Data Center / Cloud

Generative AI / LLMs

Robotics

Content Creation / Rendering

Data Science

Networking

Simulation / Modeling / Design

Conversational AI

Forums

Sign In

Searc

NVIDIA Doc ... king Solutions › RDG for DPF Zero Trust (

On Thi

- Scope
- Abbreviations and Acronyms
- Introduction
- References
- Solution Architecture
  - Key Components and Technologies
  - Solution Design
    - Solution Logical Design
    - Firewall Design
  - Software Stack Components
  - Bill of Materials
- Deployment and Configuration
  - Node and Switch Definitions
  - Wiring
    - Hypervisor Node
    - Bare Metal Worker Node
  - Fabric Configuration
    - Updating Cumulus Linux
    - Configuring the Cumulus Linux Switch
  - Host Configuration
    - Hypervisor Installation and Configuration
    - Prepare Infrastructure Servers
    - Provision Master VMs Using MaaS
- K8s Cluster Deployment and Configuration
  - Kubespray Deployment and Configuration
  - Deploying Cluster Using Kubespray Ansible Playbook
  - K8s Deployment Verification
- DPF Installation
  - Software Prerequisites and Required Variables
  - DPF Operator Installation
    - Create Storage Required by the DPF Operator
    - Verify the rshim service
    - Additional Dependencies
    - DPF Operator Deployment
  - DPF System Installation
  - (Optional) DPF system installation verification
  - Optional
- Authors

RDG for DPF Zero Trust (DPF-ZT)

Is this page helpful?

Created Sep 08, 2025

Updated Dec 29 2025 (v25.10 GA)

Scope

This Reference Deployment Guide (RDG) provides comprehensive instructions for deploying the NVIDIA DOCA Platform Framework (DPF) on high-performance, bare-metal infrastructure in **Zero-Trust mode**. It focuses on the setup and use of DPU-based services on **NVIDIA® BlueField®-3 DPUs** to deliver secure, isolated, and hardware-accelerated environments.

The guide is intended for experienced system administrators, systems engineers, and solution architects who build highly secure bare-metal environments using NVIDIA BlueField DPUs for acceleration, isolation, and infrastructure offload.

✓ Note

- This reference implementation, as the name implies, is a specific, opinionated deployment example designed to address the use case described above.
- Although other approaches may exist for implementing similar solutions, this document provides a detailed guide for this specific method.

Abbreviations and Acronyms

Term	Definition	Term	Definition
BFB	BlueField Bootstream	NGC	NVIDIA GPU Cloud
DOCA	Data Center Infrastructure-on-a-Chip Architecture	NFS	Network File System
DPF	DOCA Platform Framework	OOB	Out-of-Band
DPU	Data Processing Unit	PF	Physical Function

Term	Definition	Term	Definition
K8S	Kubernetes	RDG	Reference Deployment Guide
KVM	Kernel-based Virtual Machine	RDMA	Remote Direct Memory Access
MAAS	Metal as a Service	RoCE	RDMA over Converged Ethernet
MTU	Maximum Transmission Unit	ZT	Zero Trust

Introduction

The **NVIDIA BlueField-3 Data Processing Unit (DPU)** is a 400 Gb/s infrastructure compute platform designed for line-rate processing of software-defined networking, storage, and cybersecurity workloads. It combines powerful compute resources, high-speed networking, and advanced programmability to deliver **hardware-accelerated, software-defined solutions** for modern data centers.

NVIDIA DOCA unleashes the full potential of the BlueField platform by enabling rapid development of applications and services that **offload, accelerate, and isolate** data center workloads.

However, deploying and managing DPUs, especially at scale, presents operational challenges. Without a robust provisioning and orchestration system, tasks such as lifecycle management, service deployment, and network configuration for service function chaining (SFC) can quickly become **complex and error prone**. This is where the **DOCA Platform Framework (DPF)** comes into play.

DPF automates the full DPU lifecycle, and simplifies advanced network configurations. With DPF, services can be deployed seamlessly, allowing for **efficient offloading and intelligent routing** of traffic through the DPU data plane.

By leveraging DPF, users can **scale and automate DPU management** across Bare Metal, Virtual, and Kubernetes customer environments - **optimizing performance while simplifying operations**.

DPF supports multiple deployment models. This guide focuses on the **Zero Trust bare-metal deployment model**. In this scenario:

- The DPU is **managed** through its **Baseboard Management Controller (BMC)**
- All management traffic occurs over the **DPU's out-of-band (OOB)** network
- The host is considered as an untrusted entity towards the data center network. The DPU acts as a barrier between the host and the network.
- The host sees the **DPU** as a standard NIC, with no access to the internal DPU management plane **(Zero Trust Mode)**

This **Reference Deployment Guide (RDG)** provides a **step-by-step example** for installing DPF in Zero-Trust mode. It also includes **practical demonstrations of performance optimization**, validated using standard **RDMA and TCP workloads**.

As part of the reference implementation, **open-source components outside the scope of DPF** (e.g., MAAS, pfSense, Kubespray) are used to simulate a realistic customer deployment environment. The guide includes the full end-to-end deployment process, including:

- Infrastructure provisioning
- DPF deployment
- DPU provisioning (redfish)
- Service configuration and deployment
- Service chaining.

References

- **NVIDIA BlueField DPU**
- **NVIDIA DOCA**
- **NVIDIA DPF Release Notes**
- **NVIDIA DPF GitHub Repository**
- **NVIDIA DPF System Overview**
- **NVIDIA Ethernet Switching**
- **NVIDIA Cumulus Linux**
- **What is K8s?**
- **Kubespray**

Solution Architecture

Key Components and Technologies

- **NVIDIA BlueField® Data Processing Unit (DPU)**
  The NVIDIA® BlueField® data processing unit (DPU) ignites unprecedented innovation for modern data centers and supercomputing clusters. With its robust compute power and integrated software-defined hardware accelerators for networking, storage, and security, BlueField creates a secure and accelerated infrastructure for any workload in any environment, ushering in a new era of accelerated computing and AI.

- **NVIDIA DOCA Software Framework**
  NVIDIA DOCA™ unlocks the potential of the NVIDIA® BlueField® networking platform. By harnessing the power of BlueField DPUs and SuperNICs, DOCA enables the rapid creation of applications and services that offload, accelerate, and isolate data center workloads. It lets developers create software-defined, cloud-native, DPU- and SuperNIC-accelerated services with zero-trust protection, addressing the performance and security demands of modern data centers.

- **NVIDIA ConnectX SmartNICs**
  10/25/40/50/100/200 and 400G Ethernet Network Adapters
  The industry-leading NVIDIA® ConnectX® family of smart network interface cards (SmartNICs) offer advanced hardware offloads and accelerations.
  NVIDIA Ethernet adapters enable the highest ROI and lowest Total Cost of Ownership for hyperscale, public and private clouds, storage, machine learning, AI, big data, and telco platforms.

- **NVIDIA LinkX Cables**
  The NVIDIA® LinkX® product family of cables and transceivers provides the industry's most complete line of 10, 25, 40, 50, 100, 200, and 400GbE in Ethernet and 100, 200 and 400Gb/s InfiniBand products for Cloud, HPC, hyperscale, Enterprise, telco, storage and artificial intelligence, data center applications.

- **NVIDIA Spectrum Ethernet Switches**
  Flexible form-factors with 16 to 128 physical ports, supporting 1GbE through 400GbE speeds.
  Based on a ground-breaking silicon technology optimized for performance and scalability, NVIDIA Spectrum switches are ideal for building high-performance, cost-effective, and efficient Cloud Data Center Networks, Ethernet Storage Fabric, and Deep Learning Interconnects.
  NVIDIA combines the benefits of **NVIDIA Spectrum™** switches, based on an industry-leading application-specific integrated circuit (ASIC) technology, with a wide variety of modern network operating system choices, including **NVIDIA Cumulus® Linux** , **SONiC** and **NVIDIA Onyx®**.

- **NVIDIA Cumulus Linux**
  NVIDIA® Cumulus® Linux is the industry's most innovative open network operating system that allows you to automate, customize, and scale your data center network like no other.

- **Kubernetes**
  Kubernetes is an open-source container orchestration platform for deployment automation, scaling, and management of containerized applications.

- **Kubespray**

Kubespray is a composition of **Ansible** playbooks, inventory, provisioning tools, and domain knowledge for generic OS/Kubernetes clusters configuration management tasks and provides:

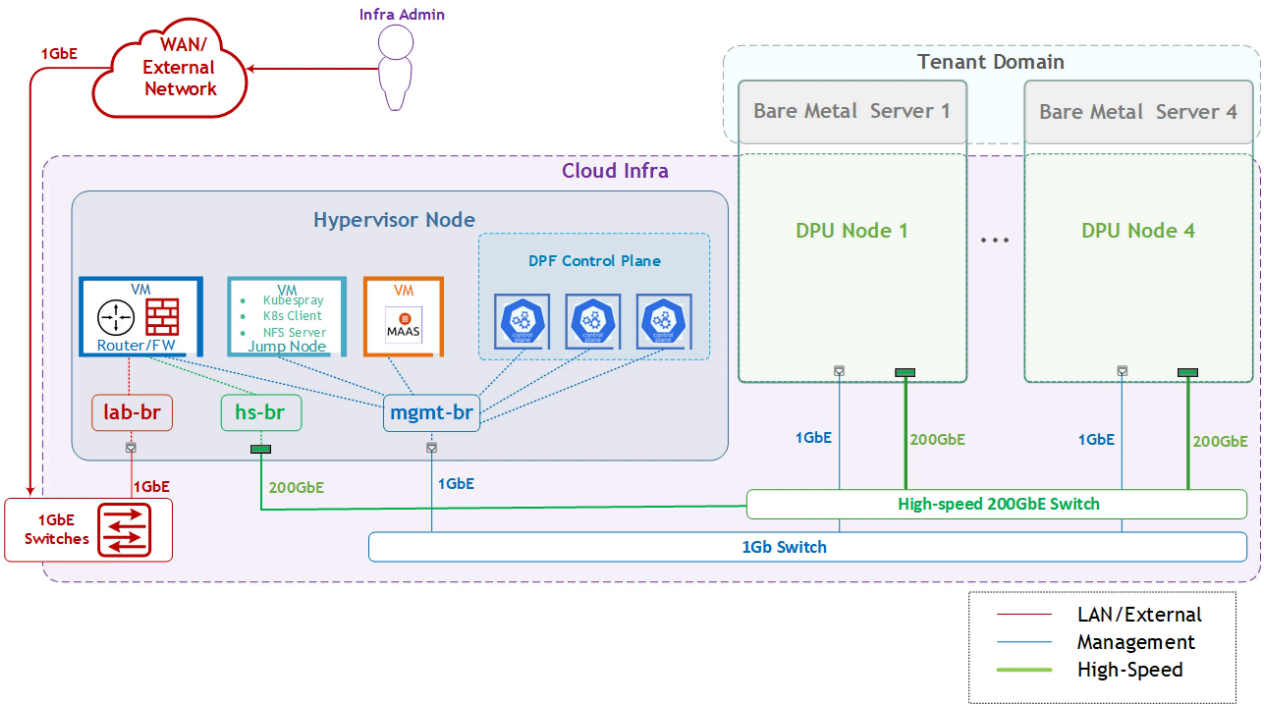
- A highly available cluster
- Composable attributes
- Support for most popular Linux distributions

