepSeek-R1

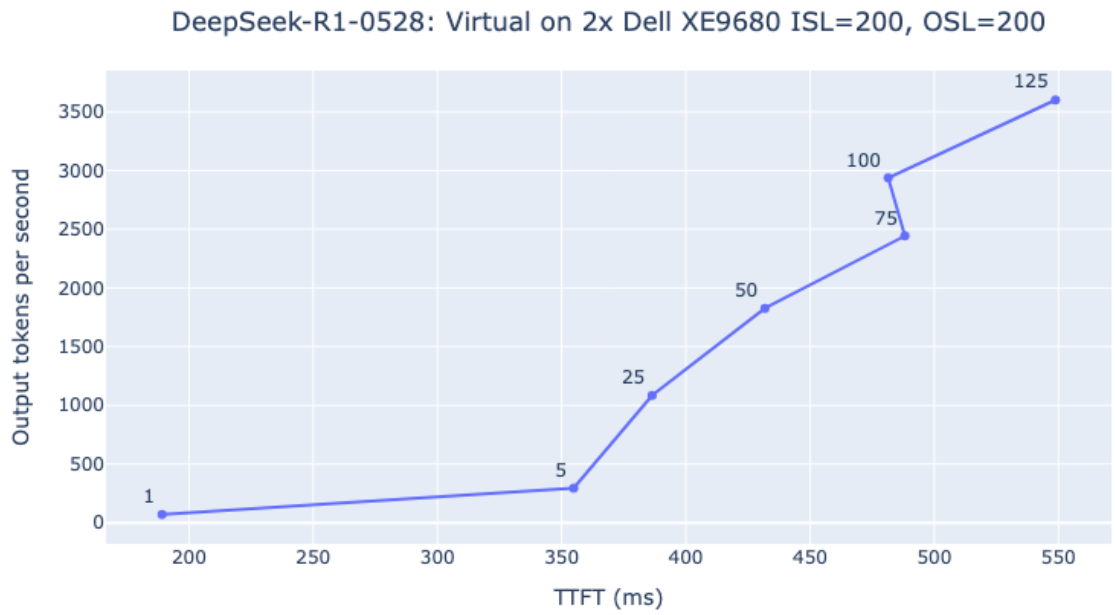


Figure 19. Medium translation (1000/1000) for DeepSeek-R1

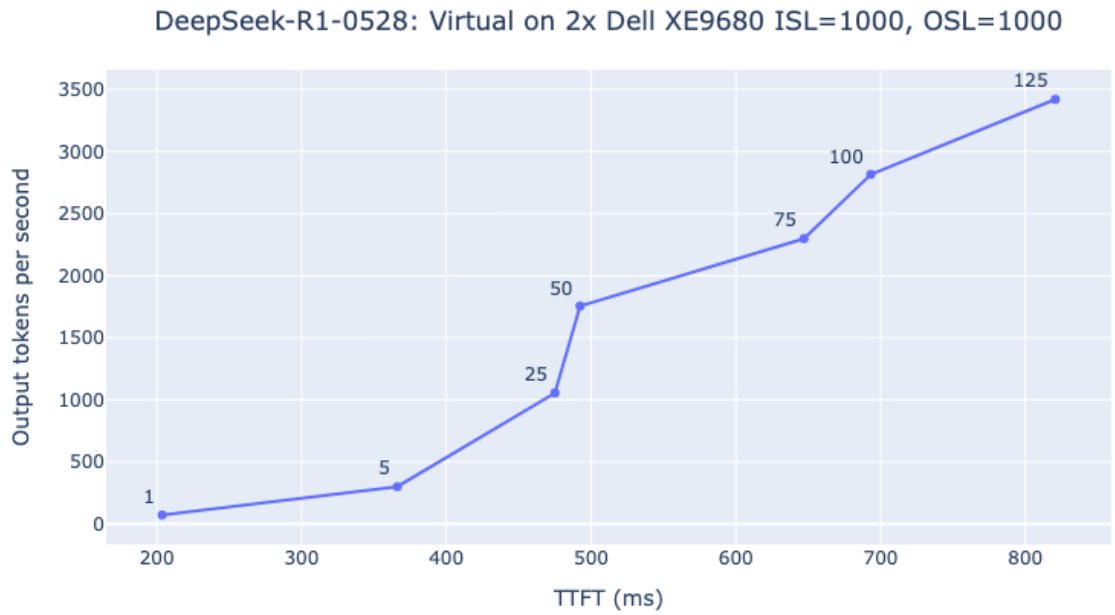


Figure 20. Generation (500/2000) for DeepSeek-R1

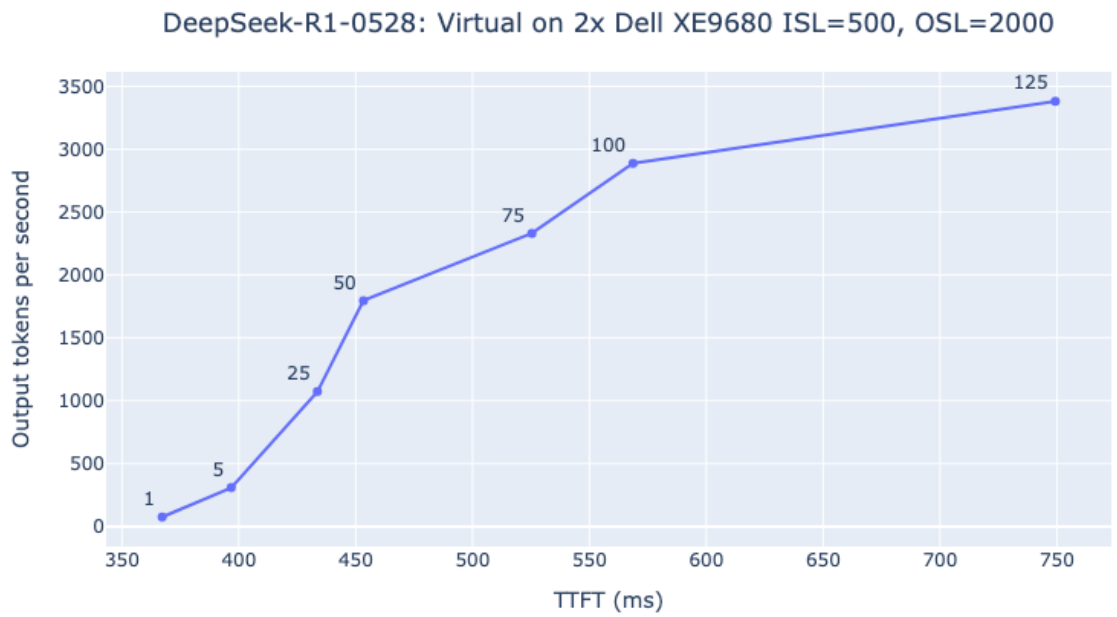
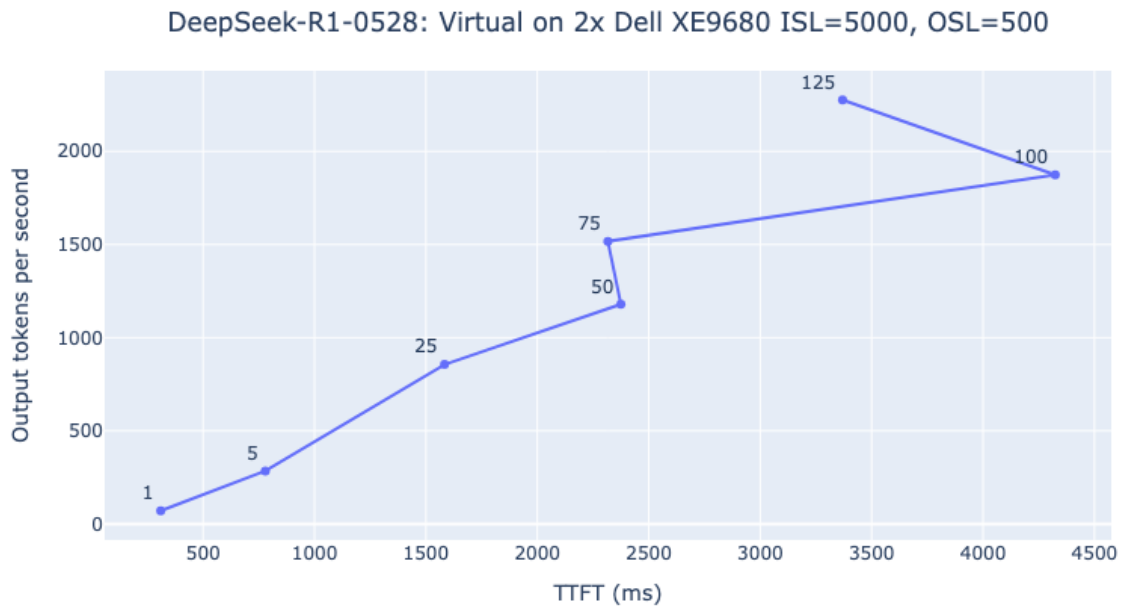


Figure 21. Summarization task (5000/500) for DeepSeek-R1



11.2.2 Llama-3.1-405B performance

This model achieved a peak throughput of ~2,200 tokens/second at concurrency=125 for Task 1~3 and get around 650 tokens/s for Task-4.

Figure 22. Short translation (200/200) for Llama-3.1-405B

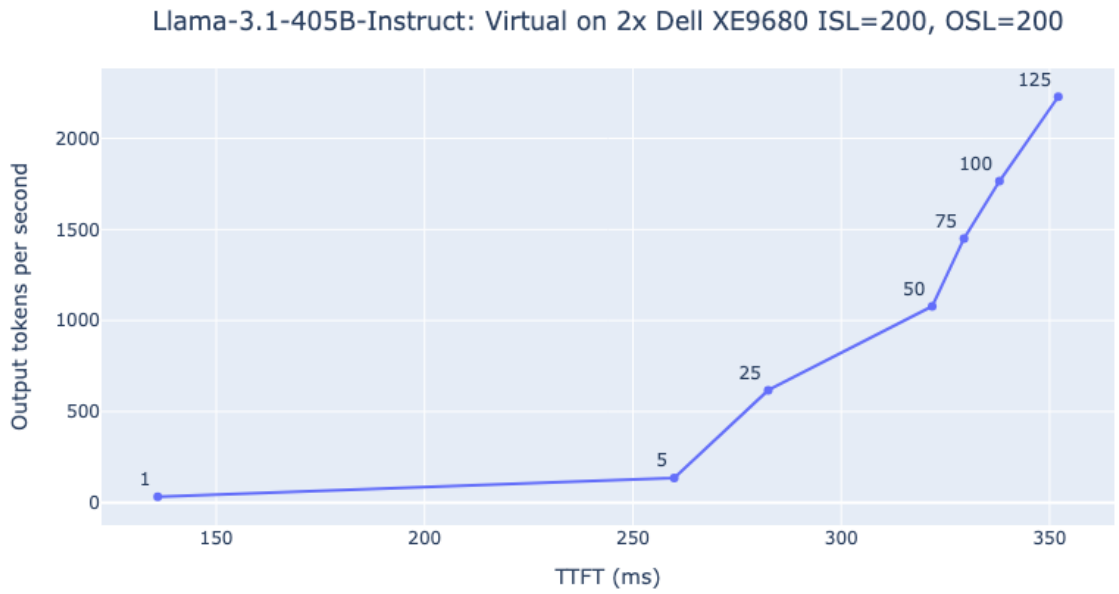


Figure 23. Medium translation (1000/1000) for Llama-3.1-405B

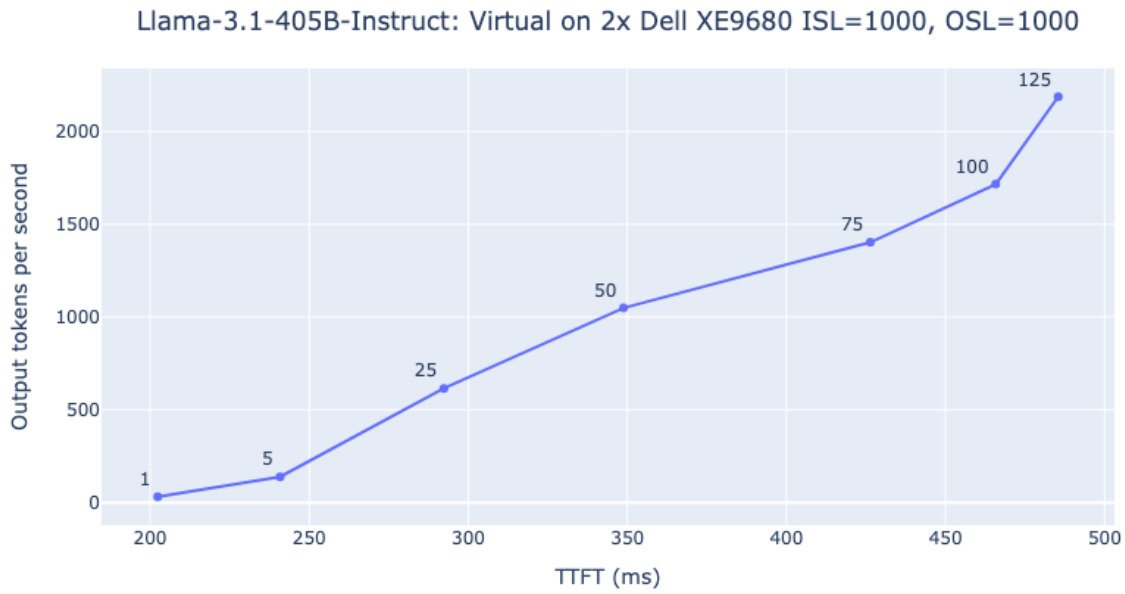


Figure 24. Generation (500/2000) for Llama-3.1-405B

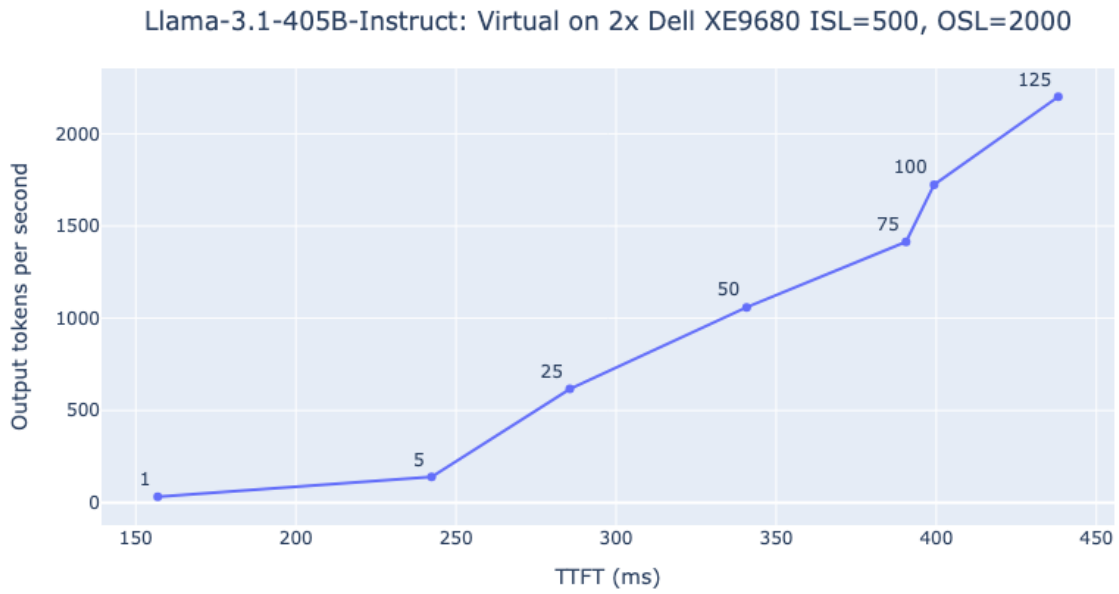
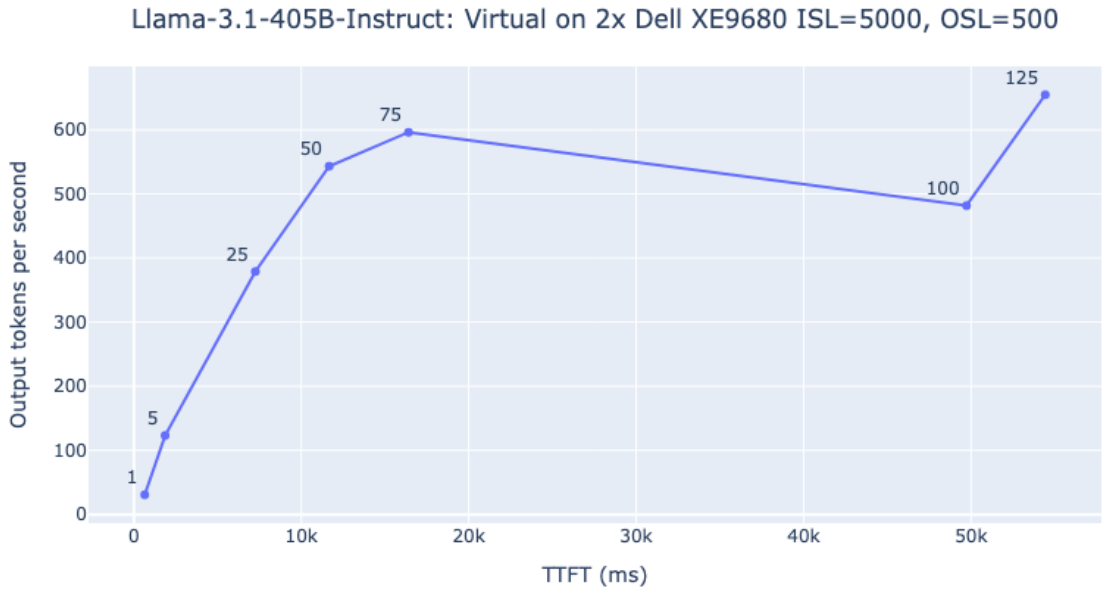


Figure 25. Summarization task (5000/500) for Llama-3.1-405B



12. Conclusion

This document provides a comprehensive guide for deploying distributed LLM inference in VMware Private AI, leveraging GPUDirect RDMA over InfiniBand. It covers critical architectural components, from high-performance NVIDIA HGX servers with NVLink and NVSwitch for intra-node communication to high-bandwidth InfiniBand interconnects for seamless inter-node scaling.