Solution Design

Solution Logical Design

The logical design includes the following components:

- 1 x Hypervisor node (KVM-based) with ConnectX-7:
  - 1 x Firewall VM
  - 1 x Jump Node VM
  - 1 x MaaS VM
  - 3 x K8s Master VMs running all K8s management components
- 4 x Worker nodes (PCI Gen5), each with a 1 x BlueField-3 NIC
- Single High-Speed (HS) switch
- 1 Gb Host Management network



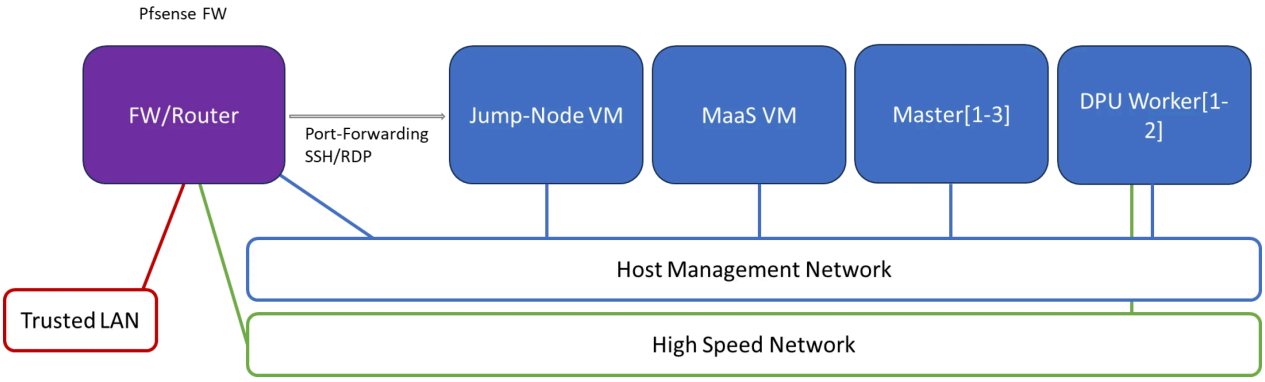
Firewall Design

The pfSense firewall in this solution serves a dual purpose:

- Firewall—provides an isolated environment for the DPF system, ensuring secure operations

- Router—enables Internet access for the management network

Port-forwarding rules for SSH and RDP are configured on the firewall to route traffic to the jump node's IP address in the host management network. From the jump node, administrators can manage and access various devices in the setup, as well as handle the deployment of the Kubernetes (K8s) cluster and DPF components.The following diagram illustrates the firewall design used in this solution:



Software Stack Components

Kubernetes	K8s 1.32.8
	NVIDIA DPF Operator v25.10.0
Host OS	Ubuntu 24.04 Server
	Kernel 6.8.0-90
BlueField-3 DPU	BFB: bf-bundle-3.2.1-34_25.11_ubuntu-24.04_64k_prod
	Firmware: 32.47.1088
Spectrum Ethernet Switch	NVIDIA Cumulus Linux v5.15.1

ⓘ Warning

Make sure to use the **exact same versions** for the software stack as described above.

Bill of Materials

#	Part	OPN	Qnty	Description
1	Hypervisor Server	x86 based server	1	x86 server system to run all virtualized instances CPU: 2 x Intel Xeon Platinum 8168 (24 cores @ 2.7GHz) with PCI Gen4x32 RAM: 384GB Storage: 1.6TB (NVMe) Network: ConnectX-7 dual-port adapter
2	Bare-Metal Server	x86 based server	4	x86 server system CPU: 2 x Intel(R) Xeon(R) Platinum 8380 (40 cores @ 2.3GHz) with PCI Gen4x32 RAM: 256GB Storage: 3.8TB (NVMe) Network: BlueField-3 P-Series
3	SN2201 Ethernet Switch 	MSN2201-CB2FC	1	Spectrum OOB Management Switch with 48 RJ45 1GbT + 4 QSFP28 100GbE Ports
4	MSN3700 Ethernet Switch 	MSN3700-VS2FC	1	Spectrum-2 Based 200GbE 1U Open Ethernet Switch 32 QSFP56 Ports (Used for High-speed Network)
5	Ethernet Direct Attach Copper Cables 	MCP1650-H	9	QSFP56 passive copper cable (200Gbps)
6	BlueField-3 DPU 	900-9D3B6-00CV-AA0	4	BlueField-3 B3220 P-Series DPU, dual-port 200GbE QSFP112, PCIe Gen 5.0 x16 16 Arm cores, 32GB memory ***The network adapter cards are used in item #2.***

Deployment and Configuration

Node and Switch Definitions

These are the definitions and parameters used for deploying the demonstrated fabric:

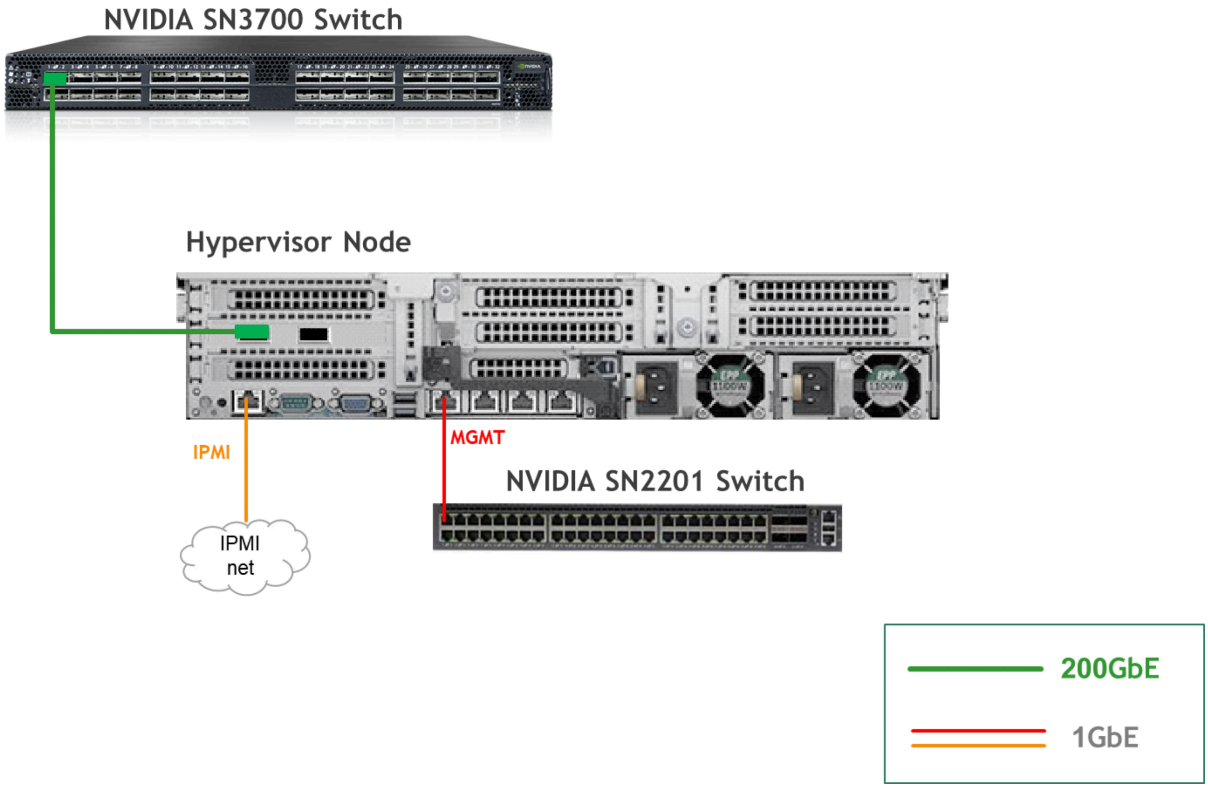
Switches Ports Usage		
Hostname	**Rack ID**	**Ports**
`mgmt-switch`	1	swp1-5
`hs-switch`	1	swp1-9