A key focus is on enabling GPUDirect RDMA in VCF through proper configuration of ACS and ATS settings in ESX and on ConnectX-7 NICs, ensuring low-latency, high-throughput data transfer between GPUs and network adapters in a virtualized environment. The guide also provides methods for determining the minimum number of HGX servers needed for different LLMs, accounting for attention heads, context length, and resource bottlenecks.

Practical step-by-step instructions are included for configuring BIOS and ESX settings, deploying Service VMs for Fabric Manager operations, and setting up a Kubernetes-native environment using vSphere Kubernetes Service (VKS). This includes deploying VPC, enabling Workload Management, creating custom VMClasses with passthrough devices, and installing NVIDIA Network and GPU Operators to expose and manage GPU resources effectively.

Finally, the guide addresses best practices for persistent storage with PVCs for model weights and details the deployment of LLMs with SGLang, preparing the reader for robust, scalable AI inference in a VCF private cloud.

13. References

1. VMware Private AI
2. VMware Private AI Foundation with NVIDIA on HGX Servers Reference Design for Inference
3. LLM Inference Sizing and Performance Guidance - VMware Cloud Foundation (VCF) Blog
4. InfiniBand Configuration on VMware vSphere 8
5. InfiniBand and RoCE Setup and Performance Study on vSphere 7.x

---

Ready to get started on your AI and ML journey? Check out these helpful resources:

- Complete this form to contact us!
- Read the VMware Private AI solution brief.
- Learn more about VMware Private AI.

Connect with us on X at @VMwareVCF and on LinkedIn at VMware VCF.

14. Appendix

A. Firmware update

A.1 Atlas2 PCIe Switch Board (PSB) firmware

For the latest PSB firmware updates, please contact your vendor. This document is based on version 4.160.3.0 and above. A new firmware update will be released soon; please install the most recent version when available.

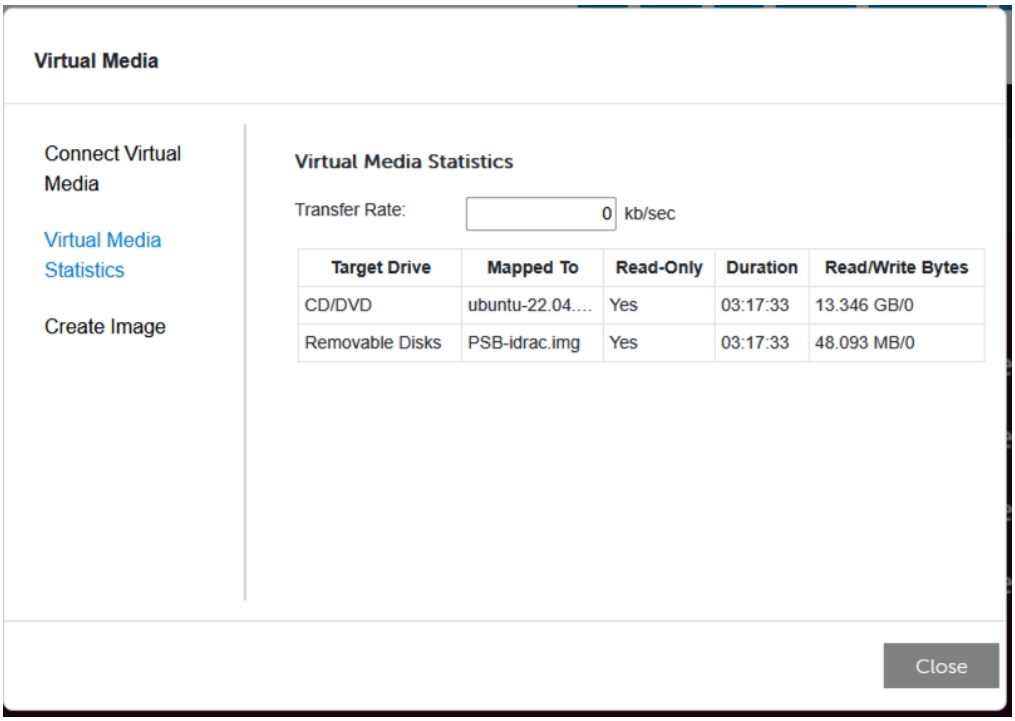
For servers with dual-system installations

Reboot to bare metal and update the PSB using the command:

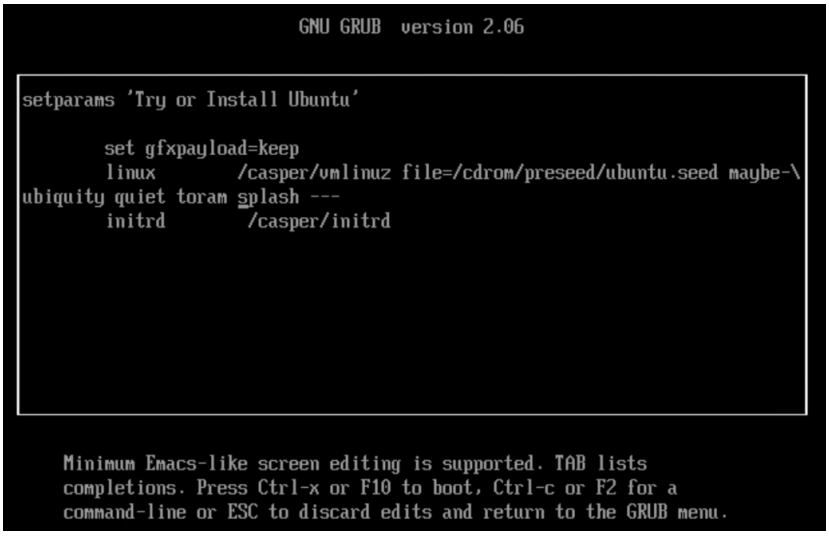
```
chmod +x Firmware_xxx.BIN
sudo ./Firmware_xxx.BIN
```

For servers with only ESX installed:

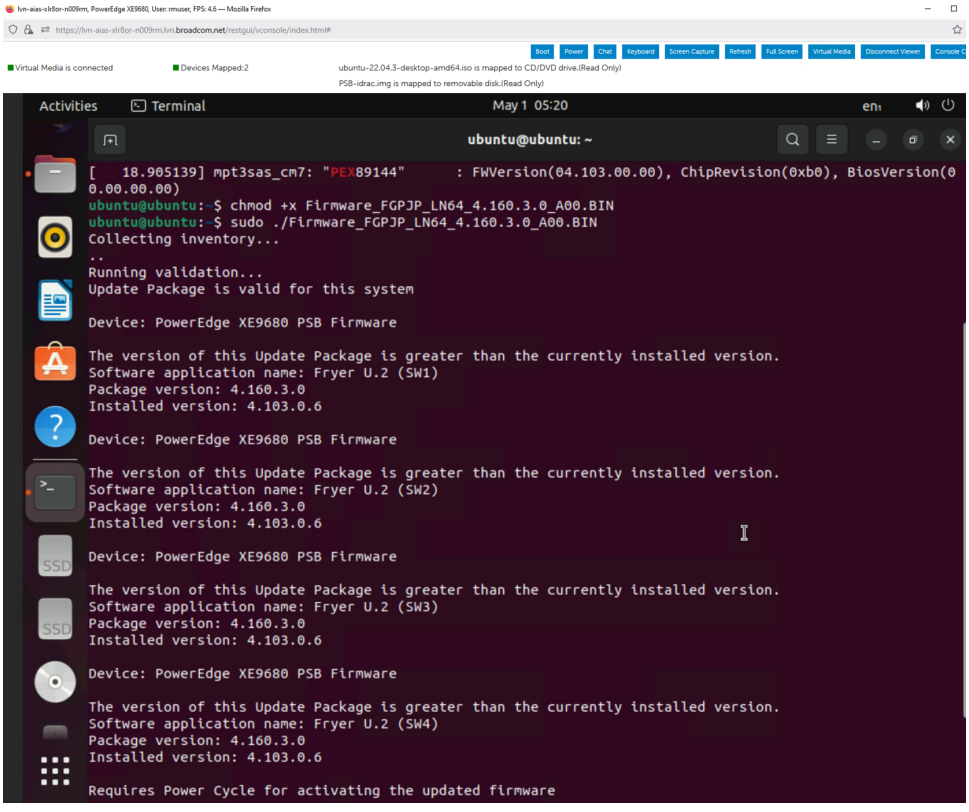
1. To update the PSB's firmware, you will need to boot into an Ubuntu live-ISO. This process may take approximately 30 minutes to load into RAM.



2. Select **Try or install Ubuntu**.



3. Once logged into the live-ISO, run the same bare metal commands to upgrade the PSB's firmware (this will take around 3 minutes). You can check the logs to verify which firmware version has been installed.



4. Reboot the host normally into ESX.

B. Install MFT and NMST on ESX

Download MFT & NMST from https://network.nvidia.com/products/adapter-software/firmware-tools/, then extract them

```
# Check existing CX-7
[ESX] lspci | grep -i mellanox
0000:1a:00.0 Infiniband controller: Mellanox Technologies MT2910 Family [ConnectX-7]
0000:3c:00.0 Infiniband controller: Mellanox Technologies MT2910 Family [ConnectX-7]
0000:4d:00.0 Infiniband controller: Mellanox Technologies MT2910 Family [ConnectX-7]
0000:5e:00.0 Infiniband controller: Mellanox Technologies MT2910 Family [ConnectX-7]
0000:9c:00.0 Infiniband controller: Mellanox Technologies MT2910 Family [ConnectX-7]
0000:bc:00.0 Infiniband controller: Mellanox Technologies MT2910 Family [ConnectX-7]
0000:cc:00.0 Infiniband controller: Mellanox Technologies MT2910 Family [ConnectX-7]
0000:dc:00.0 Infiniband controller: Mellanox Technologies MT2910 Family [ConnectX-7]

# Install NMST
[ESX] esxcli software vib install -v
/vmfs/volumes/datastore_n008_1.5T/VIBs/MEL_bootbank_nmst_4.31.0.149-1OEM.801.0.0.21495797.vib

# Install MFT by VIB
[ESX] esxcli software vib install -v
/vmfs/volumes/datastore_n008_1.5T/VIBs/MEL_bootbank_mft_4.31.0.149-1OEM.801.0.0.21495797.vib

# Or Install MFT or NMST by the zip file by using the -d flag
[ESX] esxcli software vib install -d /vmfs/volumes/vsan\:52dace1d26e912ce-
28312041578d40cf/VIBs/Mellanox-NATIVE-NMST_4.32.0.120-1OEM.801.0.0.21495797_24731977.zip

# Check whether mellanox's VIBs are installed
[ESX] esxcli software vib list | grep -i mel
```

If this is first time install, you may need to enter the recovery mode, please check the section 2.4 in https://www.vmware.com/docs/infiniband-config-vsphere8-perf by running the following example.

```
# Enabling Recovery Mode
esxcli system module parameters set -m nmlx5_core -p "mst_recovery=1"

# After reboot, check CX-7 status
/opt/mellanox/bin/mst status -vvv

# Reverting to Normal Mode
esxcli system module parameters set -m nmlx5_core -p "mst_recovery=0"
```

C. VKS deployment prerequisites

The following appendix applies to VCF 9, if for VCF 5.2.1, refer to page 21 in https://www.vmware.com/docs/ref-design-private-ai-nvidia-hgx-inference.

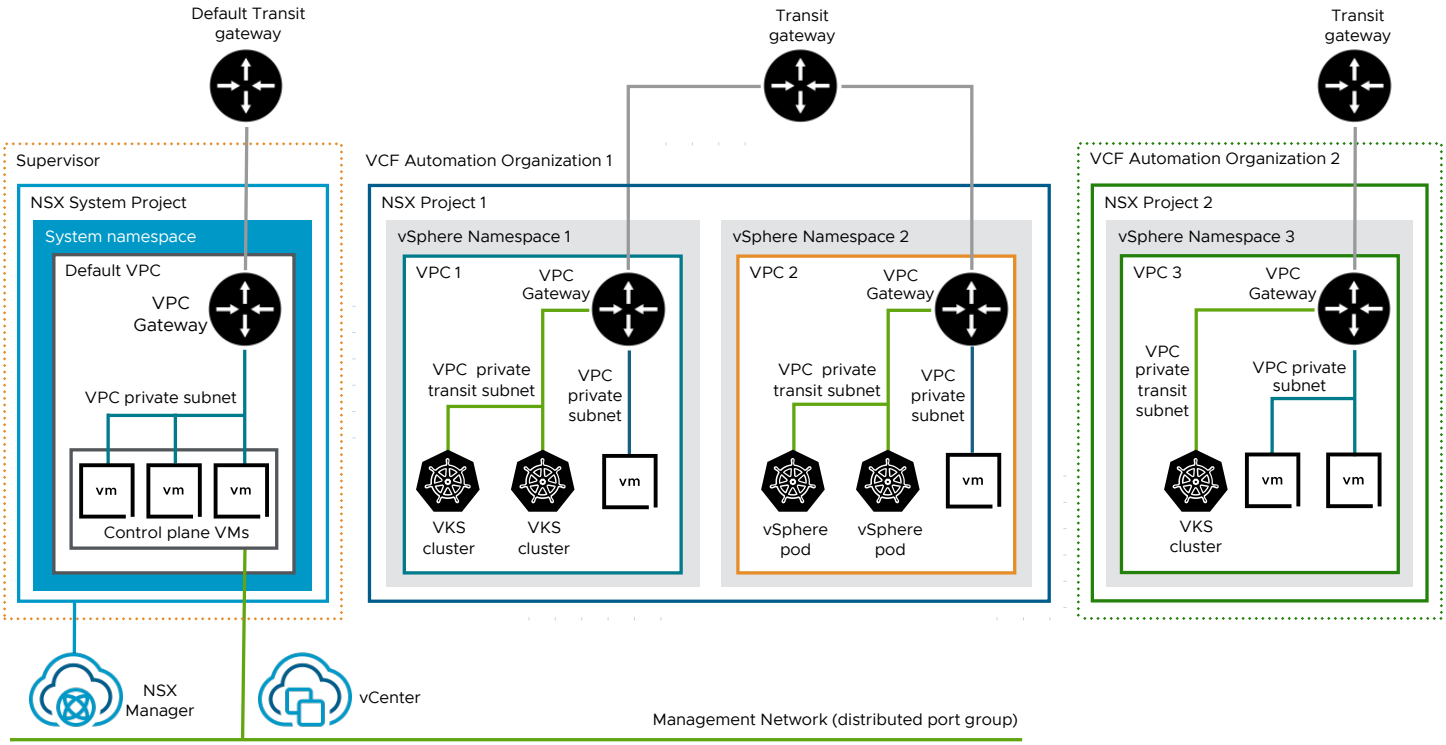
C.1 VKS with VPC-NSX architecture

NSX integrates with the VKS cluster to manage networking for vSphere namespaces and workloads, such as Kubernetes clusters, pods and supervisor services. NSX Virtual Private Clouds enhance this by creating namespace-specific, isolated networking domains, each with dedicated subnets, NAT, DHCP, and firewall policies, mimicking public cloud networking models. This allows for fine-grained control, enabling secure multi-tenancy and simplified traffic management.

All the ingress and egress of the workloads running natively on VKS Kubernetes Clusters will be exposed through an external network, simplifying the network setup, internal communication of vSphere Pods, containers and VKS Kubernetes clusters happens on internal VPC subnets.

VCF Supports multiple network topologies for VKS networking, ranging from Foundation Load Balance (FLB), NSX with AVI and NSX Native Load Balancing. Refer to https://techdocs.broadcom.com/us/en/vmware-cis/vcf/vcf-9-0-and-later/9-0/vsphere-supervisor-installation-and-configuration.html for more details on the different architectural approaches.

Figure 26. Supervisor networking with NSX VPC (image source)



Supervisor Cluster

- The core Kubernetes control plane of the entire VKS environment is composed of a cluster or control plane VMs and the ESX hosts, and it exposes the Kubernetes API.
- Uses the **Management Network** for vCenter management and control plane communication.

Management Network

- Provides the management interface for Supervisor Control Plane nodes.
- Integrated with the VDS network for vCenter control communication.
- Does **not** carry user workloads.

Workload Networks

- **VPC Private subnets**: These subnets are the default access mode for VPC workloads and are only accessible only within the NSX VPC.
- **External subnets**: These are public or external IPs that are accessible both from the project and outside the project. These are primarily used for LoadBalancer-type services and ingresses and egress of the NSX project.
- **Private Transit Gateway subnets**: These subnets are accessible by other workloads in different VPCs within the same NSX project. These subnets are primarily used by SNAT on egress of the VPC and vSphere Pods.

VKS Clusters

- Deployed on a Private VPC subnets, ingress and egress traffic is provided by the External network.
- Each Kubernetes cluster node communicates through the same NSX Logical segment.

DevOps Users / External Services

- Kubernetes API access for the Supervisor and Kubernetes clusters is provided via a VIP managed by the NSX load balancer. This VIP, on TCP port 6443, distributes traffic across control plane nodes for high availability.
- Kubernetes Services of type LoadBalancer in guest Kubernetes Clusters on the Supervisor are exposed via a Virtual IP (VIP) provisioned by the NSX load balancer. This VIP is assigned from the external network, routes external traffic to the Service's endpoints, ensuring load balancing across pods.

Table 6. Network requirements for the VKS setup.

Network	Function	VLAN or Overlay	Routed or Internal
Management Network for Supervisor Control Plane	Routable network with reachability to ESX hosts, vCenter, Supervisor and other components. Control Plane VMs of the supervisor will be connected to this network.	VLAN	Routed
External IP Block	Required for VPC Connectivity Profile with Centralized Gateway. This non-overlapping subnet provided will be configured as an NSX overlay and should be routable via the Edge Cluster. Provides external connectivity for services.	Overlay	Routed
Private Transit Gateway IP Blocks	Configured in VPC Connectivity Profile for transit connectivity.	Overlay	Internal
Private (VPC) CIDRs	Private non-overlapping IP range for VPC segments and IP allocation. Ideally a /16 network for scalability	Overlay	Internal
Service CIDR	IP range for Kubernetes services in the Supervisor.	Overlay	Internal

C.2 Enable Workload Management

Once the network pre-requisites have been documented the next step is to enable the Workload Management with VPC networking in vSphere. This transforms your vSphere cluster into a Supervisor Cluster capable of running Kubernetes workloads directly on ESX hosts within a Virtual Private Cloud (VPC) networking model using NSX native load balancing.

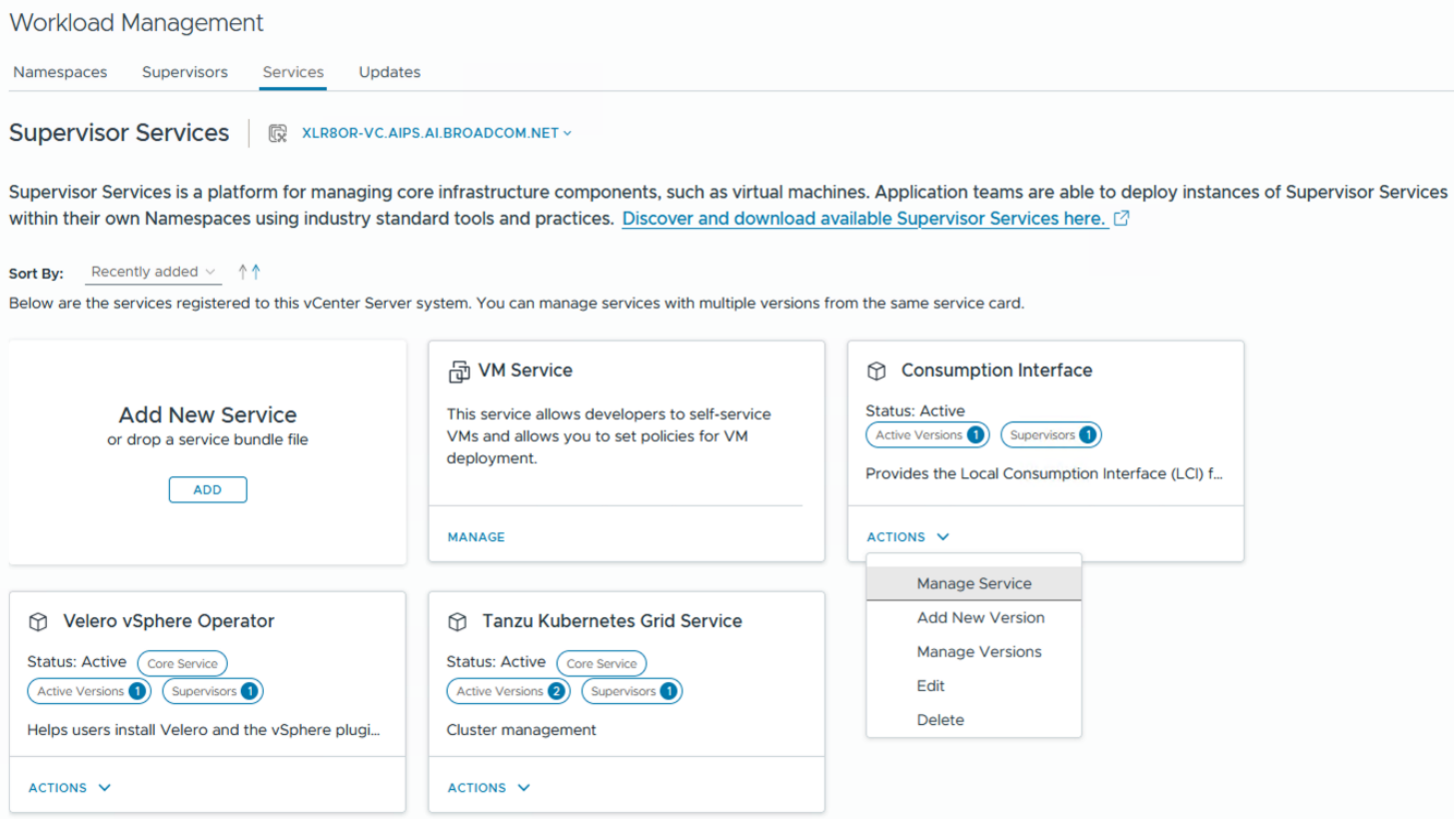
Follow the detailed procedure to configure NSX and enable the Supervisor with VPC, refer to the VMware documentation at https://techdocs.broadcom.com/us/en/vmware-cis/vcf/vcf-9-0-and-later/9-0/vsphere-supervisor-installation-and-configuration/supervisor-networking-with-virtual-private-clouds.html

Enabling the Supervisor with VPC sets up the Supervisor control plane VMs, integrates them with NSX native load balancing for traffic management, and configures VPC-based networking to support isolated and scalable Kubernetes workloads.

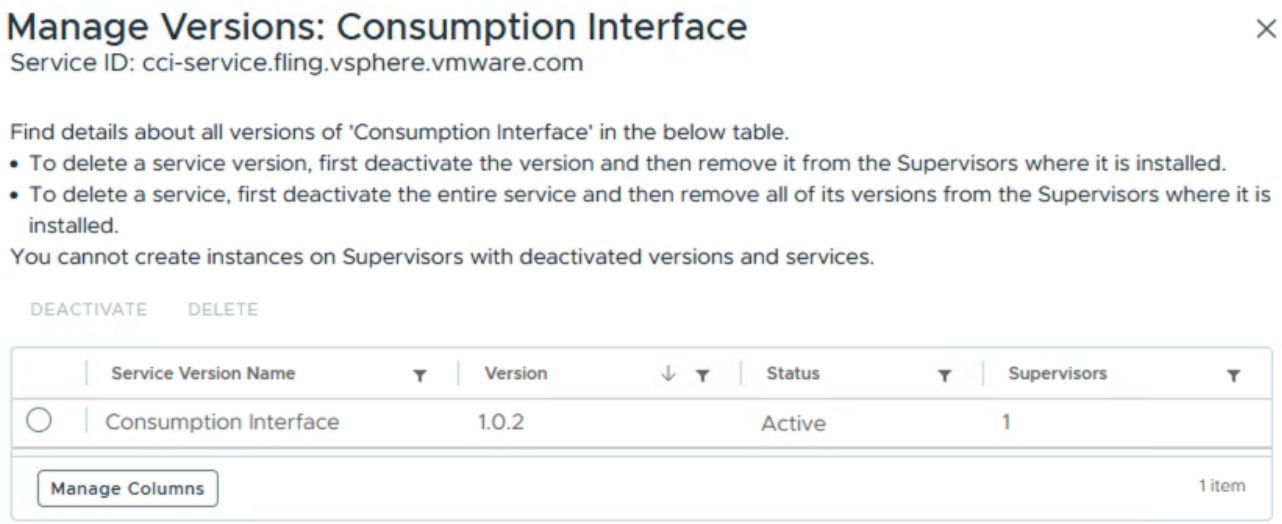
C.3 Deploy Local Consumption Interface (LCI)

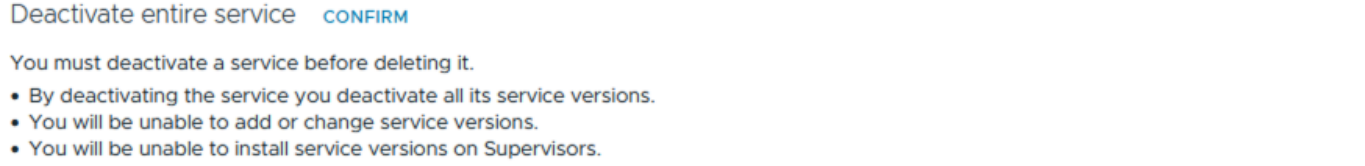
Follow the steps in https://vsphere-tmm.github.io/Supervisor-Services/ to download LCI service YAML file. Then click **Workload Management → Services → Add New Service → Upload the YAML File**.

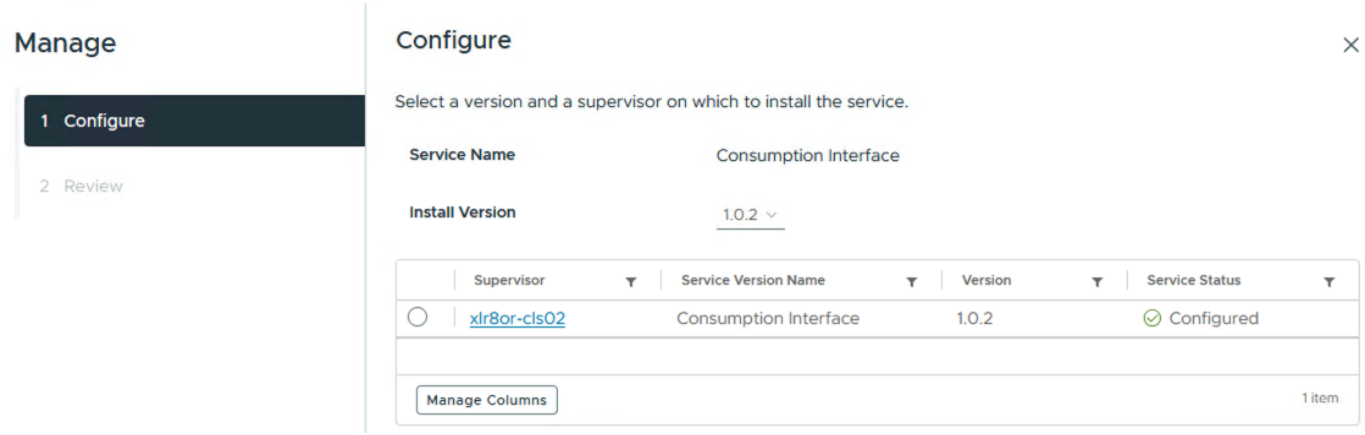
Then the LCI service will show up as demonstrated below.



Manage Service → Activate Consumption Interface service







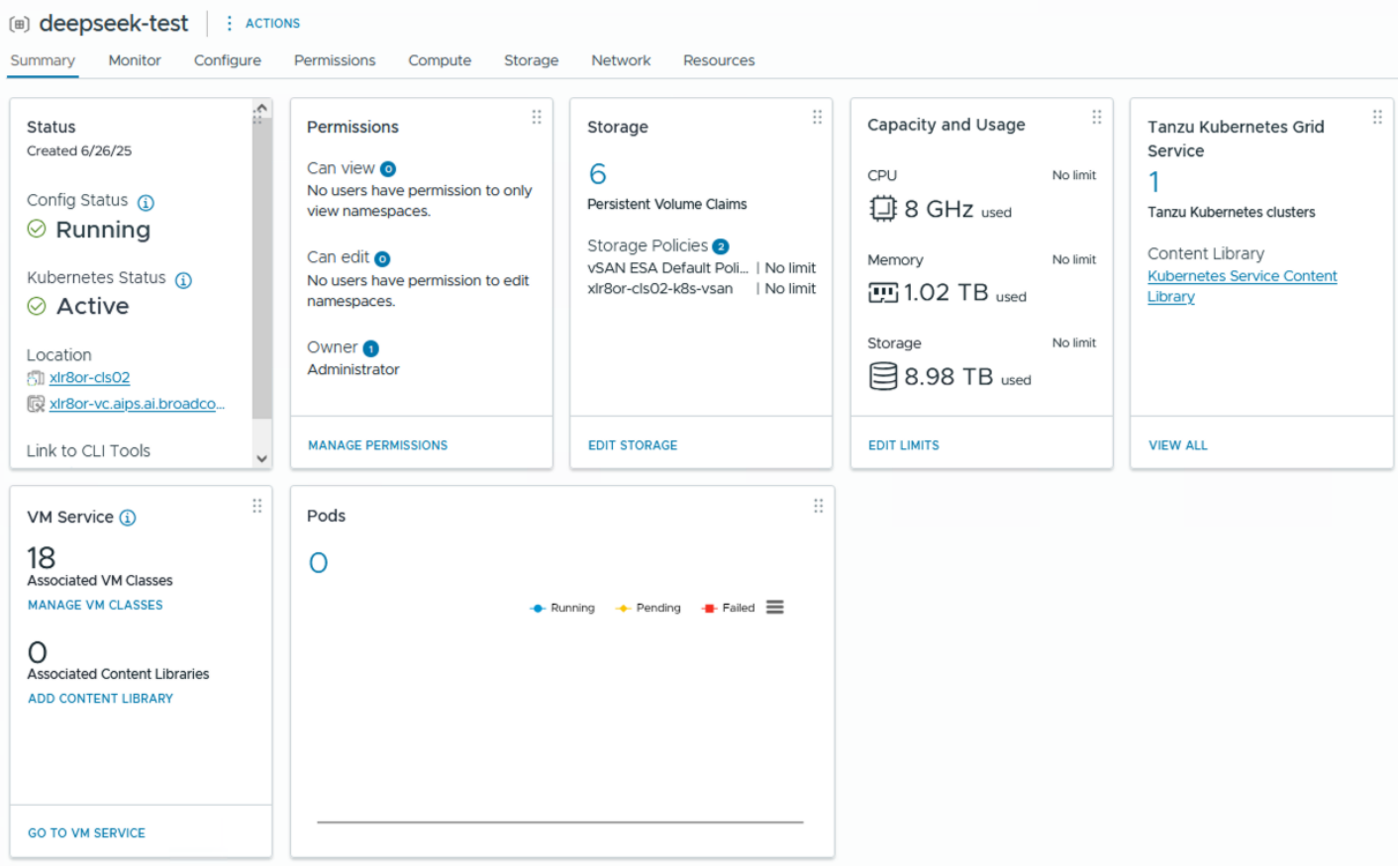
Next, verify that the Cloud Consumption Interface (CCI) service and its pods are visible in the inventory.

Note: In this context, Local Consumption Interface (LCI) and CCI refer to the same component—the terms are used interchangeably in VMware documentation.

C.4 Create a Namespace

Namespaces in VKS are logical constructs used to provide multi-tenancy and resource isolation for developers. Create a Supervisor Namespace to serve as the environment where you will provision and manage workloads.

This namespace will later host your custom **VMClasses**, which define VM sizing, resource allocation, and passthrough device configurations.

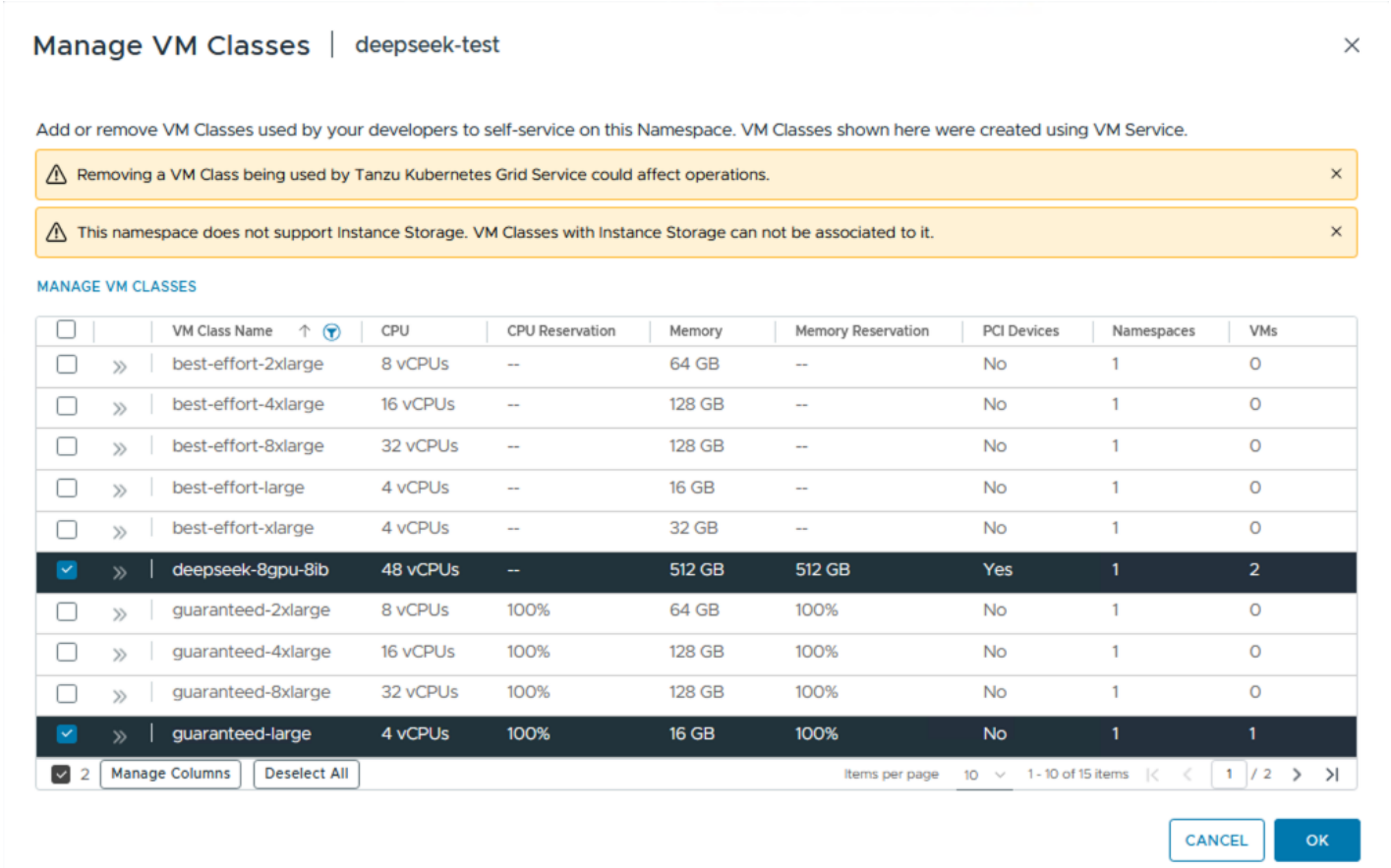


C.5 Manage VM Classes in Namespace

Once the VMClass has been created, assign it to your Supervisor Namespace so that developers and workloads within that namespace can request VMs with the defined passthrough hardware configuration.