Expand

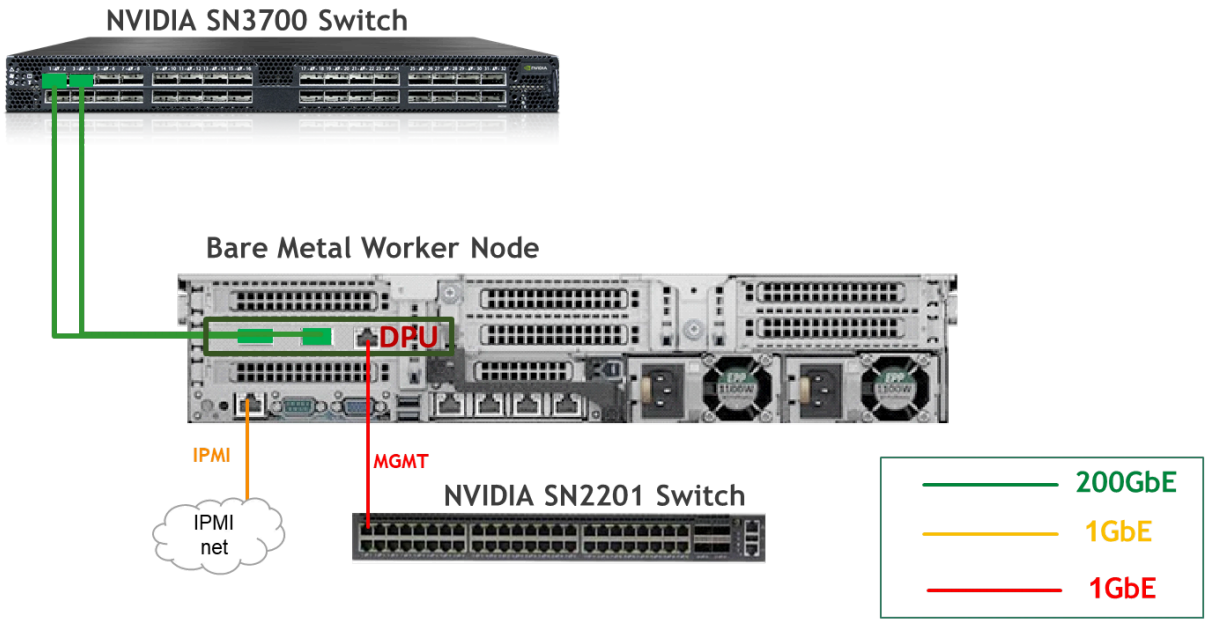
Hosts				
Rack	**Server Type**	**Server Name**	**Switch Port**	**IP and NICs**
Rack1	Hypervisor Node	`hypervisor`	**mgmt-switch:** `swp1` **hs-switch:** `swp1`	**lab-br (interfac** **eno1):** Trusted I **mgmt-br (inter** **eno2):** - **hs-br (interfac** **enp1s0):** -
Rack1	Firewall (Virtual)	`fw`	-	**WAN (lab-br):** T LAN IP **LAN (mgmt-br)** 10.0.110.254/2 **OPT1(hs-br):** 172.169.50.1/3(
Rack1	Jump Node (Virtual)	`jump`	-	**enp1s0:** 10.0.110.253/2
Rack1	MaaS (Virtual)	`maas`	-	**enp1s0:** 10.0.110.252/2
Rack1	Master Node (Virtual)	`master1`	-	**enp1s0:** 10.0.11
Rack1	Master Node (Virtual)	`master2`	-	**enp1s0:** 10.0.11
Rack1	Master Node (Virtual)	`master3`	-	**enp1s0:** 10.0.11

Wiring

Hypervisor Node



Bare Metal Worker Node



Fabric Configuration

Updating Cumulus Linux

As a best practice, make sure to use the latest released Cumulus Linux NOS version.

For information on how to upgrade Cumulus Linux, refer to the **Cumulus Linux User Guide**.

Configuring the Cumulus Linux Switch

The SN3700 switch (`hs-switch`), is configured as follows:

SN3700 Switch Console — Collapse Source

```
nv set bridge domain br_hs untagged 1
nv set interface swp1-5 bridge domain br_hs
nv set interface swp1-5 link state up
nv set interface swp1-5 type swp
nv config apply -y
nv config save
```

The SN2201 switch (`mgmt-switch`) is configured as follows:

SN2201 Switch Console — Collapse Source

```
nv set interface swp1-3 link state up
nv set interface swp1-3 type swp
nv set interface swp1-3 bridge domain br_default
nv set bridge domain br_default untagged 1
nv config apply -y
nv config save
```

Host Configuration

ⓘ Warning

Make sure that the BIOS settings on the worker node servers have SR-IOV enabled and that the servers are tuned for maximum performance.

All worker nodes must have the same PCIe placement for the BlueField-3 NIC and must display the same interface name.

Make sure that you have DPU BMC and OOB MAC addresses.

Hypervisor Installation and Configuration

The hypervisor used in this Reference Deployment Guide (RDG) is based on Ubuntu 24.04 with KVM.

While this document does not detail the KVM installation process, it is important to note that the setup requires the following ISOs to deploy the Firewall, Jump, and MaaS virtual machines (VMs):

- Ubuntu 24.04
- pfSense-CE-2.7.2

To implement the solution, three Linux bridges must be created on the hypervisor:

✓ Note

Ensure a DHCP record is configured for the `lab-br` bridge interface in your trusted LAN to assign it an IP address.

- `lab-br` – connects the Firewall VM to the trusted LAN.
- `mgmt-br` – Connects the various VMs to the host management network.
- hs-br – Connects the Firewall VM to the high-speed network.

Additionally, an MTU of 9000 must be configured on the management and high-speed bridges (mgmt-br and hs-br) as well as their uplink interfaces to ensure optimal performance.

Hypervisor netplan configuration — Collapse Source

```
network:
    ethernets:
        eno1:
            dhcp4: false
        eno2:
            dhcp4: false
            mtu: 9000
        ens2f0np0:
            dhcp4: false
            mtu: 9000
```

```
    bridges:
      lab-br:
         interfaces: [eno1]
         dhcp4: true
      mgmt-br:
         interfaces: [eno2]
         dhcp4: false
         mtu: 9000
      hs-br:
         interfaces: [ens2f0np0]
         dhcp4: false
         mtu: 9000
    version: 2
```

Apply the configuration:

Hypervisor Console — Collapse Source

```
$ sudo netplan apply
```

Prepare Infrastructure Servers

Firewall VM - pfSense Installation and Interface Configuration

Download the pfSense CE (Community Edition) ISO to your hypervisor and proceed with the software installation.

Suggested spec:

- vCPU: 2
- RAM: 2GB
- Storage: 10GB
- Network interfaces
  - Bridge device connected to lab-br
  - Bridge device connected to mgmt-br
  - Bridge device connected to hs-br

The Firewall VM must be connected to all three Linux bridges on the hypervisor. Before beginning the installation, ensure that three virtual network interfaces of type **"Bridge device"** are configured. Each interface should be connected to a different bridge (lab-br, mgmt-br, and hs-br) as illustrated in the diagram below.

After completing the installation, the setup wizard displays a menu with several options, such as "Assign Interfaces" and "Reboot System." During this phase, you must configure the network interfaces for the Firewall VM.

1. Select **Option 2: "Set interface(s) IP address"** and configure the interfaces as follows:
   - **WAN (lab-br)** – Trusted LAN IP (Static/DHCP)
   - **LAN (mgmt-br)** – Static IP 10.0.110.254/24
   - **OPT1 (hs-br)** – Static IP 10.0.123.254/22
2. Once the interface configuration is complete, use a web browser within the host management network to access the Firewall web interface and finalize the configuration.

Next, proceed with installing the **Jump VM**. This VM serves as a platform for running a browser for accessing the firewall's web interface (UI) for post-installation configuration.

Jump VM

Suggested specifications:

- vCPU: 4
- RAM: 8GB
- Storage: 100GB
- Network interface: Bridge device, connected to `mgmt-br`

Procedure:

1. Proceed with a standard Ubuntu 24.04 installation. Use the following login credentials across all hosts in this setup:

Username	Password
depuser	user

2. Enable internet connectivity and DNS resolution by creating the following Netplan configuration:

> ✓ **Note**
>
> Use `10.0.110.254` as a temporary DNS nameserver until the MaaS VM is installed and configured. After completing the MaaS installation, update the

Netplan file to replace this address with the MaaS IP: `10.0.110.252`.

Jump Node netplan — Collapse Source

```
network:
    ethernets:
        enp1s0:
            dhcp4: false
            addresses: [10.0.110.253/24]
            nameservers:
              search: [dpf.rdg.local.domain]
              addresses: [10.0.110.254]
            routes:
              - to: default
                via: 10.0.110.254
    version: 2
```

3. Apply the configuration:

Jump Node Console — Collapse Source

```
depuser@jump:~$ sudo netplan apply
```

4. Update and upgrade the system:

Jump Node Console — Collapse Source

```
depuser@jump:~$ sudo apt update -y
depuser@jump:~$ sudo apt upgrade -y
```

5. Install and configure the **Xfce desktop environment** and **XRDP** (complementary packages for RDP):

Jump Node Console Collapse Source

```
depuser@jump:~$ sudo apt install -y xfce4 xfce4-goodies
depuser@jump:~$ sudo apt install -y lightdm-gtk-greeter
depuser@jump:~$ sudo apt install -y xrdp
depuser@jump:~$ echo "xfce4-session" | tee .xsession
depuser@jump:~$ sudo systemctl restart xrdp
```

6. Install Firefox for accessing the Firewall web interface:

Jump Node Console Collapse Source

```
$ sudo apt install -y firefox
```

7. Install and configure an NFS server with the `/mnt/dpf_share` directory:

Jump Node Console Collapse Source

```
$ sudo apt install -y nfs-server
$ sudo mkdir -m 777 /mnt/dpf_share
$ sudo vi /etc/exports
```

8. Add the following line to `/etc/exports`:

Jump Node Console Collapse Source

```
/mnt/dpf_share 10.0.110.0/24(rw,sync,no_subtree_check)
```

9. Restart the NFS server:

Jump Node Console Collapse Source

```
$ sudo systemctl restart nfs-server
```

10. Create the directory `bfb` under `/mnt/dpf_share` with the same permissions as the parent directory:

Jump Node Console — Collapse Source

```
$ sudo mkdir -m 777 /mnt/dpf_share/bfb
```

11. Generate an SSH key pair for `depuser` in the jump node (later on will be imported to the admin user in MaaS to enable password-less login to the provisioned servers):

Jump Node Console — Collapse Source

```
depuser@jump:~$ ssh-keygen -t rsa
```

12. Reboot the jump node to display the graphical user interface:

Jump Node Console — Collapse Source

```
depuser@jump:~$ sudo reboot
```

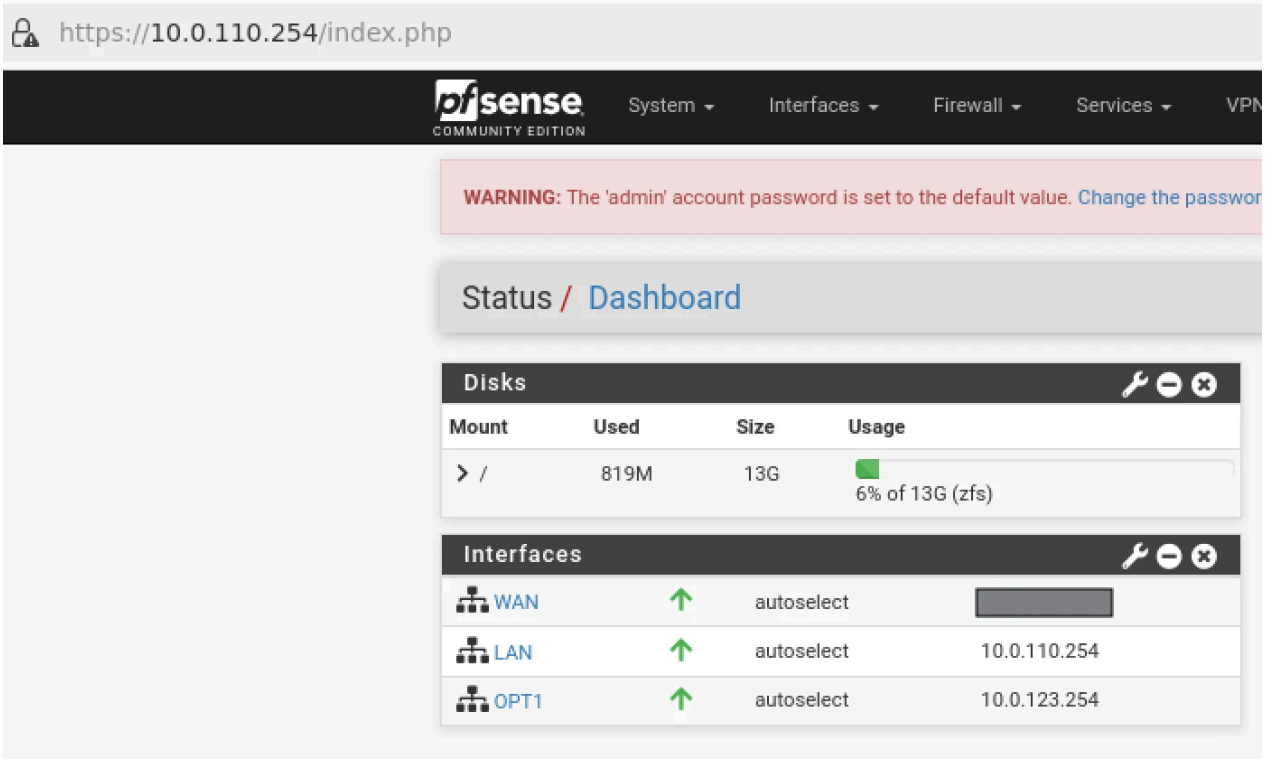
> **Note**
>
> After setting up port-forwarding rules on the firewall (next steps), remote login to the graphical interface of the Jump node will be available. Concurrent login to the local graphical console and using RDP isn't possible, make sure to first log out from the local console when switching to RDP connection.

Firewall VM – Web Configuration

From your Jump node, open a Firefox web browser and navigate to the pfSense web UI (`http://10.0.110.254`. The default login credentials are `admin/pfsense`). The login page should appear as follows:

✓ Note

The IP addresses from the trusted LAN network under "DNS servers" and "Interfaces - WAN" are blurred.

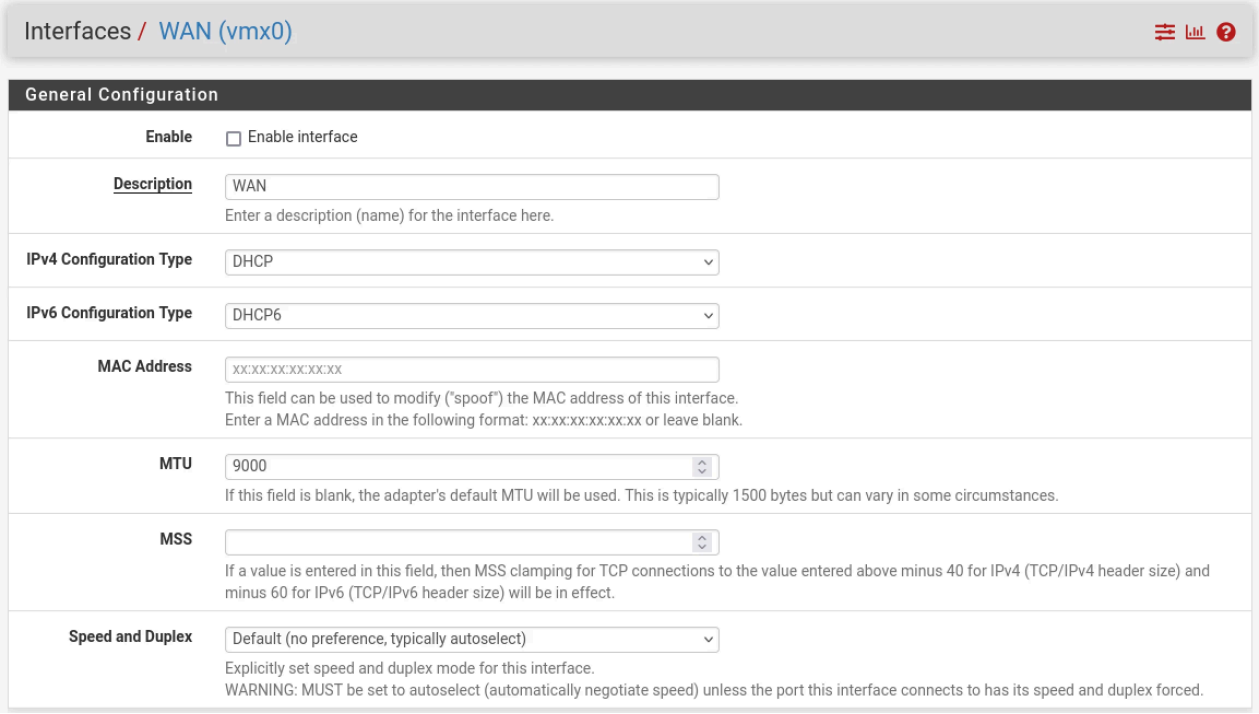


Proceed with the following configurations:

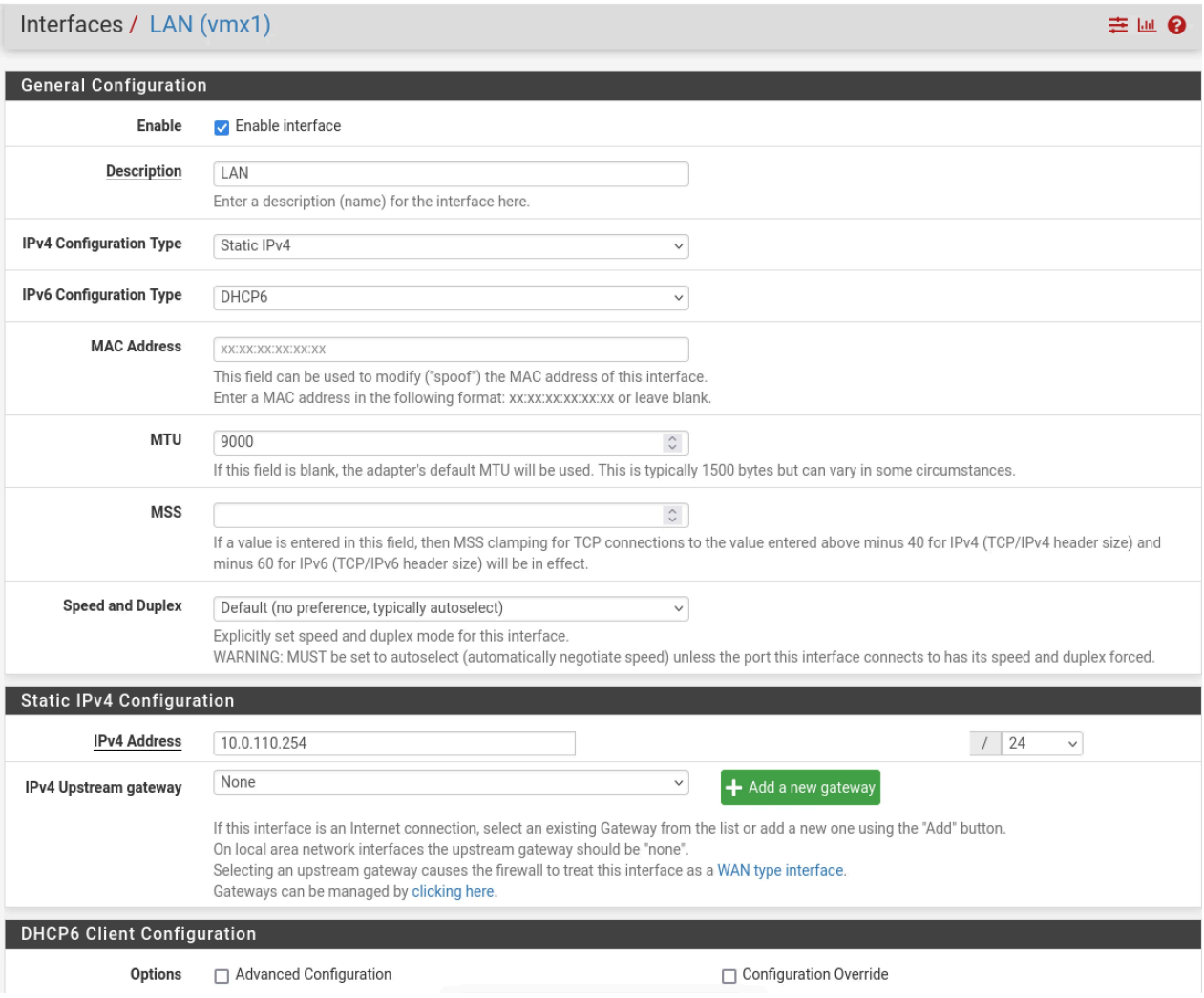
✓ Note

The following screenshots display only a part of the configuration view. Make sure to not miss any of the steps mentioned below!