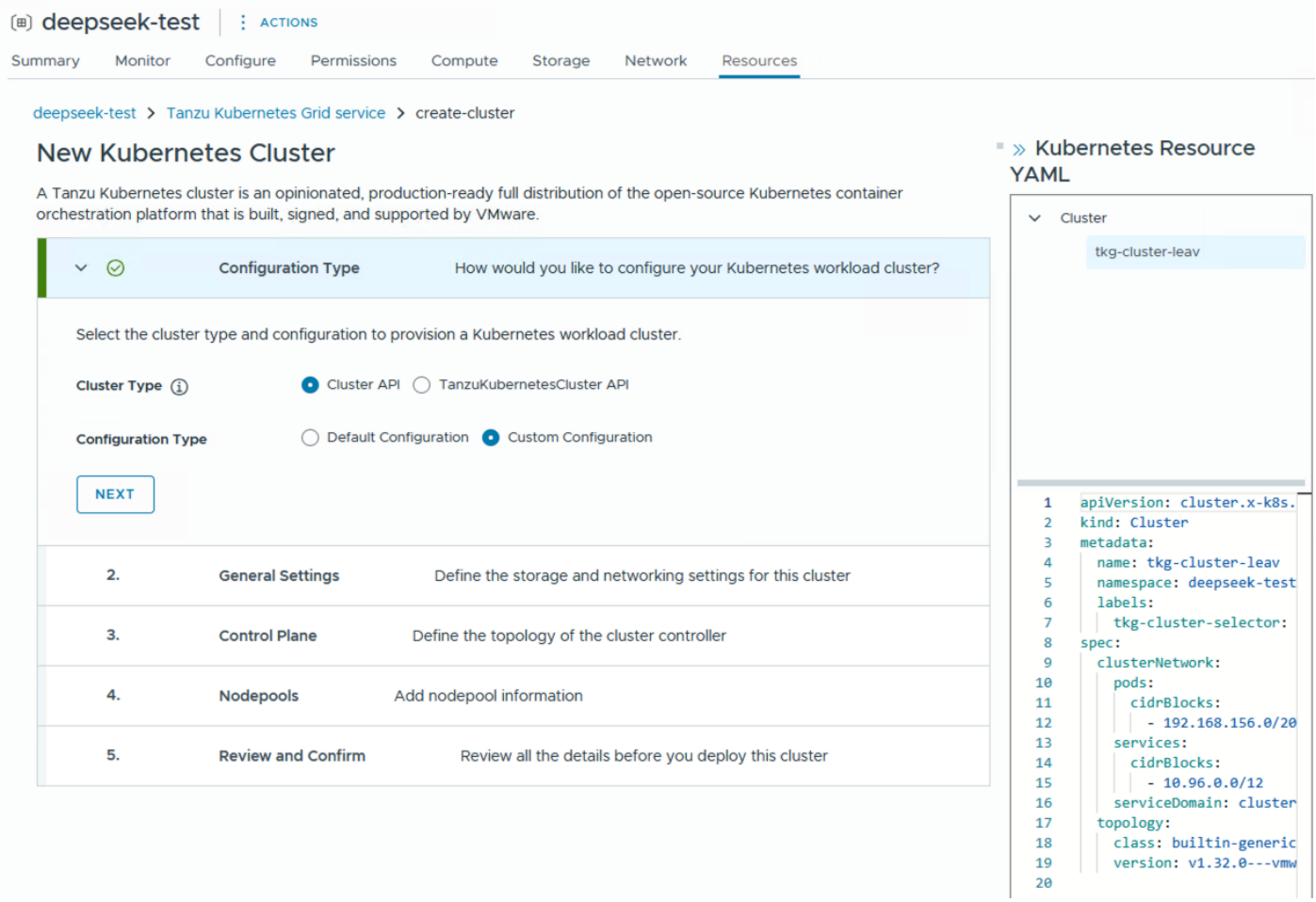
For this environment, two VMClasses will be used in later deployment stages:

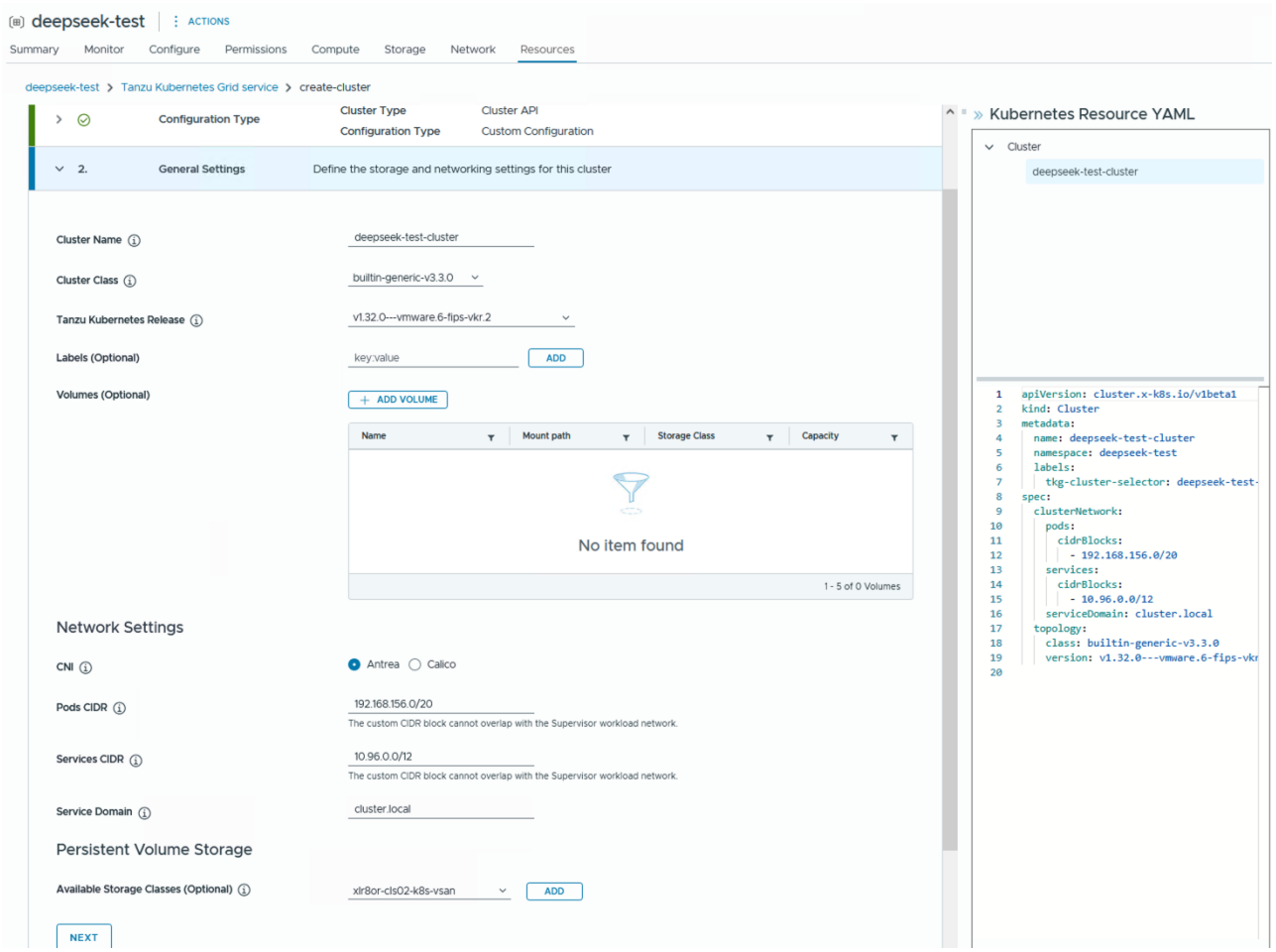
1. An 8xGPU-8xIB VMClass for compute-intensive workloads.
2. A standard VMClass for non-GPU workloads (e.g., management or lightweight applications).

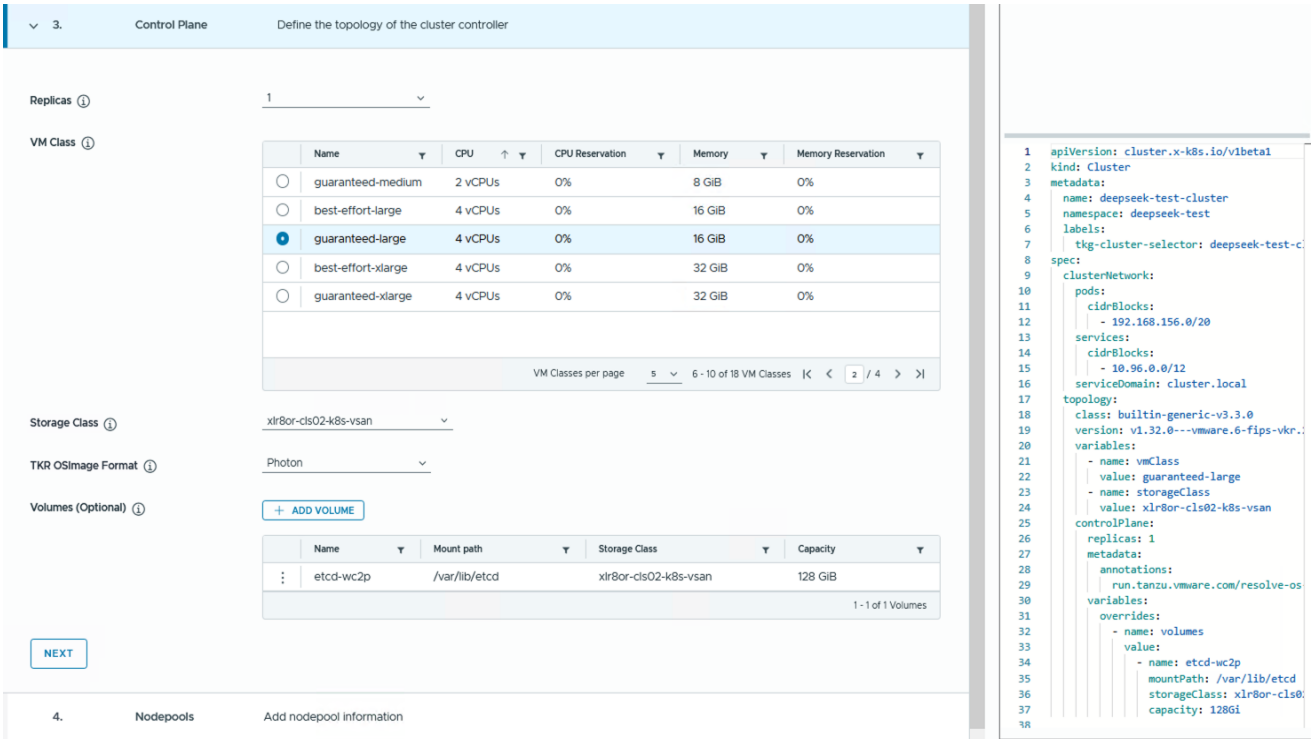


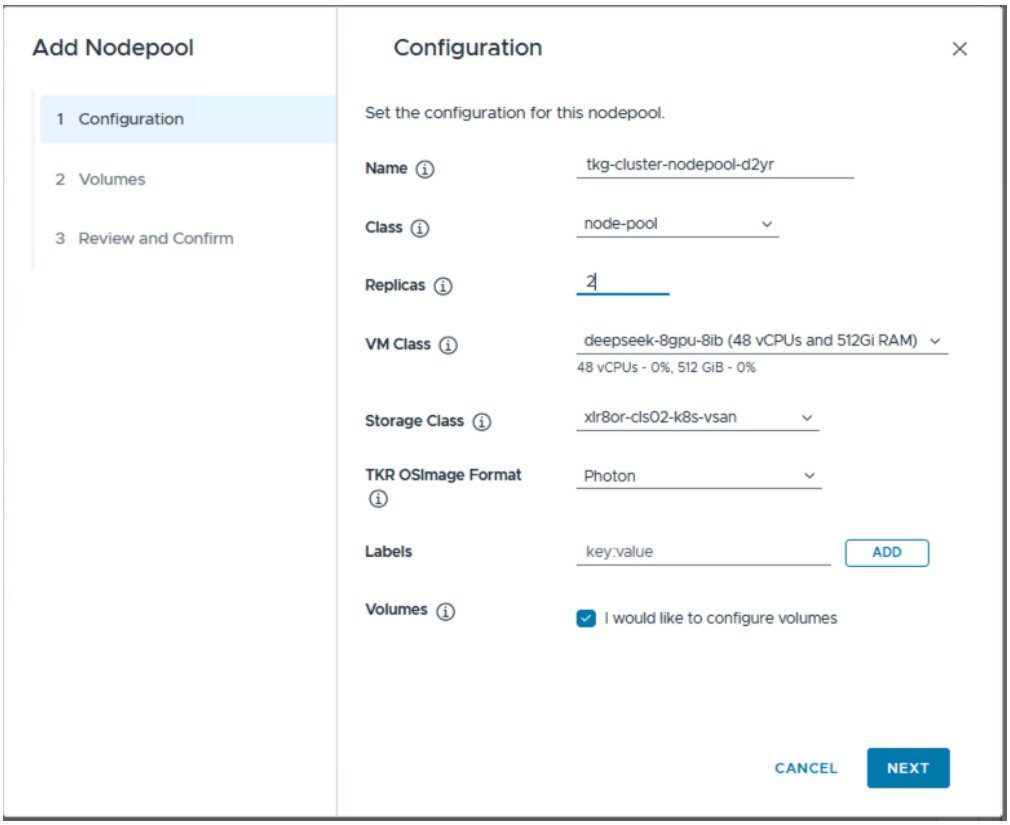
D. Use LCI to deploy a VKS cluster

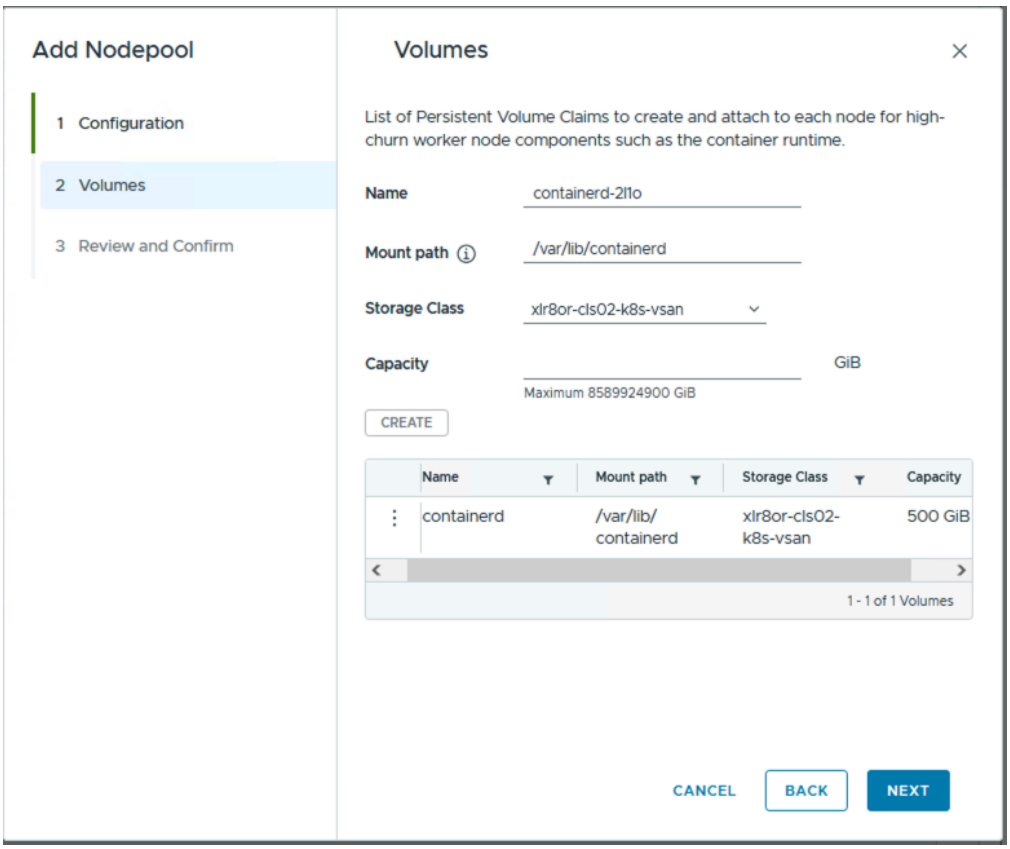
The following is a step-by-step guide to create a VKS cluster in VCF 9 (or a TKG cluster in VCF 5.2.1) by LCI.