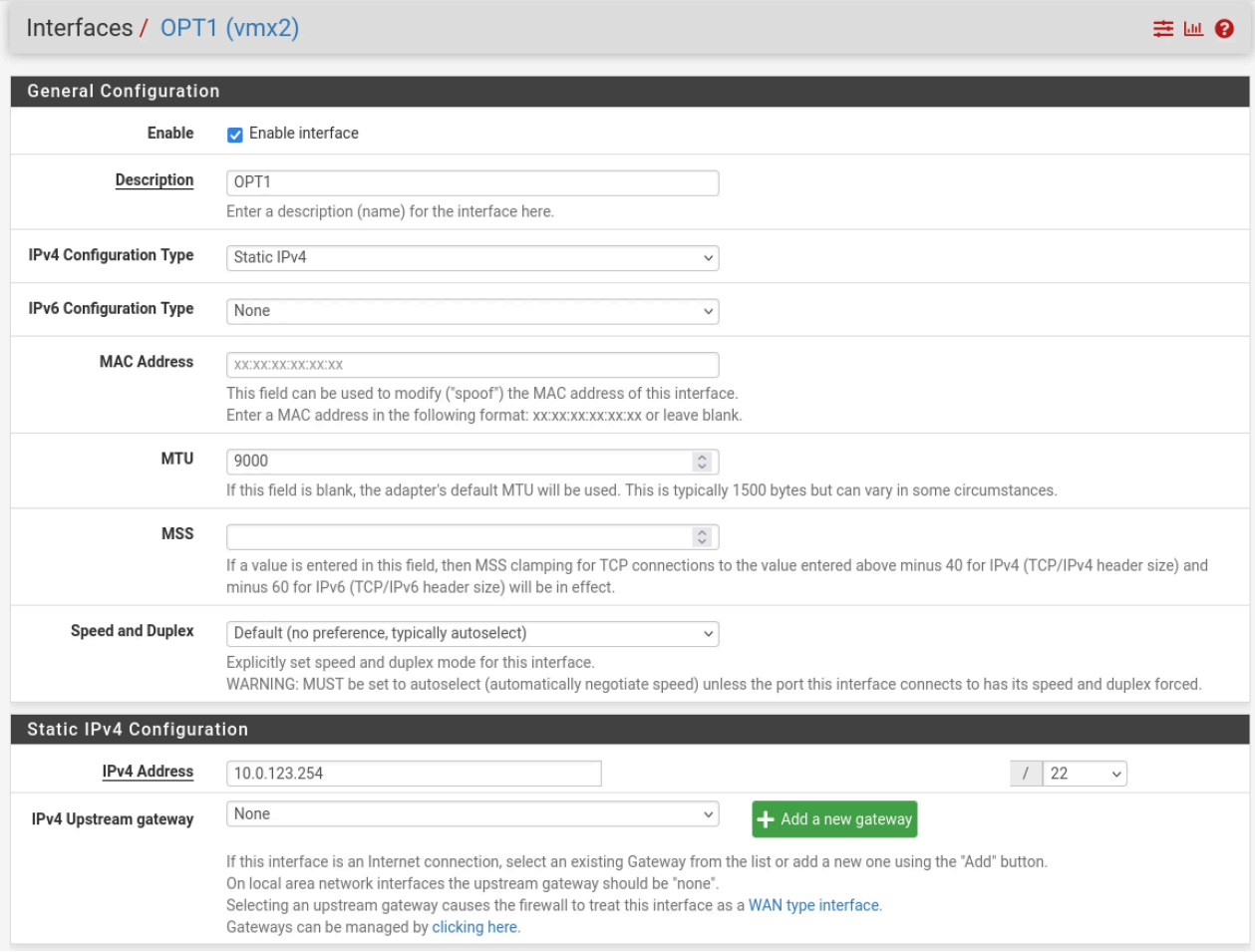
- Interfaces:
  - WAN (lab-br) – mark “Enable interface”, unmark “Block private networks and loopback addresses”



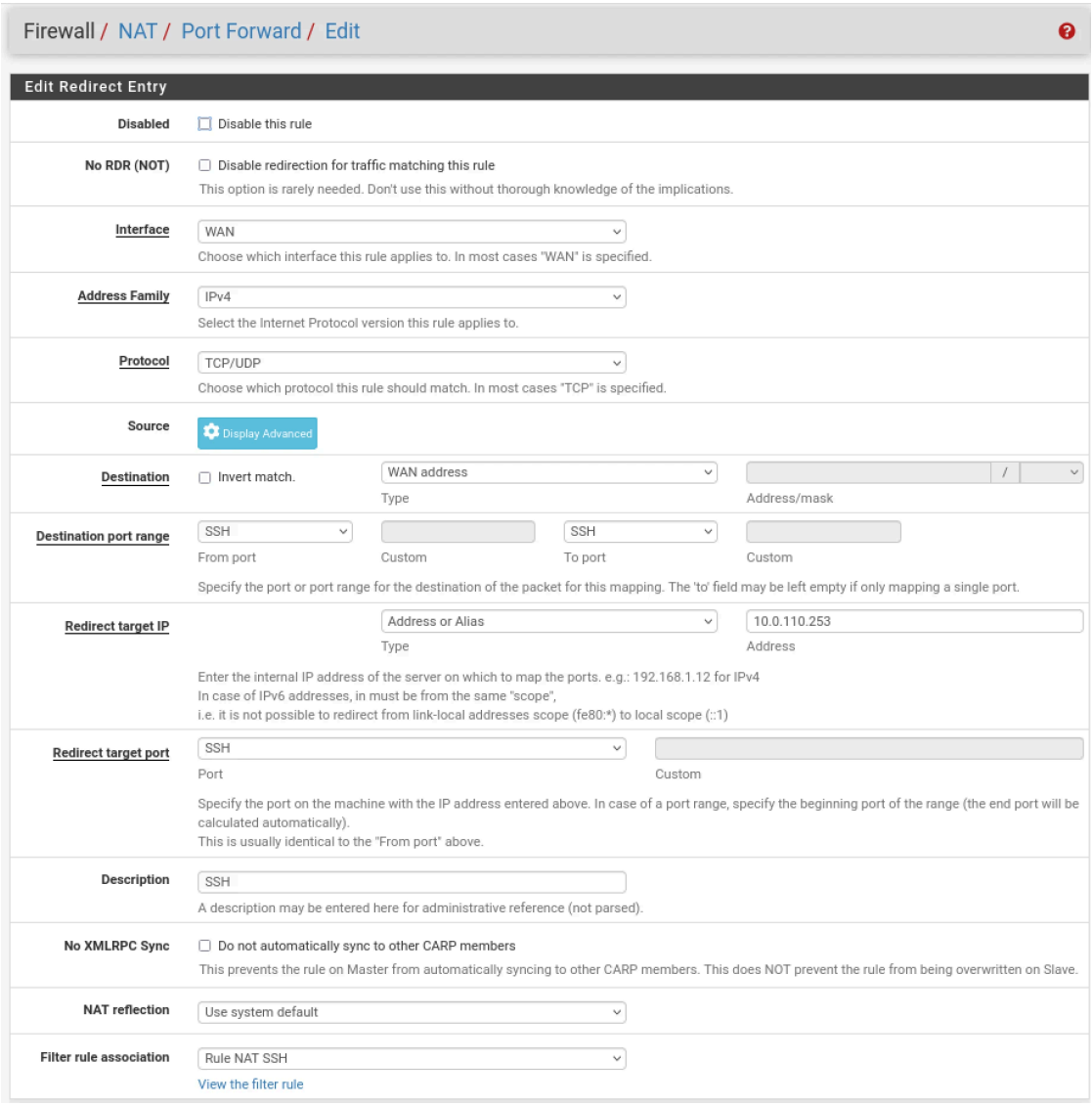
- LAN (mgmt-br) – mark “Enable interface”, “IPv4 configuration type”: **Static IPv4** ("IPv4 Address": **10.0.110.254/24**, "IPv4 Upstream Gateway": **None**), “MTU”: **9000**