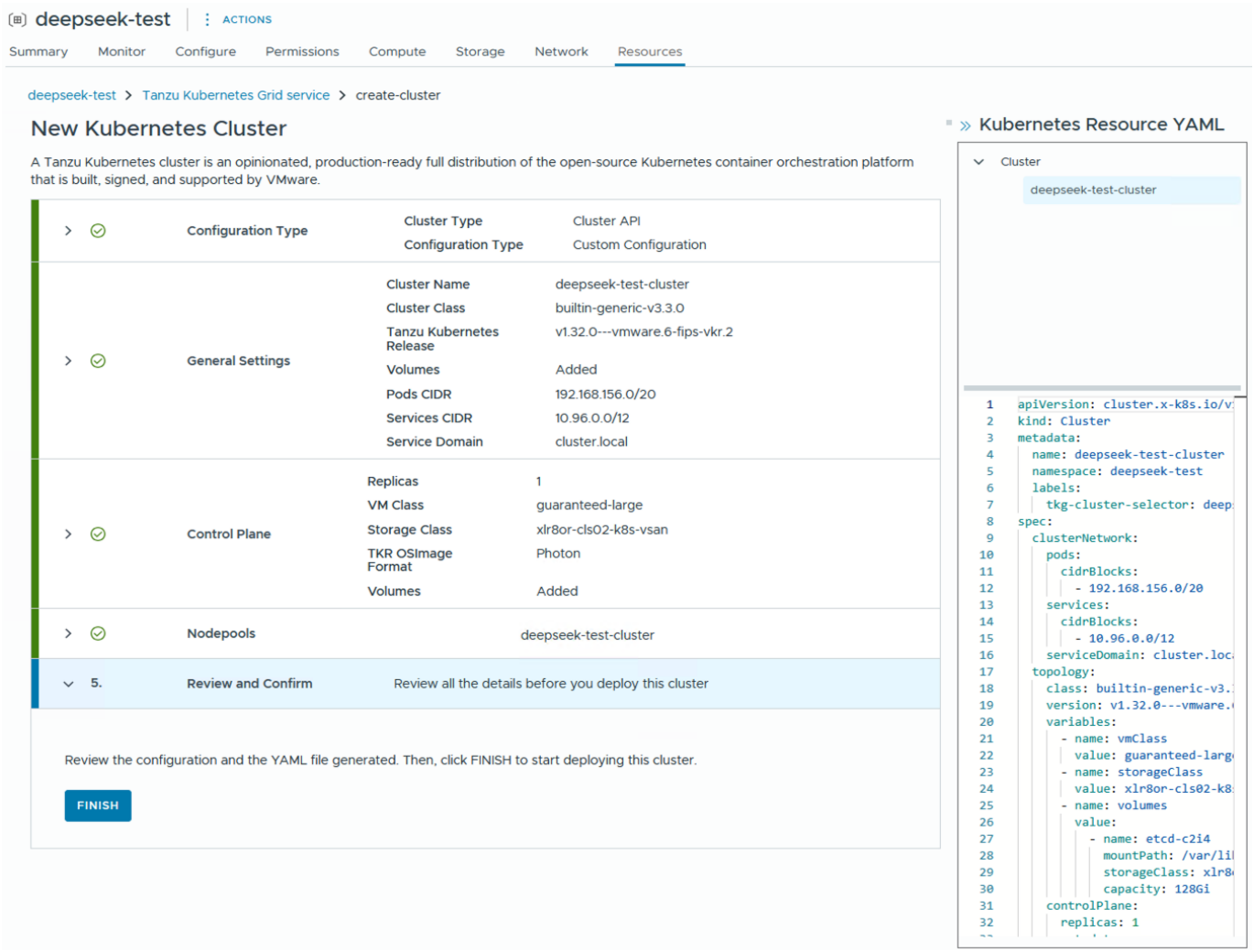



deepseek-test
ACTIONS
Summary
Monitor
Configure
Permissions
Compute
Storage
Network
Resources
deepseek-test > Tanzu Kubernetes Grid service > create-cluster
Configuration Type
Cluster Type
Cluster API
Configuration Type
Custom Configuration
2.
General Settings
Define the storage and networking settings for this cluster
Cluster Name
deepseek-test-cluster
Cluster Class
builtin-generic-v3.3.0
Tanzu Kubernetes Release
v1.32.0---vmware.6-fips-vkr.2
Labels (Optional)
key:value
ADD
Volumes (Optional)
ADD VOLUME
Name
Mount path
Storage Class
Capacity
No item found
1 - 5 of 0 Volumes
Network Settings
CNI
Antrea
Calico
Pods CIDR
192.168.156.0/20
The custom CIDR block cannot overlap with the Supervisor workload network.
Services CIDR
10.96.0.0/12
The custom CIDR block cannot overlap with the Supervisor workload network.
Service Domain
cluster.local
Persistent Volume Storage
Available Storage Classes (Optional)
xlr8or-cls02-k8s-vsan
ADD
NEXT
Kubernetes Resource YAML
Cluster
deepseek-test-cluster
apiVersion: cluster.x-k8s.io/v1beta1
kind: Cluster
metadata:
name: deepseek-test-cluster
namespace: deepseek-test
labels:
tkg-cluster-selector: deepseek-test-
spec:
clusterNetwork:
pods:
cidrBlocks:
- 192.168.156.0/20
services:
cidrBlocks:
- 10.96.0.0/12
serviceDomain: cluster.local
topology:
class: builtin-generic-v3.3.0
version: v1.32.0---vmware.6-fips-vkr


3.
Control Plane
Define the topology of the cluster controller
Replicas
1
VM Class
Name
CPU
CPU Reservation
Memory
Memory Reservation
guaranteed-medium
2 vCPUs
0%
8 GiB
0%
best-effort-large
4 vCPUs
0%
16 GiB
0%
guaranteed-large
4 vCPUs
0%
16 GiB
0%
best-effort-xlarge
4 vCPUs
0%
32 GiB
0%
guaranteed-xlarge
4 vCPUs
0%
32 GiB
0%
VM Classes per page
5
6 - 10 of 18 VM Classes
2 / 4
Storage Class
xlr8or-cls02-k8s-vsan
TKR OSImage Format
Photon
Volumes (Optional)
ADD VOLUME
Name
Mount path
Storage Class
Capacity
etcd-wc2p
/var/lib/etcd
xlr8or-cls02-k8s-vsan
128 GiB
1 - 1 of 1 Volumes
NEXT
4.
Nodepools
Add nodepool information
apiVersion: cluster.x-k8s.io/v1beta1
kind: Cluster
metadata:
name: deepseek-test-cluster
namespace: deepseek-test
labels:
tkg-cluster-selector: deepseek-test-c
spec:
clusterNetwork:
pods:
cidrBlocks:
- 192.168.156.0/20
services:
cidrBlocks:
- 10.96.0.0/12
serviceDomain: cluster.local
topology:
class: builtin-generic-v3.3.0
version: v1.32.0---vmware.6-fips-vkr.
variables:
- name: vmClass
value: guaranteed-large
- name: storageClass
value: xlr8or-cls02-k8s-vsan
controlPlane:
replicas: 1
metadata:
annotations:
run.tanzu.vmware.com/resolve-os
variables:
overrides:
- name: volumes
value:
- name: etcd-wc2p
mountPath: /var/lib/etcd
storageClass: xlr8or-cls0
capacity: 128Gi


Add Nodepool
1 Configuration
2 Volumes
3 Review and Confirm
Configuration
Set the configuration for this nodepool.
Name
tkg-cluster-nodepool-d2yr
Class
node-pool
Replicas
2
VM Class
deepseek-8gpu-8ib (48 vCPUs and 512Gi RAM)
48 vCPUs - 0%, 512 GiB - 0%
Storage Class
xlr8or-cls02-k8s-vsan
TKR OSImage Format
Photon
Labels
key:value
ADD
Volumes
I would like to configure volumes
CANCEL
NEXT


Add Nodepool
1 Configuration
2 Volumes
3 Review and Confirm
Volumes
List of Persistent Volume Claims to create and attach to each node for high-churn worker node components such as the container runtime.
Name
containerd-2l1o
Mount path
/var/lib/containerd
Storage Class
xlr8or-cls02-k8s-vsan
Capacity
GiB
Maximum 8589924900 GiB
CREATE
Name
Mount path
Storage Class
Capacity
containerd
/var/lib/containerd
xlr8or-cls02-k8s-vsan
500 GiB
1 - 1 of 1 Volumes
CANCEL
BACK
NEXT


deepseek-test
ACTIONS
Summary
Monitor
Configure
Permissions
Compute
Storage
Network
Resources
deepseek-test > Tanzu Kubernetes Grid service > create-cluster
New Kubernetes Cluster
A Tanzu Kubernetes cluster is an opinionated, production-ready full distribution of the open-source Kubernetes container orchestration platform that is built, signed, and supported by VMware.
Configuration Type
Cluster Type
Cluster API
Configuration Type
Custom Configuration
General Settings
Cluster Name
deepseek-test-cluster
Cluster Class
builtin-generic-v3.3.0
Tanzu Kubernetes Release
v1.32.0---vmware.6-fips-vkr.2
Volumes
Added
Pods CIDR
192.168.156.0/20
Services CIDR
10.96.0.0/12
Service Domain
cluster.local
Control Plane
Replicas
1
VM Class
guaranteed-large
Storage Class
xlr8or-cls02-k8s-vsan
TKR OSImage Format
Photon
Volumes
Added
Nodepools
deepseek-test-cluster
5.
Review and Confirm
Review all the details before you deploy this cluster
Review the configuration and the YAML file generated. Then, click FINISH to start deploying this cluster.
FINISH
Kubernetes Resource YAML
Cluster
deepseek-test-cluster
apiVersion: cluster.x-k8s.io/v
kind: Cluster
metadata:
name: deepseek-test-cluster
namespace: deepseek-test
labels:
tkg-cluster-selector: deep
spec:
clusterNetwork:
pods:
cidrBlocks:
- 192.168.156.0/20
services:
cidrBlocks:
- 10.96.0.0/12
serviceDomain: cluster.loc
topology:
class: builtin-generic-v3.
version: v1.32.0---vmware.
variables:
- name: vmClass
value: guaranteed-larg
- name: storageClass
value: xlr8or-cls02-k8
- name: volumes
value:
- name: etcd-c2i4
mountPath: /var/li
storageClass: xlr8
capacity: 128Gi
controlPlane:
replicas: 1

E. Deploy Network Operator and GPU Operator

To enable high-performance GPU workloads with RDMA and InfiniBand support in Kubernetes, both the **NVIDIA Network Operator** and the **NVIDIA GPU Operator** need to be installed. The recommended sequence is to deploy the **Network Operator first**, followed by the **GPU Operator**, since GPU features such as GPUDirect RDMA depend on the networking components.

E.1 Login to VKS cluster

```
# Log into supervisor cluster:
kubectl vsphere login --server=10.191.83.194 --vsphere-username administrator@vsphere.local --insecure-skip-tls-verify

# Log into workload cluster:
kubectl vsphere login --server=10.191.83.194 --tanzu-kubernetes-cluster-name deepseek-test-cluster --tanzu-kubernetes-cluster-namespace deepseek-test --vsphere-username administrator@vsphere.local --insecure-skip-tls-verify
```

E.2 Install Helm

Helm is a package manager for Kubernetes that simplifies the deployment and management of applications. It uses "charts" which are packages of pre-configured Kubernetes resources.

```
# Add the GPG key for the Helm repository
curl https://baltocdn.com/helm/signing.asc | gpg --dearmor | sudo tee /usr/share/keyrings/helm.gpg > /dev/null

# Install apt-transport-https to allow apt to retrieve packages over HTTPS
sudo apt-get install apt-transport-https --yes

# Add the Helm stable Debian repository to your apt sources list
echo "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/helm.gpg]
https://baltocdn.com/helm/stable/debian/ all main" | sudo tee /etc/apt/sources.list.d/helm-stable-debian.list

# Update your package list to include the new Helm repository
sudo apt-get update

# Install Helm
sudo apt-get install helm

# Verify the Helm installation by checking its version
helm version
```

E.3 Install the NVIDIA Network Operator

The Network Operator automates the deployment of network components (e.g., RDMA, SR-IOV device plugins, Multus CNI) required for GPU communication and high-speed networking.

```
# Add NVIDIA Helm repo
sudo helm repo add nvidia https://helm.ngc.nvidia.com/nvidia

# Create namespace for network-operator
sudo kubectl create ns nvidia-network-operator

# Label namespace to allow privileged workloads
sudo kubectl label --overwrite ns nvidia-network-operator \
  pod-security.kubernetes.io/enforce=privileged

# Install the Network Operator
sudo helm install network-operator nvidia/network-operator \
   -n nvidia-network-operator \
   --create-namespace \
   --version v25.4.0 \
   --wait

# Verify the deployment
kubectl -n nvidia-network-operator get pods
```

You should see the **network-operator** and **node-feature-discovery** pods in a `Running` state, for example:

```
kubectl -n nvidia-network-operator get pods
NAME                                                  READY   STATUS    RESTARTS   AGE
network-operator-798476bc67-zbwck                     1/1     Running   0          66s
network-operator-node-feature-discovery-gc-5549bd5db-mx5t4        1/1     Running   0          66s
network-operator-node-feature-discovery-master-865bfff66d-smtd8   1/1     Running   0          66s
network-operator-node-feature-discovery-worker-q7tpn              1/1     Running   0          67s
network-operator-node-feature-discovery-worker-vtjg9              1/1     Running   0          67s
network-operator-node-feature-discovery-worker-wgqlr              1/1     Running   0          67s
```

E.4 Install the NVIDIA GPU Operator

The GPU Operator manages all NVIDIA software components required to expose GPUs in Kubernetes, including:

- Driver installation
- Container runtime hooks (NVIDIA Container Toolkit)
- Device plugins
- Monitoring (DCGM Exporter)
- MIG management (if supported)

Deploy the GPU Operator (after the Network Operator is running):

```
# Install GPU operator
helm install --wait --generate-name \
     -n gpu-operator --create-namespace \
     nvidia/gpu-operator \
     --version=v25.3.0 \
     --set driver.rdma.enabled=true
```

The `driver.rdma.enabled=true` flag ensures RDMA support is enabled, which is critical for GPU workloads leveraging InfiniBand or RoCE.

E.5 Sanity check

Once installed, confirm that all pods in the `gpu-operator` namespace are either **Running** or **Completed** (for validator jobs):

```
kubectl -n gpu-operator get pods
```

You should see components such as:

- `nvidia-driver-daemonset` (GPU drivers)
- `nvidia-device-plugin-daemonset` (device plugins for Kubernetes)
- `nvidia-container-toolkit-daemonset` (runtime hooks)
- `nvidia-dcgm-exporter` (GPU monitoring)
- `nvidia-mig-manager` (if MIG is enabled)
- `nvidia-operator-validator` and `nvidia-cuda-validator` (sanity checks)

Sample output:

```
# Sanity Check
# Wait for all Status shows completed / running (instead of showing init or PodInitializing)
k -n gpu-operator get pods
NAME                                                  READY   STATUS      RESTARTS   AGE
gpu-feature-discovery-j6t68                           1/1     Running     0          3m47s
gpu-feature-discovery-zjlm8                           1/1     Running     0          3m45s
gpu-operator-666bbffcd-bd7g7                          1/1     Running     0          4m17s
gpu-operator-node-feature-discovery-gc-7c7f68d5f4-fn4pl       1/1     Running     0          4m17s
gpu-operator-node-feature-discovery-master-58588c6967-hs5dv   1/1     Running     0          4m17s
gpu-operator-node-feature-discovery-worker-29cz2      1/1     Running     0          4m17s
gpu-operator-node-feature-discovery-worker-2jhqm      1/1     Running     0          4m17s
gpu-operator-node-feature-discovery-worker-msmxr      1/1     Running     0          4m17s
nvidia-container-toolkit-daemonset-kznzg              1/1     Running     0          3m45s
nvidia-container-toolkit-daemonset-zfxz4              1/1     Running     0          3m47s
nvidia-cuda-validator-ps7z9                           0/1     Completed   0          75s
nvidia-cuda-validator-x27vg                           0/1     Completed   0          48s
nvidia-dcgm-exporter-8dhrd                            1/1     Running     0          3m45s
nvidia-dcgm-exporter-tgdqg                            1/1     Running     0          3m47s
nvidia-device-plugin-daemonset-6gbdm                  1/1     Running     0          3m47s
nvidia-device-plugin-daemonset-b6hsz                  1/1     Running     0          3m45s
nvidia-driver-daemonset-g8kj5                         1/1     Running     0          4m7s
nvidia-driver-daemonset-qvf9k                         1/1     Running     0          4m7s
nvidia-mig-manager-2l8mk                              1/1     Running     0          20s
nvidia-mig-manager-gsrmt                              1/1     Running     0          16s
nvidia-operator-validator-bz5p2                       1/1     Running     0          3m47s
nvidia-operator-validator-gkh99                       1/1     Running     0          3m45s
```

E.6 Deploy NicClusterPolicy CRD

To enable RDMA capabilities across the cluster, we deploy a **NicClusterPolicy**. This custom resource definition (CRD) configures both the **Mellanox OFED drivers** and the **RDMA Shared Device Plugin**, allowing Kubernetes pods to leverage InfiniBand devices efficiently.

The example `nicClusterPolicy-rdma-share-dp-separate.yaml` defines:

- OFED driver:
  - Uses the `doca-driver` from NVIDIA's Mellanox repository (`nvcr.io/nvidia/mellanox:25.04-0.6.1.0-2`).
  - Supports automatic upgrades with controlled pod draining and parallelism.
  - Includes startup, liveness, and readiness probes to ensure reliability.
- RDMA shared device plugin:
  - Deploys the plugin (`ghcr.io/mellanox/k8s-rdma-shared-dev-plugin:v1.5.3`) to expose virtualized RDMA devices to pods.
  - Configures **multiple RDMA resource groups** (`rdma_shared_devices_a` through `rdma_shared_devices_h`) with dedicated selectors for vendor ID, device ID, driver, and interface name.
  - Each group can expose up to **63 virtual RDMA devices**, enabling high-density GPU-to-NIC connectivity.

Each RDMA group is mapped to a specific InfiniBand interface (`ibs65`, `ibs67`, etc.), ensuring **dedicated virtual RDMA devices for workloads** without conflicts.