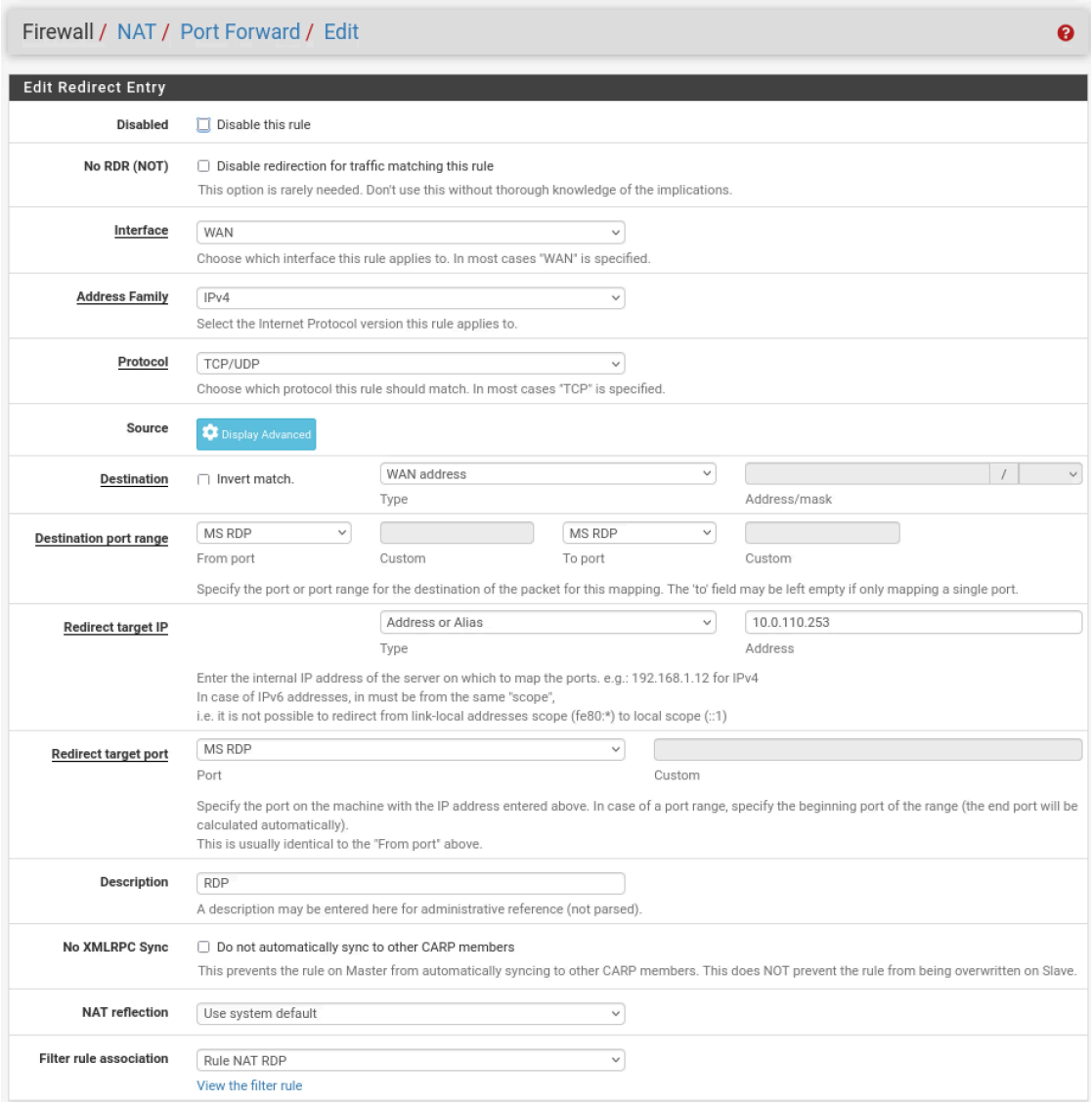
- - OPT1 (hs-br) – mark “Enable interface”, “IPv4 configuration type”: **Static IPv4** ("IPv4 Address": **10.0.123.254/22,** "IPv4 Upstream Gateway": **None**), “MTU”: **9000**

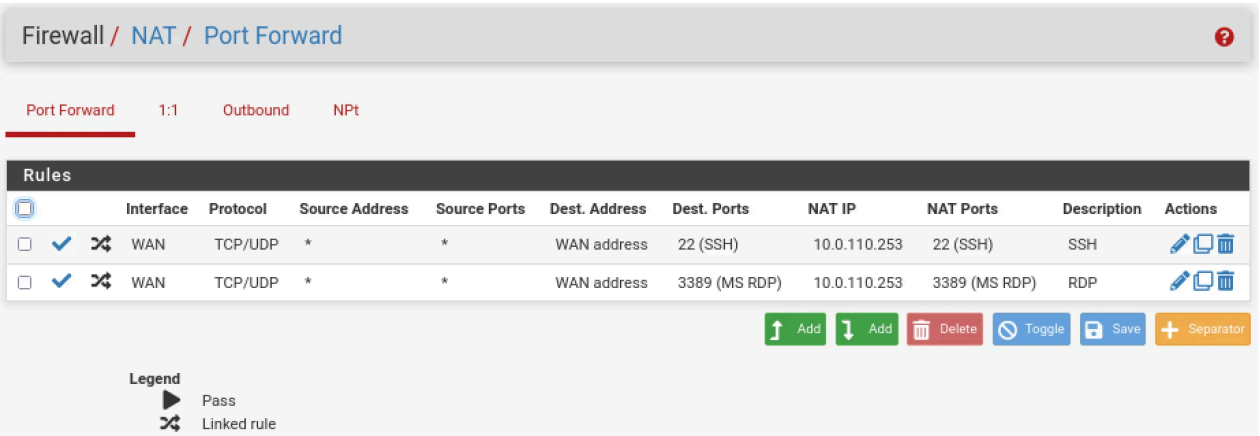


- Firewall:
  - NAT -> Port Forward -> Add rule -> “Interface”: **WAN**, “Address Family”: **IPv4**, “Protocol”: **TCP**, “Destination”: **WAN address**, “Destination port range”: (“From port”: **SSH**, “To port”: **SSH**), “Redirect target IP”: (“Type”: **Address or Alias**, “Address”: **10.0.110.253**), “Redirect target port”: **SSH**, “Description”: **NAT SSH**

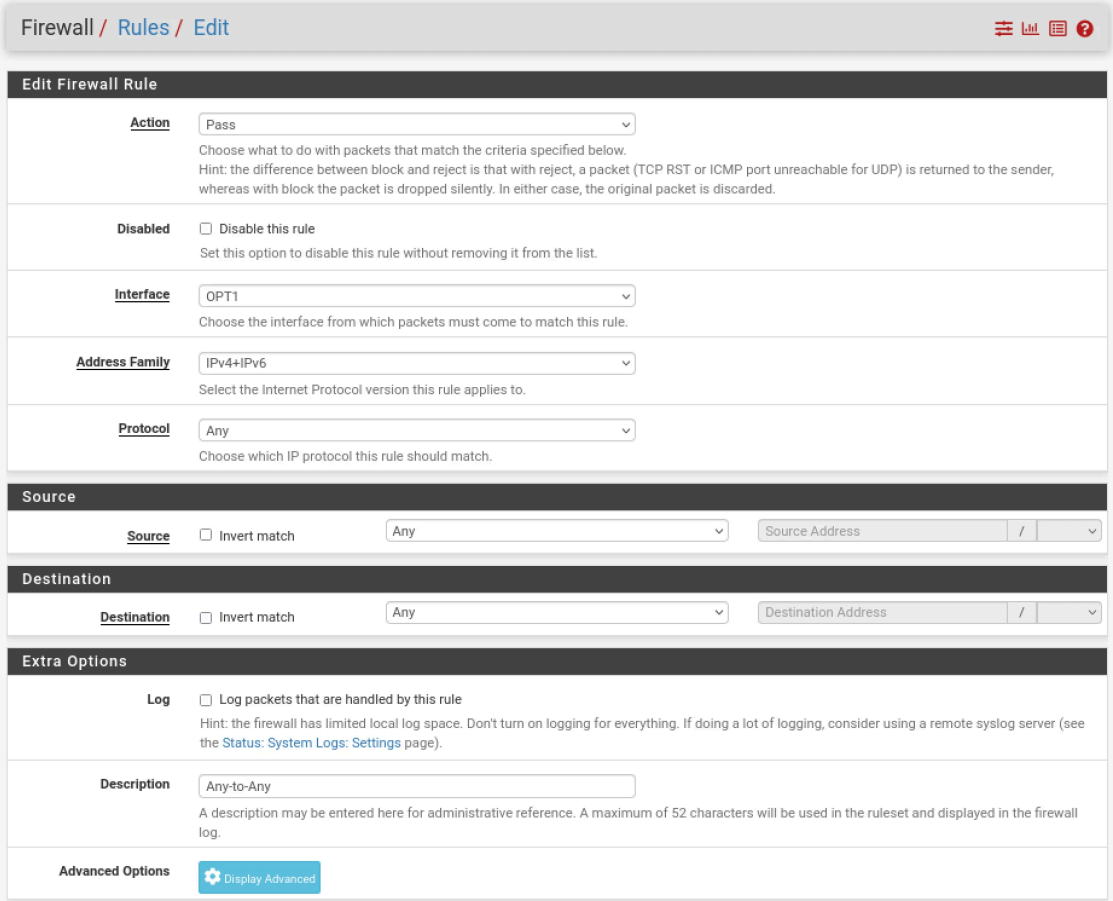


- NAT -> Port Forward -> Add rule -> “Interface”: **WAN**, “Address Family”: **IPv4**, “Protocol”: **TCP**, “Destination”: **WAN address**, “Destination port range”: (“From port”: **MS RDP**, “To port”: **MS RDP**), “Redirect target IP”: (“Type”: **Address or Alias**, “Address”: **10.0.110.253**), “Redirect target port”: **MS RDP**, “Description”: **NAT RDP**





- Rules -> OPT1 -> Add rule -> “Action”: **Pass** , “Interface”: **OPT1** , “Address Family”: **IPv4+IPv6** , “Protocol”: **Any** , “Source”: **Any** , “Destination”: **Any**



MaaS VM

Suggested specifications:

- vCPU: 4
- RAM: 4 GB
- Storage: 100 GB
- Network interface: Bridge device, connected to `mgmt-br`

Procedure:

1. Perform a regular Ubuntu installation on the MaaS VM.
2. Create the following Netplan configuration to enable internet connectivity and DNS resolution:

> ✓ **Note**
>
> Use `10.0.110.254` as a temporary DNS nameserver. After the MaaS installation, replace this with the MaaS IP address (`10.0.110.252`) in both the Jump and MaaS VM Netplan files.

Collapse Source

MaaS netplan

```
network:
    ethernets:
        enp1s0:
            dhcp4: false
            addresses: [10.0.110.252/24]
            nameservers:
              search: [dpf.rdg.local.domain]
              addresses: [10.0.110.254]
            routes:
              - to: default
                via: 10.0.110.254
    version: 2
```

3. Apply the netplan configuration:

MaaS Console — Collapse Source

```
depuser@maas:~$ sudo netplan apply
```

4. Update and upgrade the system:

MaaS Console — Collapse Source

```
depuser@maas:~$ sudo apt update -y
depuser@maas:~$ sudo apt upgrade -y
```

5. Install PostgreSQL and configure the database for MaaS:

MaaS Console — Collapse Source

```
$ sudo -i
# apt install -y postgresql
# systemctl disable --now systemd-timesyncd
# export MAAS_DBUSER=maasuser
```

```
# export MAAS_DBPASS=maaspass
# export MAAS_DBNAME=maas
# sudo -i -u postgres psql -c "CREATE USER \"$MAAS_DBUSER\" WITH E
# sudo -i -u postgres createdb -O "$MAAS_DBUSER" "$MAAS_DBNAME"
```

6. Install MaaS:

MaaS Console — Collapse Source

```
# snap install maas
```

7. Initialize MaaS:

MaaS Console — Collapse Source

```
# maas init region+rack --maas-url http://10.0.110.252:5240/MAAS -
```

8. Create an admin account:

MaaS Console — Collapse Source

```
# maas createadmin --username admin --password admin --email admir
```

9. Save the admin API key:

MaaS Console — Collapse Source

```
# maas apikey --username admin > admin-apikey
```

10. Log in to the MaaS server:

Collapse Source

MaaS Console

```
# maas login admin http://localhost:5240/MAAS "$(cat admin-apikey)
```

11. Configure MaaS (Substitute **<Trusted_LAN_NTP_IP>** and **<Trusted_LAN_DNS_IP>** with the IP addresses in your environment):

Collapse Source

MaaS Console

```
# maas admin domain update maas name="dpf.rdg.local.domain"
# maas admin maas set-config name=ntp_servers value="<Trusted_LAN_
# maas admin maas set-config name=network_discovery value="disable
# maas admin maas set-config name=upstream_dns value="<Trusted_LAN
# maas admin maas set-config name=dnssec_validation value="no"
# maas admin maas set-config name=default_osystem value="ubuntu"
```

12. Define and configure IP ranges and subnets:

Collapse Source

MaaS Console

```
# maas admin ipranges create type=dynamic start_ip="10.0.110.51" e
# maas admin ipranges create type=dynamic start_ip="10.0.110.225"
# maas admin ipranges create type=reserved start_ip="10.0.110.10"
# maas admin ipranges create type=reserved start_ip="10.0.110.200"
# maas admin ipranges create type=reserved start_ip="10.0.110.251"
# maas admin vlan update 0 untagged dhcp_on=True primary_rack=maas
# maas admin dnsresources create fqdn=kube-vip.dpf.rdg.local.domai
# maas admin dnsresources create fqdn=jump.dpf.rdg.local.domain ip
# maas admin dnsresources create fqdn=fw.dpf.rdg.local.domain ip_a
```

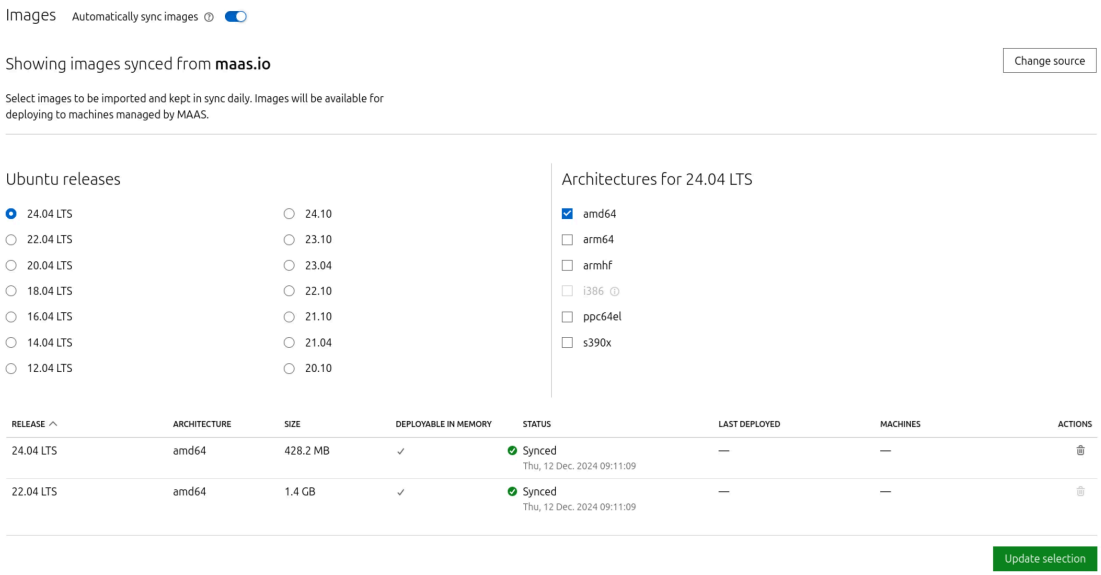
13. Configure **static DHCP leases** for **all your DPU nodes** (replace MAC address as appropriate with your workers **DPU BMC/OOB interface MAC**):

Collapse Source

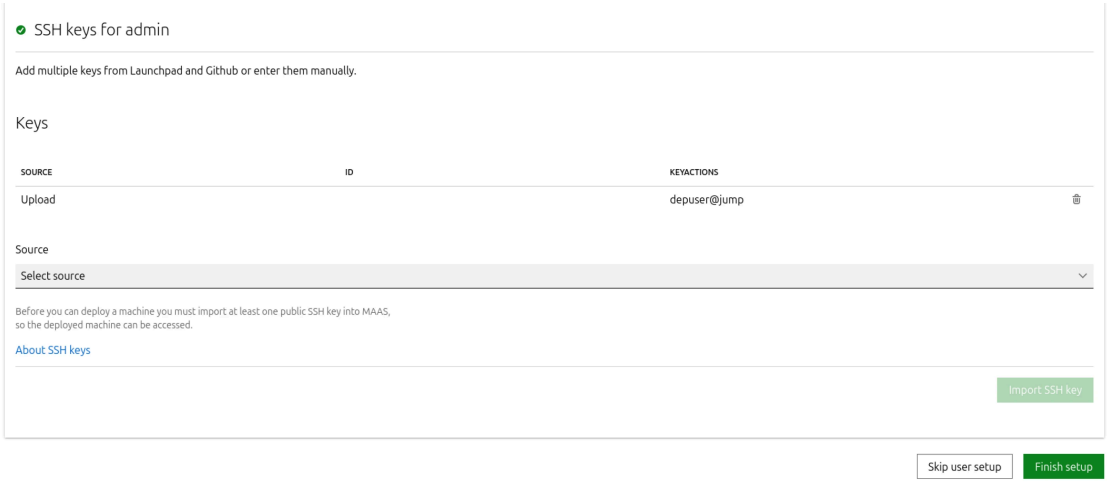
MaaS Console

```
# maas admin reserved-ips create ip="10.0.110.201" mac_address="58
# maas admin reserved-ips create ip="10.0.110.211" mac_address="58
# maas admin reserved-ips create ip="10.0.110.202" mac_address="58
# maas admin reserved-ips create ip="10.0.110.212" mac_address="58
# maas admin reserved-ips create ip="10.0.110.203" mac_address="58
# maas admin reserved-ips create ip="10.0.110.213" mac_address="58
# maas admin reserved-ips create ip="10.0.110.204" mac_address="58
# maas admin reserved-ips create ip="10.0.110.214" mac_address="58
```

14. Complete MaaS setup:
    a. Connect to the Jump node GUI and access the MaaS UI at `http://10.0.110.252:5240/MAAS`.
    b. On the first page, verify the "Region Name" and "DNS Forwarder," then continue.
    c. On the image selection page, select **Ubuntu 24.04 LTS (amd64)** and sync the image.



    d. Import the previously generated SSH key (`id_rsa.pub`) for the `depuser` into the MaaS admin user profile and finalize the setup.



15. Update the DNS nameserver IP address in the Netplan files for both the Jump and MaaS VMs from `10.0.110.254` to `10.0.110.252`, then reapply the configuration.

K8s Master VMs

Suggested specifications:

- vCPU: 8
- RAM: 16GB
- Storage: 100GB
- Network interface: Bridge device, connected to `mgmt-br`

1. Before provisioning the Kubernetes (K8s) Master VMs with MaaS, create the required virtual disks with empty storage. Use the following one-liner to create three 100 GB QCOW2 virtual disks:

   Hypervisor Console — Collapse Source

   ```
   $ for i in $(seq 1 3); do qemu-img create -f qcow2 /var/lib/libvir
   ```

   This command generates the following disks in the `/var/lib/libvirt/images/` directory:

   - `master1.qcow2`
   - `master2.qcow2`
   - `master3.qcow2`

2. Configure VMs in virt-manager:

   a. Open **virt-manager** and create three virtual machines:

- Assign the corresponding virtual disk (`master1.qcow2`, `master2.qcow2`, or `master3.qcow2`) to each VM.
- Configure each VM with the suggested specifications (vCPU, RAM, storage, and network interface).

b. During the VM setup, ensure the **NIC** is selected under the **Boot Options** tab. This ensures the VMs can PXE boot for MaaS provisioning.

c. Once the configuration is complete, shut down all the VMs.

3. After the VMs are created and configured, proceed to provision them via the MaaS interface. MaaS will handle the OS installation and further setup as part of the deployment process.

Provision Master VMs Using MaaS

Install virsh and Set Up SSH Access

1. SSH to the MaaS VM from the Jump node:

MaaS Console — Collapse Source

```
depuser@jump:~$ ssh maas
depuser@maas:~$ sudo -i
```

2. Install the `virsh` client to communicate with the hypervisor:

MaaS Console — Collapse Source

```
# apt install -y libvirt-clients
```

3. Generate an SSH key for the `root` user and copy it to the hypervisor user in the `libvirtd` group:

MaaS Console — Collapse Source

```
# ssh-keygen -t rsa
# ssh-copy-id ubuntu@<hypervisor_MGMT_IP>
```

4. Verify SSH access and `virsh` communication with the hypervisor:

MaaS Console — Collapse Source

```
# virsh -c qemu+ssh://ubuntu@<hypervisor_MGMT_IP>/system list --al
```

Expected output:

MaaS Console — Collapse Source

```
 Id   Name          State
------------------------------
 1    fw     running
 2    jump   running
 3    maas   running
 -    master1       shut off
 -    master2       shut off
 -    master3       shut off
```

5. Copy the SSH key to the required MaaS directory (for snap-based installations):

MaaS Console — Collapse Source

```
# mkdir -p /var/snap/maas/current/root/.ssh
# cp .ssh/id_rsa* /var/snap/maas/current/root/.ssh/
```

Get MAC Addresses of the Master VMs

Retrieve the MAC addresses of the Master VMs:

MaaS Console — Collapse Source

```
# for i in $(seq 1 3); do virsh -c qemu+ssh://ubuntu@<hypervisor_MGMT_I
```

Example output:

MaaS Console — Collapse Source

```
<mac address='52:54:00:a9:9c:ef'/>
<mac address='52:54:00:19:6b:4d'/>
<mac address='52:54:00:68:39:7f'/>
```

Add Master VMs to MaaS

1. Add the Master VMs to MaaS:

> **Info**
> Once added, MaaS will automatically start the newly added VMs commissioning (discovery and introspection).