`nicClusterPolicy-rdma-share-dp-separate.yaml`

```
apiVersion: mellanox.com/v1alpha1
kind: NicClusterPolicy
metadata:
  name: nic-cluster-policy
  namespace: nvidia-network-operator
spec:
  ofedDriver:
    image: doca-driver
    repository: nvcr.io/nvidia/mellanox
    version: 25.04-0.6.1.0-2
    upgradePolicy:
      autoUpgrade: true
      drain:
        deleteEmptyDir: true
        enable: true
        force: true
        timeoutSeconds: 300
      maxParallelUpgrades: 1
    startupProbe:
      initialDelaySeconds: 10
      periodSeconds: 20
    livenessProbe:
```

```
      initialDelaySeconds: 30
      periodSeconds: 30
    readinessProbe:
      initialDelaySeconds: 10
      periodSeconds: 30
  rdmaSharedDevicePlugin:
    image: k8s-rdma-shared-dev-plugin
    repository: ghcr.io/mellanox
    version: v1.5.3
    imagePullSecrets: []
    config: |
      {
        "configList": [
          {
            "resourceName": "rdma_shared_devices_a",
            "rdmaHcaMax": 63,
            "selectors": {
              "vendors": ["15b3"],
              "deviceIDs": ["1021"],
              "drivers": ["mlx5_core"],
              "ifNames": ["ibs65"],
              "linkTypes": []
            }
          },
          {
            "resourceName": "rdma_shared_devices_b",
            "rdmaHcaMax": 63,
            "selectors": {
              "vendors": ["15b3"],
              "deviceIDs": ["1021"],
              "drivers": ["mlx5_core"],
              "ifNames": ["ibs67"],
              "linkTypes": []
            }
          },
          {
            "resourceName": "rdma_shared_devices_c",
            "rdmaHcaMax": 63,
            "selectors": {
              "vendors": ["15b3"],
              "deviceIDs": ["1021"],
              "drivers": ["mlx5_core"],
              "ifNames": ["ibs69"],
              "linkTypes": []
            }
          },
```

```
    {
      "resourceName": "rdma_shared_devices_d",
      "rdmaHcaMax": 63,
      "selectors": {
        "vendors": ["15b3"],
        "deviceIDs": ["1021"],
        "drivers": ["mlx5_core"],
        "ifNames": ["ibs71"],
        "linkTypes": []
      }
    },
    {
      "resourceName": "rdma_shared_devices_e",
      "rdmaHcaMax": 63,
      "selectors": {
        "vendors": ["15b3"],
        "deviceIDs": ["1021"],
        "drivers": ["mlx5_core"],
        "ifNames": ["ibs73"],
        "linkTypes": []
      }
    },
    {
      "resourceName": "rdma_shared_devices_f",
      "rdmaHcaMax": 63,
      "selectors": {
        "vendors": ["15b3"],
        "deviceIDs": ["1021"],
        "drivers": ["mlx5_core"],
        "ifNames": ["ibs75"],
        "linkTypes": []
      }
    },
    {
      "resourceName": "rdma_shared_devices_g",
      "rdmaHcaMax": 63,
      "selectors": {
        "vendors": ["15b3"],
        "deviceIDs": ["1021"],
        "drivers": ["mlx5_core"],
        "ifNames": ["ibs77"],
        "linkTypes": []
      }
    },
    {
      "resourceName": "rdma_shared_devices_h",
```

```
            "rdmaHcaMax": 63,
            "selectors": {
              "vendors": ["15b3"],
              "deviceIDs": ["1021"],
              "drivers": ["mlx5_core"],
              "ifNames": ["ibs79"],
              "linkTypes": []
            }
          }
        ]
      }
```

After applying the NicClusterPolicy, you can list the pods in the `nvidia-network-operator` namespace to confirm that the **RDMA Shared Device Plugin DaemonSets** are running:

```
# kubectl -n nvidia-network-operator get pods
NAME                                                          READY   STATUS    RESTARTS   AGE
mofed-ubuntu22.04-5dd9bbcc4d-ds-dfhlt                         1/1     Running   0          32d
mofed-ubuntu22.04-5dd9bbcc4d-ds-kctcb                         1/1     Running   0          32d
network-operator-798476bc67-8kqqj                             1/1     Running   0          32d
network-operator-node-feature-discovery-gc-5549bd5db-mnx6r    1/1     Running   0          32d
network-operator-node-feature-discovery-master-865bfff66d-tc46q   1/1     Running   0          32d
network-operator-node-feature-discovery-worker-4rdvs          1/1     Running   0          32d
network-operator-node-feature-discovery-worker-vpgtl          1/1     Running   0          32d
network-operator-node-feature-discovery-worker-xw4mw          1/1     Running   0          32d
rdma-shared-dp-ds-6t8m9                                       1/1     Running   0          32d
rdma-shared-dp-ds-phg6g                                       1/1     Running   0          32d
```

These pods indicate that the RDMA shared device plugin is active, making virtualized RDMA devices available for Kubernetes workloads.

F. Verify RDMA performance via IB on two pods across two HGX nodes

Create two test pod YAML files .

Pod 1: `test-mofed-pod1.yaml`

```
# test-mofed-pod1.yaml
apiVersion: v1
kind: Pod
metadata:
  name: mofed-test-pod-1
spec:
  restartPolicy: OnFailure
  containers:
  - image: mellanox/mofed:23.10-1.1.9.0-ubuntu22.04-amd64
    name: mofed-test-ctr
    securityContext:
      capabilities:
        add: [ "IPC_LOCK", "SYS_RESOURCE" ]
    resources:
      requests:
        rdma/rdma_shared_devices_a: 1
        rdma/rdma_shared_devices_b: 1
        rdma/rdma_shared_devices_c: 1
        rdma/rdma_shared_devices_d: 1
        rdma/rdma_shared_devices_e: 1
        rdma/rdma_shared_devices_f: 1
        rdma/rdma_shared_devices_g: 1
        rdma/rdma_shared_devices_h: 1
    command:
    - sh
    - -c
    - |
      ls -l /dev/infiniband /sys/class/infiniband /sys/class/net
      sleep 1000000
```

Pod 2: **`test-mofed-pod2.yaml`** (identical, name and IP differ)

Then we can validate RDMA performance using InfiniBand between two pods on separate HGX nodes, follow these steps,

```
# 1. Deploy two test pods
k apply -f test-mofed-pod1.yaml
k apply -f test-mofed-pod2.yaml
```

```
# Check that the pods are running and note their node assignments
k get pods -o wide
NAME                 READY   STATUS    RESTARTS   AGE     IP               NODE
NOMINATED NODE   READINESS GATES
mofed-test-pod-1   1/1     Running   0          10m     192.168.146.75   deepseek-test-cluster-gpu-
x9d2k-82mqg-kh5jm   <none>           <none>
mofed-test-pod-2   1/1     Running   0          5m36s   192.168.145.61   deepseek-test-cluster-gpu-
x9d2k-82mqg-xh2bq   <none>           <none>

# 2. Inspect Mellanox CX-7 Devices
# Login to Pod 1 interactively
k exec -it mofed-test-pod-1 -- bash

# Check CX-7 devices
lspci | grep -i mellanox
04:01.0 Ethernet controller: Mellanox Technologies MT2910 Family [ConnectX-7]
04:03.0 Ethernet controller: Mellanox Technologies MT2910 Family [ConnectX-7]
04:05.0 Ethernet controller: Mellanox Technologies MT2910 Family [ConnectX-7]
04:07.0 Ethernet controller: Mellanox Technologies MT2910 Family [ConnectX-7]
04:09.0 Ethernet controller: Mellanox Technologies MT2910 Family [ConnectX-7]
04:0b.0 Ethernet controller: Mellanox Technologies MT2910 Family [ConnectX-7]
04:0d.0 Ethernet controller: Mellanox Technologies MT2910 Family [ConnectX-7]
04:0f.0 Ethernet controller: Mellanox Technologies MT2910 Family [ConnectX-7]

# 3. Run Bandwidth Test
# On Pod 1 (server):
# -F, --CPU-freq  suppresses CPU frequency warnings even if cpufreq_ondemand is loaded.

root@mofed-test-pod-1:~# ib_send_bw -d mlx5_0 -a -F --report_gbits -q 4

************************************
* Waiting for client to connect... *
************************************


# On Pod 2 (client):
k exec -it mofed-test-pod-2 -- bash

root@mofed-test-pod-2:~# ib_send_bw -d mlx5_1 -a  --report_gbits -q 4 192.168.146.75
---------------------------------------------------------------------------------------
                    Send BW Test
 Dual-port       : OFF          Device         : mlx5_1
 Number of qps   : 4            Transport type : IB
 Connection type : RC           Using SRQ      : OFF
 PCIe relax order: ON
 ibv_wr* API     : ON
 TX depth        : 128
```

```
 CQ Moderation   : 100
 Mtu             : 4096[B]
 Link type       : IB
 Max inline data : 0[B]
 rdma_cm QPs     : OFF
 Data ex. method : Ethernet
---------------------------------------------------------------------------------------
 local address: LID 0x27 QPN 0x0046 PSN 0xa8fad9
 local address: LID 0x27 QPN 0x0047 PSN 0x252e4f
 local address: LID 0x27 QPN 0x0048 PSN 0xb532ed
 local address: LID 0x27 QPN 0x0049 PSN 0x8a3c38
 remote address: LID 0x12 QPN 0x005c PSN 0xb304a3
 remote address: LID 0x12 QPN 0x005d PSN 0x6ed551
 remote address: LID 0x12 QPN 0x005e PSN 0xf51ac7
 remote address: LID 0x12 QPN 0x005f PSN 0x5d00a
---------------------------------------------------------------------------------------
 #bytes     #iterations    BW peak[Gb/sec]    BW average[Gb/sec]   MsgRate[Mpps]
 2          4000           0.066253            0.064243            4.015187
 4          4000             0.13               0.13              4.176859
 8          4000             0.27               0.26              4.029413
 16         4000             0.53               0.53              4.106850
 32         4000             1.07               1.07              4.164123
 64         4000             2.14               2.13              4.158780
 128        4000             4.28               4.27              4.166929
 256        4000             8.55               8.44              4.120146
 512        4000             17.17              17.09             4.173372
 1024       4000             34.35              34.20             4.174527
 2048       4000             68.70              68.37             4.173133
 4096       4000             134.30             132.34            4.038705
 8192       4000             270.25             269.50            4.112174
 16384      4000             391.26             390.76            2.981258
 32768      4000             394.50             394.25            1.503944
 65536      4000             395.24             395.19            0.753771
 131072     4000             396.06             396.02            0.377673
 262144     4000             396.29             396.26            0.188953
 524288     4000             396.38             396.38            0.094504
 1048576    4000             396.45             396.44            0.047259
 2097152    4000             396.47             396.47            0.023632
 4194304    4000             396.49             396.49            0.011816
 8388608    4000             396.50             396.50            0.005908
---------------------------------------------------------------------------------------
```

> We achieved an expected peak performance of 396.5 GB/s across 2 nodes without GPU involvement, closely aligning with the CX-7 line rate of 400 Gb/s.

```
# 4. Cleanup: Remove the test pods after validation
kubectl delete pods --all -n deepseek
```

G. Verify GPUDirect RDMA performance via IB on 2 pods across 2 HGX nodes

Create two test Pod YAML files.

Pod 1: `gdr-test-pod-1.yaml`

```
apiVersion: v1
kind: Pod
metadata:
  name: gdr-test-pod-1
  annotations:
    k8s.v1.cni.cncf.io/networks: hostdev-net
spec:
  restartPolicy: OnFailure
  containers:
  - name: cuda-perftest-ctr
    image: mellanox/cuda-perftest:latest
    imagePullPolicy: IfNotPresent
    securityContext:
      capabilities:
        add: [ "IPC_LOCK" ]
    resources:
      requests:
        rdma/rdma_shared_devices_a: 1
        rdma/rdma_shared_devices_b: 1
        rdma/rdma_shared_devices_c: 1
        rdma/rdma_shared_devices_d: 1
        rdma/rdma_shared_devices_e: 1
        rdma/rdma_shared_devices_f: 1
        rdma/rdma_shared_devices_g: 1
        rdma/rdma_shared_devices_h: 1
        nvidia.com/gpu: 8

    command:
    - sh
    - -c
    - sleep inf
```

Pod 2: gdr-test-pod-2.yaml (identical, except name is gdr-test-pod-2)

To validate **GPUDirect RDMA** performance via InfiniBand, you must set `MLX5_SCATTER_TO_CQE=0`.

```
# 1. Deploy test pods

# Refer to the YAML files in Appendix E:
kubectl apply -f gdr-test-pod-1.yaml
kubectl apply -f gdr-test-pod-2.yaml
```

```
# Check pod status and node assignment
kubectl get pods -o wide

# 2. Run GPUDirect RDMA bandwidth test

# On Pod 1 (server):
root@gdr-test-pod-1:~# MLX5_SCATTER_TO_CQE=0 ib_send_bw -a --report_gbits -d mlx5_0 --use_cuda=0 -q
4

************************************
* Waiting for client to connect... *
************************************

# On Pod 2 (client):
root@gdr-test-pod-2:~# MLX5_SCATTER_TO_CQE=0 ib_send_bw -a --report_gbits -d mlx5_0 --use_cuda=0 -q
4 192.168.146.75
initializing CUDA
Listing all CUDA devices in system:
CUDA device 0: PCIe address is 04:00
CUDA device 1: PCIe address is 04:02
CUDA device 2: PCIe address is 04:04
CUDA device 3: PCIe address is 04:06
CUDA device 4: PCIe address is 04:08
CUDA device 5: PCIe address is 04:0A
CUDA device 6: PCIe address is 04:0C
CUDA device 7: PCIe address is 04:0E

Picking device No. 0
[pid = 31, dev = 0] device name = [NVIDIA H100 80GB HBM3]
creating CUDA Ctx
making it the current CUDA Ctx
cuMemAlloc() of a 67108864 bytes GPU buffer
allocated GPU buffer address at 00007f6a1c000000 pointer=0x7f6a1c000000
---------------------------------------------------------------------------------------
                    Send BW Test
 Dual-port       : OFF          Device         : mlx5_0
 Number of qps   : 4            Transport type : IB
 Connection type : RC           Using SRQ      : OFF
 PCIe relax order: ON
 ibv_wr* API     : ON
 TX depth        : 128
 CQ Moderation   : 100
 Mtu             : 4096[B]
 Link type       : IB
 Max inline data : 0[B]
 rdma_cm QPs     : OFF
```

```
 Data ex. method : Ethernet
---------------------------------------------------------------------------------------
 local address: LID 0x0c QPN 0x0046 PSN 0x3562d8
 local address: LID 0x0c QPN 0x0047 PSN 0xe70319
 local address: LID 0x0c QPN 0x0048 PSN 0x2bd5b9
 local address: LID 0x0c QPN 0x0049 PSN 0xea61dd
 remote address: LID 0x12 QPN 0x0060 PSN 0x90ae79
 remote address: LID 0x12 QPN 0x0061 PSN 0x24dca6
 remote address: LID 0x12 QPN 0x0062 PSN 0xf224c2
 remote address: LID 0x12 QPN 0x0063 PSN 0xa81eb2
---------------------------------------------------------------------------------------
 #bytes     #iterations    BW peak[Gb/sec]    BW average[Gb/sec]   MsgRate[Mpps]
 2          4000           0.064516            0.063100            3.943768
 4          4000             0.13              0.13              4.023487
 8          4000             0.26              0.26              4.070511
 16         4000             0.52              0.52              4.072181
 32         4000             1.04              1.03              4.040300
 64         4000             2.09              2.09              4.072563
 128        4000             4.20              4.18              4.084949
 256        4000             8.38              8.35              4.077000
 512        4000             16.68             16.21             3.957823
 1024       4000             33.51             33.38             4.074230
 2048       4000             67.01             66.81             4.077857
 4096       4000             134.02            133.50            4.074127
 8192       4000             267.49            266.37            4.064415
 16384      4000             393.02            392.49            2.994425
 32768      4000             394.79            394.75            1.505861
 65536      4000             395.69            395.60            0.754548
 131072     4000             396.14            396.12            0.377770
 262144     4000             396.32            396.32            0.188979
 524288     4000             396.42            396.41            0.094512
 1048576    4000             396.46            396.46            0.047262
 2097152    4000             396.48            396.48            0.023632
 4194304    4000             396.50            396.50            0.011817
 8388608    4000             396.50            396.50            0.005908
---------------------------------------------------------------------------------------
deallocating RX GPU buffer 00007f6a1c000000
destroying current CUDA Ctx
```

We achieved an expected peak performance of 396.5 GB/s across 2 nodes without GPU involvement, closely aligning with the CX-7 line rate of 400 Gb/s.

```
# 3. Cleanup
kubectl delete pods --all -n deepseek
```

H. Verify NCCL performance on two pods in VKS

The Docker file for CUDA + NCCL + MPI development environment follows.

```
# Dockerfile for CUDA + NCCL + MPI development environment
# Base image
FROM nvcr.io/nvidia/cuda:12.8.0-devel-ubuntu22.04

# Build arguments
ARG OMPI_VERSION=4.1.8
ARG SSH_PORT=22
ARG ROOT_PASSWORD=your_secure_password

# Labels for metadata
LABEL maintainer="Yuankun Fu"
LABEL description="Development environment with CUDA, NCCL, and OpenMPI support"
LABEL version="0.1"

# Environment variables
ENV
LD_LIBRARY_PATH=/usr/local/nvidia/lib:/usr/local/nvidia/lib64:/workspace/ucx/ucx_install/lib:/workspace/ompi/omp
i_install/lib:/usr/local/lib:$LD_LIBRARY_PATH \
    PATH=/usr/local/bin:/workspace/ucx/ucx_install/bin:/workspace/ompi/ompi_install/bin:$PATH \
    NCCL_DEBUG=VERSION

# Set working directory
WORKDIR /workspace

# Copy all configuration and script files at once
# This creates a single layer for all static files
COPY vm_topo_8h100_8ib_mod.xml \
     run_gdr.sh \
     run_nccl_test.sh \
     run_2nodes_nccl_test.sh \
     /workspace/

# Set proper permissions for scripts
RUN chmod +x /workspace/*.sh && \
    # Validate script syntax
    bash -n /workspace/*.sh

# Install system dependencies
RUN apt-get update && \
    DEBIAN_FRONTEND=noninteractive apt-get install -y --no-install-recommends \
        # Build essentials
        build-essential \
        autoconf \
        automake \
        libtool \
        devscripts \
        debhelper \
        fakeroot \
```

```
        pkg-config \
        dkms \
        # Development tools
        git \
        wget \
        vim \
        unzip \
        plocate \
        # RDMA dependencies
        libsysfs-dev \
        libibverbs-dev \
        librdmacm-dev \
        libibumad-dev \
        # InfiniBand tools
        infiniband-diags \
        ibverbs-utils \
        rdma-core \
        # Network tools
        iproute2 \
        net-tools \
        iputils-ping \
        # PCI tools
        libpci-dev \
        pciutils \
        # SSH server
        openssh-server \
        openssh-client && \
    updatedb && \
    # Configure SSH
    mkdir /var/run/sshd && \
    echo "root:${ROOT_PASSWORD}" | chpasswd && \
    sed -i 's/#PermitRootLogin prohibit-password/PermitRootLogin yes/' /etc/ssh/sshd_config && \
    # Generate SSH key for root user
    mkdir -p /root/.ssh && \
    ssh-keygen -t rsa -f /root/.ssh/id_rsa -N "" && \
    cp /root/.ssh/id_rsa.pub /root/.ssh/authorized_keys && \
    chmod 600 /root/.ssh/authorized_keys

# Install gdr-copy
RUN mkdir /workspace/gdrcopy && cd /workspace/gdrcopy && \
    wget https://github.com/NVIDIA/gdrcopy/archive/refs/tags/v2.5.zip && \
    unzip -q v2.5.zip && \
    cd gdrcopy-2.5/packages && \
    CUDA=/usr/local/cuda ./build-deb-packages.sh && \
    dpkg -i *.deb
    # && \
    # rm -rf /workspace/gdrcopy/v2.5.zip

# Install UCX
RUN mkdir -p /workspace/ucx && cd /workspace/ucx && \
    git clone https://github.com/openucx/ucx.git && \
```

```
    cd ucx && \
    ./autogen.sh && \
    mkdir build && \
    cd build && \
    ../contrib/configure-release \
        --prefix=/workspace/ucx/ucx_install \
        --with-cuda=/usr/local/cuda \
        --with-verbs \
        --enable-debug && \
    make -j$(nproc) && \
    make install

# Install OpenMPI
RUN mkdir /workspace/ompi && cd /workspace/ompi && \
    wget https://download.open-mpi.org/release/open-mpi/v4.1/openmpi-${OMPI_VERSION}.tar.gz && \
    tar zxf openmpi-${OMPI_VERSION}.tar.gz && \
    cd openmpi-${OMPI_VERSION} && \
    ./configure --prefix=/workspace/ompi/ompi_install \
        --with-cuda=/usr/local/cuda \
        --with-cuda-libdir=/usr/local/cuda/targets/x86_64-linux/lib/stubs \
        --with-verbs \
        --with-ucx=/workspace/ucx/ucx_install \
        --enable-debug && \
    make -j$(nproc) && \
    make install

# Install CUDA perftest
RUN wget https://github.com/linux-rdma/perftest/releases/download/25.01.0-0.80/perftest-25.01.0-
0.80.g6730e97.tar.gz && \
    tar zxf perftest-25.01.0-0.80.g6730e97.tar.gz && \
    cd perftest-25.01.0/ && \
    ./autogen.sh && \
    ./configure CUDA_H_PATH=/usr/local/cuda/include/cuda.h && \
    make -j$(nproc) && \
    rm -f /workspace/perftest-25.01.0-0.80.g6730e97.tar.gz

# Install NCCL tests
RUN git clone https://github.com/NVIDIA/nccl-tests.git && \
    cd nccl-tests && \
    make MPI=1 NAME_SUFFIX=_mpi MPI_HOME=/workspace/ompi/ompi_install -j$(nproc) && \
    # Create symbolic links for executables only (excluding .o files)
    find /workspace/nccl-tests/build -type f -executable -exec ln -s {} /usr/local/bin/ \;

# Expose SSH port
EXPOSE ${SSH_PORT}

# Start SSH daemon
CMD ["/usr/sbin/sshd", "-D"]
```

StatefulSet for distributed NCCL pods

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: mpi-nccl-cluster
spec:
  serviceName: mpi-nccl-svc
  replicas: 2  # 2 servers with 8x GPU + 8x IB
  selector:
    matchLabels:
      app: mpi-nccl
  template:
    metadata:
      labels:
        app: mpi-nccl
    spec:
      dnsPolicy: ClusterFirstWithHostNet
      hostNetwork: true  # Passthru host interface to pod
      hostPID: true      # Enable host PID namespace for better networking
      hostIPC: true      # Enable host IPC namespace
      containers:
      - name: mpi-node
        image: {Your_harbor_repo}/cuda-nccl-mpi:latest
        command: ["/bin/bash", "-c", "sleep infinity"]
        securityContext:
          privileged: true  # Required for RDMA and GPU access
        volumeMounts:
        - name: nvidia-driver
          mountPath: /run/nvidia/driver
          mountPropagation: HostToContainer
        - name: rdma-devices
          mountPath: /dev/infiniband
        - name: gdr-device
          mountPath: /dev/gdrdrv
        - name: ib-tools
          mountPath: /usr/sbin/ibstat
          subPath: ibstat
        - name: ib-tools
          mountPath: /usr/sbin/ibnetdiscover
          subPath: ibnetdiscover
        resources:
          limits:
            nvidia.com/gpu: 8
            rdma/rdma_shared_devices_a: 1
            rdma/rdma_shared_devices_b: 1
            rdma/rdma_shared_devices_c: 1
            rdma/rdma_shared_devices_d: 1
            rdma/rdma_shared_devices_e: 1
            rdma/rdma_shared_devices_f: 1
            rdma/rdma_shared_devices_g: 1
```

```
            rdma/rdma_shared_devices_h: 1
        ports:
        - containerPort: 2222  # SSH ports
      volumes:
      - name: nvidia-driver
        hostPath:
          path: /run/nvidia/driver
      - name: rdma-devices
        hostPath:
          path: /dev/infiniband
      - name: gdr-device
        hostPath:
          path: /run/nvidia/driver/dev/gdrdrv
      - name: ib-tools
        hostPath:
          path: /usr/sbin
      nodeSelector:
        nvidia.com/gpu.present: "true"
```

Deploy the statefulsets

```
k apply
k get po -o wide

NAME                  READY   STATUS    RESTARTS   AGE   IP                NODE
NOMINATED NODE   READINESS GATES
mpi-nccl-cluster-0   1/1     Running   0          11m   192.168.145.94    deepseek-test-cluster-gpu-x9d2k-82mqg-
xh2bq   <none>           <none>
mpi-nccl-cluster-1   1/1     Running   0          11m   192.168.146.119   deepseek-test-cluster-gpu-x9d2k-82mqg-
kh5jm   <none>           <none>

# Generate ssh-key
ssh-keygen -t rsa
# Set password on 2 pods
passwd
# Allow root login, modify /etc/ssh/sshd_config
echo "PermitRootLogin yes" >> /etc/ssh/sshd_config

# Startup sshd in each pod
nohup /usr/sbin/sshd -D > /dev/null 2>&1  &
# Copy ssh key to all Pods including itself
ssh-copy-id -i ~/.ssh/id_rsa.pub root@192.168.146.133
ssh-copy-id -i ~/.ssh/id_rsa.pub root@192.168.145.109

# Get IP from the two pods
kubectl get po -o wide | tail -n +2 | awk '{print $6, $1}'

echo "192.168.145.111 mpi-nccl-cluster-0
192.168.146.135 mpi-nccl-cluster-1" | tee -a /etc/hosts

echo "192.168.145.111 mpi-nccl-cluster-0
```

```
192.168.146.135 mpi-nccl-cluster-1" | tee -a ~/.ssh/known_hosts

# Test hostname in each pod
cat /etc/hosts

ping -c 3 mpi-nccl-cluster-0
ping -c 3 mpi-nccl-cluster-1

# Test ssh login functionality on both pods
ssh mpi-nccl-cluster-0 # Then exit
ssh mpi-nccl-cluster-1 # Then exit
```

Next, we conduct NCCL-all-reduce test on single pods and on 2 pods.

```
# Interactively
k exec -it mpi-nccl-cluster-0 -- bash
k exec -it mpi-nccl-cluster-1 -- bash

# 1. Single-node NCCL performance test
# nThread 1 nGpus 8 minBytes 8 maxBytes 17179869184 step: 2(factor) warmup iters: 5 iters: 20 agg iters: 1
validation: 1 graph: 0
#
# Using devices
#  Rank  0 Group  0 Pid  78123 on       pt01 device  0 [0000:04:00] NVIDIA H100 80GB HBM3
#  Rank  1 Group  0 Pid  78123 on       pt01 device  1 [0000:06:00] NVIDIA H100 80GB HBM3
#  Rank  2 Group  0 Pid  78123 on       pt01 device  2 [0000:0e:00] NVIDIA H100 80GB HBM3
#  Rank  3 Group  0 Pid  78123 on       pt01 device  3 [0000:10:00] NVIDIA H100 80GB HBM3
#  Rank  4 Group  0 Pid  78123 on       pt01 device  4 [0000:17:00] NVIDIA H100 80GB HBM3
#  Rank  5 Group  0 Pid  78123 on       pt01 device  5 [0000:19:00] NVIDIA H100 80GB HBM3
#  Rank  6 Group  0 Pid  78123 on       pt01 device  6 [0000:21:00] NVIDIA H100 80GB HBM3
#  Rank  7 Group  0 Pid  78123 on       pt01 device  7 [0000:23:00] NVIDIA H100 80GB HBM3
#
#                                                              out-of-place                       in-place
#       size         count      type   redop    root     time   algbw   busbw #wrong     time   algbw   busbw #wrong
#        (B)    (elements)                               (us)  (GB/s)  (GB/s)            (us)  (GB/s)  (GB/s)
           8             2     float     sum      -1    41.81    0.00    0.00      0    41.62    0.00    0.00      0
          16             4     float     sum      -1    40.53    0.00    0.00      0    40.90    0.00    0.00      0
          32             8     float     sum      -1    41.85    0.00    0.00      0    41.31    0.00    0.00      0
          64            16     float     sum      -1    42.05    0.00    0.00      0    41.42    0.00    0.00      0
         128            32     float     sum      -1    41.16    0.00    0.01      0    41.57    0.00    0.01      0
         256            64     float     sum      -1    41.38    0.01    0.01      0    41.64    0.01    0.01      0
         512           128     float     sum      -1    41.34    0.01    0.02      0    41.92    0.01    0.02      0
        1024           256     float     sum      -1    43.00    0.02    0.04      0    41.31    0.02    0.04      0
        2048           512     float     sum      -1    41.89    0.05    0.09      0    41.48    0.05    0.09      0
        4096          1024     float     sum      -1    41.46    0.10    0.17      0    41.36    0.10    0.17      0
        8192          2048     float     sum      -1    41.19    0.20    0.35      0    41.57    0.20    0.34      0
       16384          4096     float     sum      -1    41.77    0.39    0.69      0    41.21    0.40    0.70      0
       32768          8192     float     sum      -1    41.60    0.79    1.38      0    42.23    0.78    1.36      0
       65536         16384     float     sum      -1    41.48    1.58    2.76      0    42.29    1.55    2.71      0
      131072         32768     float     sum      -1    42.89    3.06    5.35      0    43.49    3.01    5.27      0
      262144         65536     float     sum      -1    43.70    6.00   10.50      0    44.90    5.84   10.22      0
      524288        131072     float     sum      -1    47.07   11.14   19.49      0    46.99   11.16   19.53      0
```

```
     1048576        262144     float     sum      -1    49.65   21.12   36.96      0    47.89   21.90   38.32      0
     2097152        524288     float     sum      -1    46.48   45.12   78.96      0    46.91   44.71   78.24      0
     4194304       1048576     float     sum      -1    56.45   74.31  130.04      0    55.39   75.72  132.51      0
     8388608       2097152     float     sum      -1    85.45   98.17  171.79      0    84.47   99.31  173.79      0
    16777216       4194304     float     sum      -1    125.8  133.39  233.44      0    125.2  134.05  234.58      0
    33554432       8388608     float     sum      -1    200.9  167.05  292.34      0    201.7  166.35  291.10      0
    67108864      16777216     float     sum      -1    328.3  204.40  357.70      0    328.9  204.05  357.08      0
   134217728      33554432     float     sum      -1    589.5  227.68  398.44      0    590.4  227.33  397.82      0
   268435456      67108864     float     sum      -1   1115.4  240.67  421.17      0   1115.6  240.63  421.10      0
   536870912     134217728     float     sum      -1   2153.9  249.26  436.20      0   2152.8  249.38  436.41      0
  1073741824     268435456     float     sum      -1   4014.0  267.50  468.12      0   4015.6  267.40  467.94      0
  2147483648     536870912     float     sum      -1   7918.9  271.18  474.57      0   7931.8  270.74  473.80      0
  4294967296    1073741824     float     sum      -1    15763  272.48  476.84      0    15779  272.20  476.35      0
  8589934592    2147483648     float     sum      -1    31325  274.22  479.88      0    31288  274.54  480.45      0
 17179869184    4294967296     float     sum      -1    62235  276.05  483.08      0    62311  275.71  482.50      0

# Out of bounds values : 0 OK
# Avg bus bandwidth    : 155.669
#
Finished command at 2025-05-13 02:52:51
Elapsed time: 39 seconds


# 2-node NCCL performance test
root@mpi-nccl-cluster-1:/workspace#
mpirun -np 16 \
       -H mpi-nccl-cluster-0:8,mpi-nccl-cluster-1:8 \
        -x NCCL_TOPO_FILE=/workspace/vm_topo_8h100_8ib_mod.xml \
       --allow-run-as-root \
       -x NCCL_DEBUG=VERSION \
        --mca pml ucx --mca btl_openib_warn_no_device_params_found 0 --mca btl ^openib \
        --prefix /workspace/ompi/ompi_install \
        -x NCCL_SOCKET_IFNAME=eth0 \
        -x NCCL_NET_GDR_LEVEL=1 \
        -x NCCL_P2P_LEVEL=NVL \
        -x NCCL_IB_DISABLE=0 \
        -x NCCL_IB_HCA=mlx5_0:1,mlx5_1:1,mlx5_2:1,mlx5_3:1,mlx5_4:1,mlx5_5:1,mlx5_6:1,mlx5_7:1 \
        -x NCCL_IB_GID_INDEX=0 \
       /workspace/nccl-tests/build/all_reduce_perf_mpi -b 8 -e 16G -f 2 -g 1
# nThread 1 nGpus 1 minBytes 8 maxBytes 17179869184 step: 2(factor) warmup iters: 5 iters: 20 agg iters: 1
validation: 1 graph: 0
#
# Using devices
#  Rank  0 Group  0 Pid    254 on mpi-nccl-cluster-1 device  0 [0000:04:00] NVIDIA H100 80GB HBM3
#  Rank  1 Group  0 Pid    255 on mpi-nccl-cluster-1 device  1 [0000:04:02] NVIDIA H100 80GB HBM3
#  Rank  2 Group  0 Pid    256 on mpi-nccl-cluster-1 device  2 [0000:04:04] NVIDIA H100 80GB HBM3
#  Rank  3 Group  0 Pid    257 on mpi-nccl-cluster-1 device  3 [0000:04:06] NVIDIA H100 80GB HBM3
#  Rank  4 Group  0 Pid    258 on mpi-nccl-cluster-1 device  4 [0000:04:08] NVIDIA H100 80GB HBM3
#  Rank  5 Group  0 Pid    259 on mpi-nccl-cluster-1 device  5 [0000:04:0a] NVIDIA H100 80GB HBM3
#  Rank  6 Group  0 Pid    261 on mpi-nccl-cluster-1 device  6 [0000:04:0c] NVIDIA H100 80GB HBM3
#  Rank  7 Group  0 Pid    263 on mpi-nccl-cluster-1 device  7 [0000:04:0e] NVIDIA H100 80GB HBM3
#  Rank  8 Group  0 Pid    273 on mpi-nccl-cluster-0 device  0 [0000:04:00] NVIDIA H100 80GB HBM3
```

```
#  Rank  9 Group  0 Pid    274 on mpi-nccl-cluster-0 device  1 [0000:04:02] NVIDIA H100 80GB HBM3
#  Rank 10 Group  0 Pid    275 on mpi-nccl-cluster-0 device  2 [0000:04:04] NVIDIA H100 80GB HBM3
#  Rank 11 Group  0 Pid    276 on mpi-nccl-cluster-0 device  3 [0000:04:06] NVIDIA H100 80GB HBM3
#  Rank 12 Group  0 Pid    277 on mpi-nccl-cluster-0 device  4 [0000:04:08] NVIDIA H100 80GB HBM3
#  Rank 13 Group  0 Pid    278 on mpi-nccl-cluster-0 device  5 [0000:04:0a] NVIDIA H100 80GB HBM3
#  Rank 14 Group  0 Pid    280 on mpi-nccl-cluster-0 device  6 [0000:04:0c] NVIDIA H100 80GB HBM3
#  Rank 15 Group  0 Pid    282 on mpi-nccl-cluster-0 device  7 [0000:04:0e] NVIDIA H100 80GB HBM3
NCCL version 2.25.1+cuda12.8
#
#                                                              out-of-place                       in-place
#       size         count      type   redop    root     time   algbw   busbw #wrong     time   algbw   busbw #wrong
#        (B)    (elements)                               (us)  (GB/s)  (GB/s)            (us)  (GB/s)  (GB/s)
           8             2     float     sum      -1    71.30    0.00    0.00      0    25.09    0.00    0.00      0
          16             4     float     sum      -1    25.19    0.00    0.00      0    25.20    0.00    0.00      0
          32             8     float     sum      -1    25.55    0.00    0.00      0    25.57    0.00    0.00      0
          64            16     float     sum      -1    26.87    0.00    0.00      0    25.72    0.00    0.00      0
         128            32     float     sum      -1    26.37    0.00    0.01      0    26.13    0.00    0.01      0
         256            64     float     sum      -1    77.68    0.00    0.01      0    26.62    0.01    0.02      0
         512           128     float     sum      -1    40.41    0.01    0.02      0    27.04    0.02    0.04      0
        1024           256     float     sum      -1    30.07    0.03    0.06      0    27.47    0.04    0.07      0
        2048           512     float     sum      -1    28.83    0.07    0.13      0    28.72    0.07    0.13      0
        4096          1024     float     sum      -1    30.86    0.13    0.25      0    30.51    0.13    0.25      0
        8192          2048     float     sum      -1    36.59    0.22    0.42      0    34.73    0.24    0.44      0
       16384          4096     float     sum      -1    37.05    0.44    0.83      0    35.30    0.46    0.87      0
       32768          8192     float     sum      -1    37.90    0.86    1.62      0    36.78    0.89    1.67      0
       65536         16384     float     sum      -1    38.41    1.71    3.20      0    36.55    1.79    3.36      0
      131072         32768     float     sum      -1    39.08    3.35    6.29      0    37.12    3.53    6.62      0
      262144         65536     float     sum      -1    47.17    5.56   10.42      0    46.48    5.64   10.57      0
      524288        131072     float     sum      -1    81.93    6.40   12.00      0    70.76    7.41   13.89      0
     1048576        262144     float     sum      -1    73.31   14.30   26.82      0    73.24   14.32   26.84      0
     2097152        524288     float     sum      -1    78.37   26.76   50.17      0    77.47   27.07   50.76      0
     4194304       1048576     float     sum      -1    95.22   44.05   82.59      0    94.78   44.25   82.97      0
     8388608       2097152     float     sum      -1    133.7   62.74  117.63      0    132.2   63.45  118.97      0
    16777216       4194304     float     sum      -1    187.1   89.68  168.16      0    184.5   90.96  170.54      0
    33554432       8388608     float     sum      -1    269.0  124.75  233.90      0    267.5  125.43  235.17      0
    67108864      16777216     float     sum      -1    468.2  143.32  268.73      0    465.8  144.06  270.11      0
   134217728      33554432     float     sum      -1    744.8  180.21  337.89      0    740.5  181.24  339.83      0
   268435456      67108864     float     sum      -1   1289.0  208.25  390.47      0   1287.5  208.49  390.91      0
   536870912     134217728     float     sum      -1   2349.8  228.48  428.39      0   2357.6  227.72  426.97      0
  1073741824     268435456     float     sum      -1   4469.2  240.25  450.47      0   4482.3  239.55  449.16      0
  2147483648     536870912     float     sum      -1   8722.2  246.21  461.64      0   8722.1  246.21  461.65      0
  4294967296    1073741824     float     sum      -1    17233  249.23  467.31      0    17248  249.01  466.89      0
  8589934592    2147483648     float     sum      -1    34338  250.16  469.05      0    34352  250.06  468.85      0
 17179869184    4294967296     float     sum      -1    68599  250.44  469.58      0    68550  250.62  469.91      0
# Out of bounds values : 0 OK
# Avg bus bandwidth    : 139.462

# Cleanup
kubectl delete -f test-mpi-statefulsets.yaml
```

For more information about whether you achieved your expected NCCL performance more 4 or more nodes, refer to https://github.com/NVIDIA/nccl-tests/issues/309.

You should verify the performance from the following key points:

- **NCCL version**: 2.25.1+CUDA12.8
- **Devices**: 8 GPUs per node, properly mapped by customized NCCL topology file (`vm_topo_8h100_8ib_mod.xml`) as shown Appendix I.
- **Expected peak performance (2 nodes)**: ~470 GB/s with the TREE protocol (idealized case). Two-node NCCL results should be interpreted with caution, as they often appear artificially high—this is essentially a best-case scenario for the TREE protocol. In practice, forcing the use of the Ring protocol for 2-node testing provides a more realistic picture of distributed performance, and aligns better with what can be expected when scaling beyond two nodes.
- **Realistic distributed performance**: For multi-node scaling (especially with 4+ nodes), `NCCL_ALGO=Ring` provides more consistent results. Additional tuning — such as `NCCL_MIN_CTAS=24` or `NCCL_IB_QPS_PER_CONNECTION=2` — can help benchmarks approach the InfiniBand line-rate limit of ~392 GB/s. can push benchmarks closer to the InfiniBand line-rate limit of ~392 GB/s. However, these optimizations also increase GPU compute overhead, which can reduce actual application performance. We observed this trade-off during real workload testing.

I. VM customized NCCL topology file

The following NCCL topology file is suited for VMs that have 8x H100 and 8x CX-7 IB HCAs.

vm_topo_8h100_8ib_mod.xml

```
<system version="1">
  <cpu host_hash="0xabd045be9cccc099" numaid="0" affinity="5555,55555555,55555555,55555555,55555555,55555555,55555555"
arch="x86_64" vendor="GenuineIntel" familyid="6" modelid="143">
    <pci busid="0000:16:00.0" class="0x060400" vendor="0x1000" device="0xc030" subsystem_vendor="0x1028"
subsystem_device="0x2330" link_speed="32.0 GT/s PCIe" link_width="16">
      <pci busid="0000:04:00.0" class="0x030200" vendor="0x10de" device="0x2330" subsystem_vendor="0x10de"
subsystem_device="0x16c1" link_speed="32.0 GT/s PCIe" link_width="16">
        <gpu dev="0" sm="90" rank="0" gdr="1">
          <nvlink target="0000:04:12.0" count="5" tclass="0x068000"/>
          <nvlink target="0000:04:11.0" count="5" tclass="0x068000"/>
          <nvlink target="0000:04:10.0" count="4" tclass="0x068000"/>
          <nvlink target="0000:04:13.0" count="4" tclass="0x068000"/>
        </gpu>
      </pci>
      <pci busid="0000:04:01.0" class="0x020000" vendor="0x15b3" device="0x1021" subsystem_vendor="0x15b3"
subsystem_device="0x0041" link_speed="32.0 GT/s PCIe" link_width="16">
        <nic>
          <net name="mlx5_0" dev="0" latency="0" speed="400000" port="1" guid="0x5c24fb0003ae6d94" maxconn="131072"
gdr="1"/>
        </nic>
      </pci>
    </pci>
    <pci busid="0000:38:00.0" class="0x060400" vendor="0x1000" device="0xc030" subsystem_vendor="0x1028"
subsystem_device="0x2330" link_speed="32.0 GT/s PCIe" link_width="16">
      <pci busid="000004:03.0" class="0x020000" vendor="0x15b3" device="0x1021" subsystem_vendor="0x15b3"
subsystem_device="0x0041" link_speed="32.0 GT/s PCIe" link_width="16">
        <nic>
          <net name="mlx5_1" dev="1" latency="0" speed="400000" port="1" guid="0x1c24fb0003ae6d94" maxconn="131072"
gdr="1"/>
        </nic>
      </pci>
      <pci busid="0000:04:02.0" class="0x030200" vendor="0x10de" device="0x2330" subsystem_vendor="0x10de"
subsystem_device="0x16c1" link_speed="32.0 GT/s PCIe" link_width="16">
        <gpu dev="1" sm="90" rank="1" gdr="1">
          <nvlink target="0000:04:12.0" count="5" tclass="0x068000"/>
          <nvlink target="0000:04:11.0" count="5" tclass="0x068000"/>
          <nvlink target="0000:04:10.0" count="4" tclass="0x068000"/>
          <nvlink target="0000:04:13.0" count="4" tclass="0x068000"/>
        </gpu>
      </pci>
    </pci>
    <pci busid="0000:49:00.0" class="0x060400" vendor="0x1000" device="0xc030" subsystem_vendor="0x1028"
subsystem_device="0x2331" link_speed="32.0 GT/s PCIe" link_width="16">
      <pci busid="0000:04:05.0" class="0x020000" vendor="0x15b3" device="0x1021" subsystem_vendor="0x15b3"
subsystem_device="0x0041" link_speed="32.0 GT/s PCIe" link_width="16">
        <nic>
          <net name="mlx5_2" dev="2" latency="0" speed="400000" port="1" guid="0x820fb0003ae6d94" maxconn="131072"
gdr="1"/>
        </nic>
      </pci>
```

```
      <pci busid="0000:04:04.0" class="0x030200" vendor="0x10de" device="0x2330" subsystem_vendor="0x10de"
subsystem_device="0x16c1" link_speed="32.0 GT/s PCIe" link_width="16">
        <gpu dev="2" sm="90" rank="2" gdr="1">
          <nvlink target="0000:04:12.0" count="5" tclass="0x068000"/>
          <nvlink target="0000:04:11.0" count="5" tclass="0x068000"/>
          <nvlink target="0000:04:10.0" count="4" tclass="0x068000"/>
          <nvlink target="0000:04:13.0" count="4" tclass="0x068000"/>
        </gpu>
      </pci>
    </pci>
    <pci busid="0000:5a:00.0" class="0x060400" vendor="0x1000" device="0xc030" subsystem_vendor="0x1028"
subsystem_device="0x2331" link_speed="32.0 GT/s PCIe" link_width="16">
      <pci busid="0000:04:07.0" class="0x020000" vendor="0x15b3" device="0x1021" subsystem_vendor="0x15b3"
subsystem_device="0x0041" link_speed="32.0 GT/s PCIe" link_width="16">
        <nic>
          <net name="mlx5_3" dev="3" latency="0" speed="400000" port="1" guid="0x3424fb0003ae6d94" maxconn="131072"
gdr="1"/>
        </nic>
      </pci>
      <pci busid="0000:04:06.0" class="0x030200" vendor="0x10de" device="0x2330" subsystem_vendor="0x10de"
subsystem_device="0x16c1" link_speed="32.0 GT/s PCIe" link_width="16">
        <gpu dev="3" sm="90" rank="3" gdr="1">
          <nvlink target="0000:04:12.0" count="5" tclass="0x068000"/>
          <nvlink target="0000:04:11.0" count="5" tclass="0x068000"/>
          <nvlink target="0000:04:10.0" count="4" tclass="0x068000"/>
          <nvlink target="0000:04:13.0" count="4" tclass="0x068000"/>
        </gpu>
      </pci>
    </pci>
  </cpu>
  <cpu host_hash="0xabd045be9cccc099" numaid="1" affinity="aaaa,aaaaaaaa,aaaaaaaa,aaaaaaaa,aaaaaaaa,aaaaaaaa,aaaaaaaa"
arch="x86_64" vendor="GenuineIntel" familyid="6" modelid="143">
    <pci busid="0000:98:00.0" class="0x060400" vendor="0x1000" device="0xc030" subsystem_vendor="0x1028"
subsystem_device="0x2333" link_speed="32.0 GT/s PCIe" link_width="16">
      <pci busid="0000:04:09.0" class="0x020000" vendor="0x15b3" device="0x1021" subsystem_vendor="0x15b3"
subsystem_device="0x0041" link_speed="32.0 GT/s PCIe" link_width="16">
        <nic>
          <net name="mlx5_4" dev="4" latency="0" speed="400000" port="1" guid="0xac22fb0003ae6d94" maxconn="131072"
gdr="1"/>
        </nic>
      </pci>
      <pci busid="0000:04:08.0" class="0x030200" vendor="0x10de" device="0x2330" subsystem_vendor="0x10de"
subsystem_device="0x16c1" link_speed="32.0 GT/s PCIe" link_width="16">
        <gpu dev="4" sm="90" rank="4" gdr="1">
          <nvlink target="0000:04:12.0" count="5" tclass="0x068000"/>
          <nvlink target="0000:04:11.0" count="5" tclass="0x068000"/>
          <nvlink target="0000:04:10.0" count="4" tclass="0x068000"/>
          <nvlink target="0000:04:13.0" count="4" tclass="0x068000"/>
        </gpu>
      </pci>
    </pci>
    <pci busid="0000:b8:00.0" class="0x060400" vendor="0x1000" device="0xc030" subsystem_vendor="0x1028"
subsystem_device="0x2333" link_speed="32.0 GT/s PCIe" link_width="16">
      <pci busid="0000:04:0b.0" class="0x020000" vendor="0x15b3" device="0x1021" subsystem_vendor="0x15b3"
subsystem_device="0x0041" link_speed="32.0 GT/s PCIe" link_width="16">
        <nic>
```

```
          <net name="mlx5_5" dev="5" latency="0" speed="400000" port="1" guid="0x5420fb0003ae6d94" maxconn="131072"
gdr="1"/>
        </nic>
      </pci>
      <pci busid="0000:04:0a.0" class="0x030200" vendor="0x10de" device="0x2330" subsystem_vendor="0x10de"
subsystem_device="0x16c1" link_speed="32.0 GT/s PCIe" link_width="16">
        <gpu dev="5" sm="90" rank="5" gdr="1">
          <nvlink target="0000:04:12.0" count="5" tclass="0x068000"/>
          <nvlink target="0000:04:11.0" count="5" tclass="0x068000"/>
          <nvlink target="0000:04:10.0" count="4" tclass="0x068000"/>
          <nvlink target="0000:04:13.0" count="4" tclass="0x068000"/>
        </gpu>
      </pci>
    </pci>
    <pci busid="0000:c8:00.0" class="0x060400" vendor="0x1000" device="0xc030" subsystem_vendor="0x1028"
subsystem_device="0x2332" link_speed="32.0 GT/s PCIe" link_width="16">
      <pci busid="0000:04:0d.0" class="0x020000" vendor="0x15b3" device="0x1021" subsystem_vendor="0x15b3"
subsystem_device="0x0041" link_speed="32.0 GT/s PCIe" link_width="16">
        <nic>
          <net name="mlx5_6" dev="6" latency="0" speed="400000" port="1" guid="0x6024fb0003ae6d94" maxconn="131072"
gdr="1"/>
        </nic>
      </pci>
      <pci busid="0000:04:0c.0" class="0x030200" vendor="0x10de" device="0x2330" subsystem_vendor="0x10de"
subsystem_device="0x16c1" link_speed="32.0 GT/s PCIe" link_width="16">
        <gpu dev="6" sm="90" rank="6" gdr="1">
          <nvlink target="0000:04:12.0" count="5" tclass="0x068000"/>
          <nvlink target="0000:04:11.0" count="5" tclass="0x068000"/>
          <nvlink target="0000:04:10.0" count="4" tclass="0x068000"/>
          <nvlink target="0000:04:13.0" count="4" tclass="0x068000"/>
        </gpu>
      </pci>
    </pci>
    <pci busid="0000:d8:00.0" class="0x060400" vendor="0x1000" device="0xc030" subsystem_vendor="0x1028"
subsystem_device="0x2332" link_speed="32.0 GT/s PCIe" link_width="16">
      <pci busid="0000:04:0f.0" class="0x020000" vendor="0x15b3" device="0x1021" subsystem_vendor="0x15b3"
subsystem_device="0x0041" link_speed="32.0 GT/s PCIe" link_width="16">
        <nic>
          <net name="mlx5_7" dev="7" latency="0" speed="400000" port="1" guid="0xbc22fb0003ae6d94" maxconn="131072"
gdr="1"/>
        </nic>
      </pci>
      <pci busid="0000:04:0e.0" class="0x030200" vendor="0x10de" device="0x2330" subsystem_vendor="0x10de"
subsystem_device="0x16c1" link_speed="32.0 GT/s PCIe" link_width="16">
        <gpu dev="7" sm="90" rank="7" gdr="1">
          <nvlink target="0000:04:12.0" count="5" tclass="0x068000"/>
          <nvlink target="0000:04:11.0" count="5" tclass="0x068000"/>
          <nvlink target="0000:04:10.0" count="4" tclass="0x068000"/>
          <nvlink target="0000:04:13.0" count="4" tclass="0x068000"/>
        </gpu>
      </pci>
    </pci>
  </cpu>
</system>
```

J. Terminology

This section provides terminology definitions for terms used in this document.

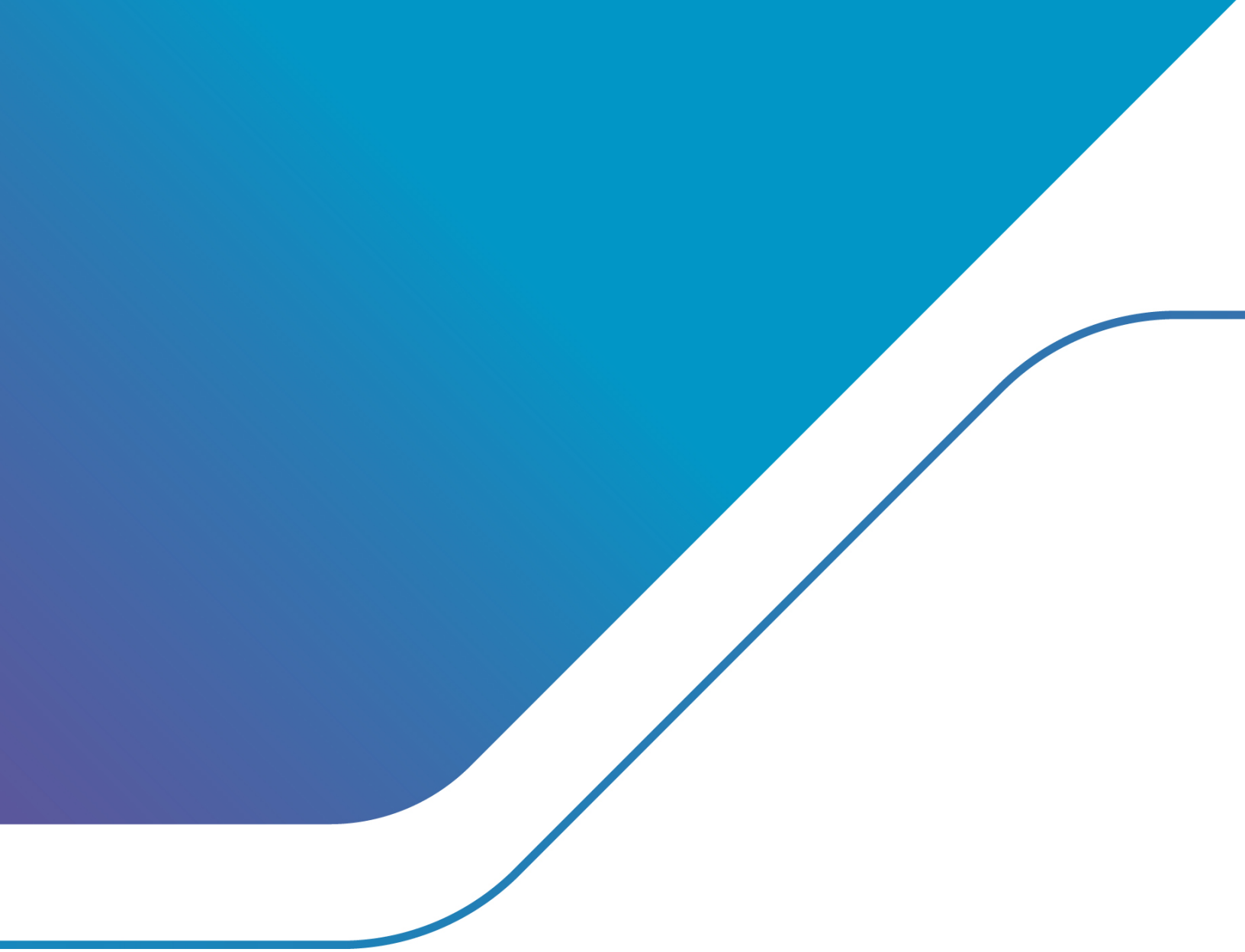
Term	Definition
ACS	PCIe Access Control Service
BTL	Byte Transfer Layer
CRD	Custom Resource Definition
DLVM	Deep Learning Virtual Machines
FLB	Foundation Load Balancer
HA	High Availability
IOMMU	Input Output Memory Management Unit
IB	InfiniBand
LCI	Local Consumption Interface
LCM	Life Cycle Management
LLM	Large Language Model
NGC	NVIDIA GPU Cloud
OOB	Out-Of-Band
RDMA	Remote Direct Memory Access
RoCE	RDMA over Converged Ethernet
SBDF	Segment–Bus–Device–Function
TKG Cluster	Tanzu Kubernetes Grid Cluster
VCF	VMware Cloud Foundation
vGPU	NVIDIA GRID Virtual GPU (C-Series)
VIB	vSphere Installation Bundles
VKS	vSphere Kubernetes Service
VM	Virtual Machine
VPC	Virtual Private Cloud
VCF WLD	VCF Workload Domain

About the author

Dr. Yuankun Fu is a performance engineer at Broadcom focusing on optimizing AI and HPC performance.

Acknowledgments

The author would like to thank **Ramesh Radhakrishnan**, **Yang Lu**, **Agustin Malanco**, and **Chris Wolf** from Broadcom's VMware Cloud Foundation division for their support in the development of this paper. Appreciation is also extended to **Frank Denneman**, **Justin Murray**, and **Chris Gully** for their thoughtful review and constructive feedback. A special thanks goes to **Julie Brodeur** for her careful editing and formatting contributions.



vmware® by Broadcom