MaaS Console — Collapse Source

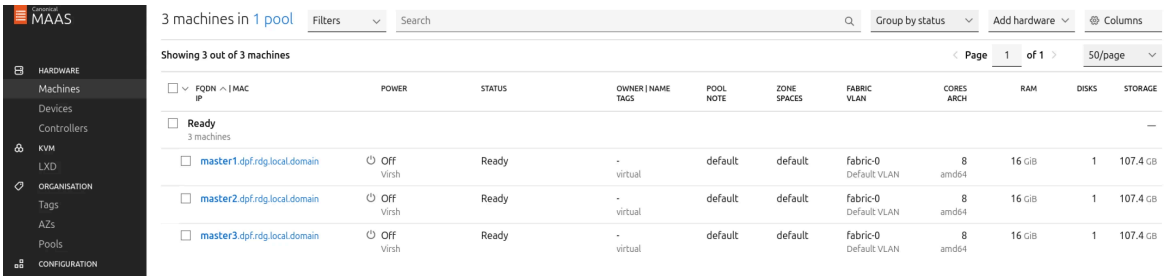
```
# maas admin machines create hostname=master1 architecture=amd64/g
Success.
Machine-readable output follows:
{
    "description": "",
    "status_name": "Commissioning",
...
    "status": 1,
...
    "system_id": "c3seyq",
...
    "fqdn": "master1.dpf.rdg.local.domain",
    "power_type": "virsh",
...
```

```
    "status_message": "Commissioning",
    "resource_uri": "/MAAS/api/2.0/machines/c3seyq/"
}

# maas admin machines create hostname=master2 architecture=amd64/g

# maas admin machines create hostname=master3 architecture=amd64/g
```

2. Repeat the command for `master2` and `master3` with their respective MAC addresses.
3. Verify commissioning by waiting for the status to change to "Ready" in MaaS.



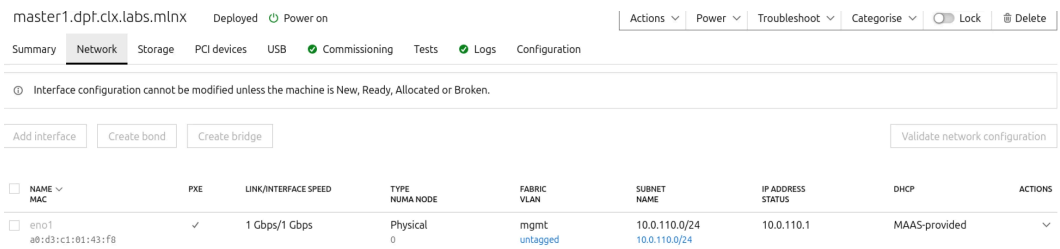
After commissioning, the next phase is deployment (OS provisioning).

Configure Master VMs Network

To ensure persistence across reboots, assign a static IP address to the management interface of the master nodes.

For each Master VM:

1. Navigate to **Network** and click "actions" near the management interface (a small arrowhead pointing down), then select "Edit Physical".
   a. Configure as follows:
      i. **Subnet:** 10.0.110.0/24
      ii. **IP Mode:** Static Assign
      iii. **Address:** Assign `10.0.110.1` for `master1`, `10.0.110.2` for `master2`, and `10.0.110.3` for `master3`.



2. Save the interface settings for each VM.

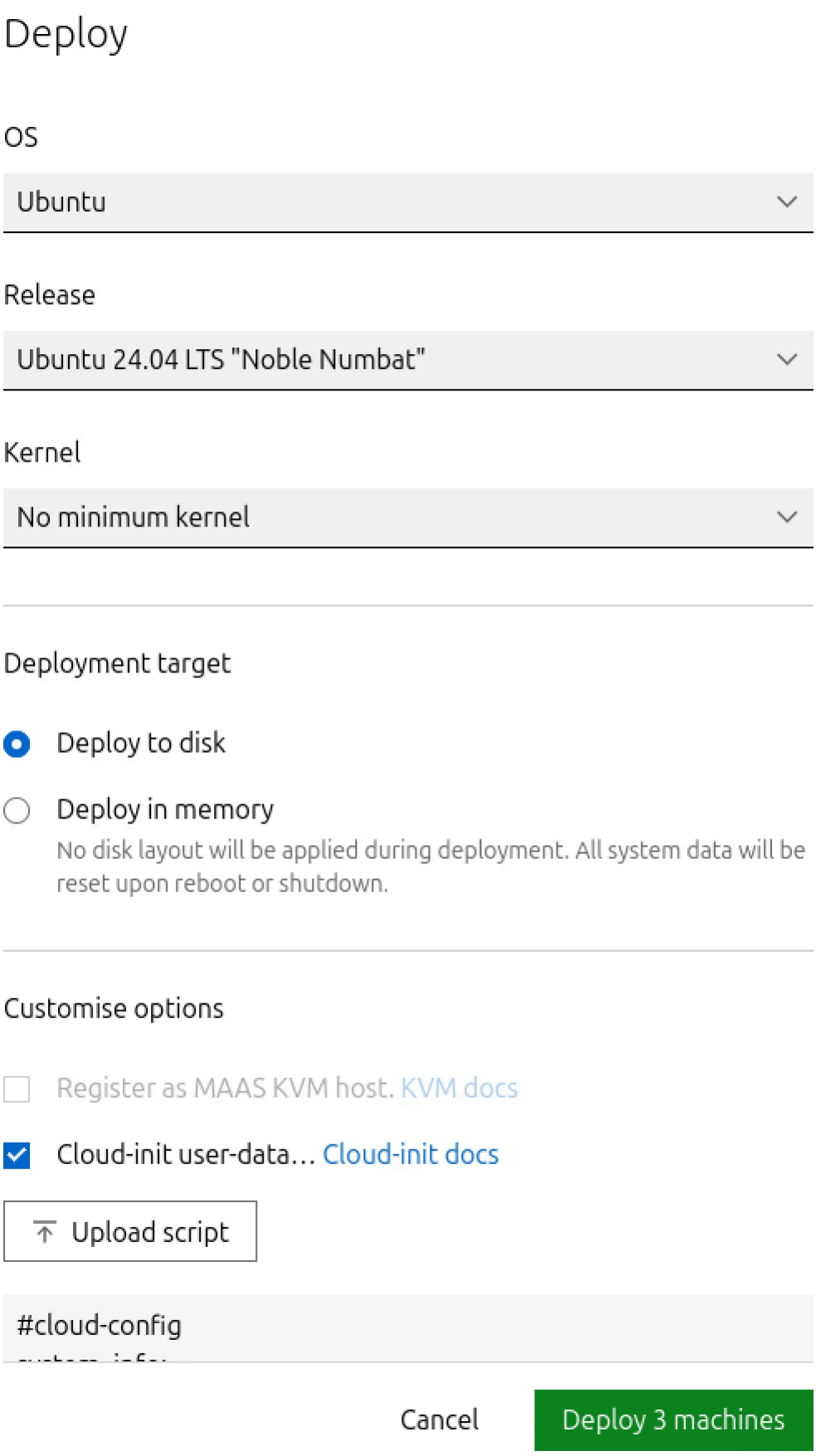
Deploy Master VMs Using Cloud-Init

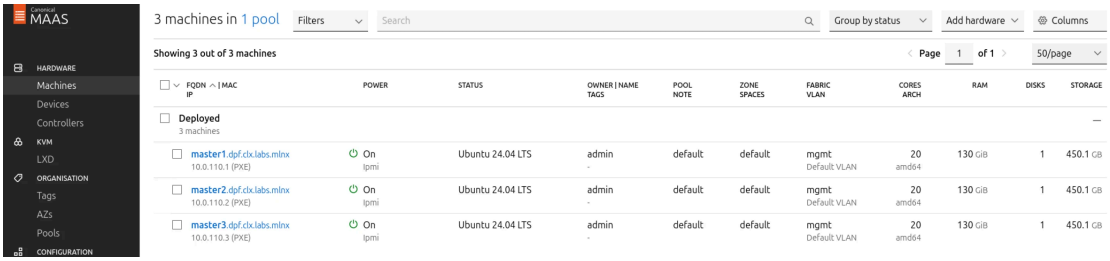
1. Use the following cloud-init script to configure the necessary software and ensure persistency:

Master nodes cloud-init — Collapse Source

```
#cloud-config
system_info:
  default_user:
    name: depuser
    passwd: "$6$jOKPZPHD9XbG72lJ$evCabLvy1GEZ5OR1Rrece3NhWpZ2CnS0E
    lock_passwd: false
    groups: [adm, audio, cdrom, dialout, dip, floppy, lxd, netdev,
    sudo: ["ALL=(ALL) NOPASSWD:ALL"]
    shell: /bin/bash
ssh_pwauth: True
package_upgrade: true
runcmd:
    - apt-get update
    - apt-get -y install nfs-common
```

2. Deploy the master VMs:
   a. Select all three Master VMs → **Actions** → **Deploy**.
   b. Toggle **Cloud-init user-data** and paste the cloud-init script.
   c. Start the deployment and wait for the status to change to **"Ubuntu 24.04 LTS"**.


Deploy
OS
Ubuntu
Release
Ubuntu 24.04 LTS "Noble Numbat"
Kernel
No minimum kernel
Deployment target
Deploy to disk
Deploy in memory
No disk layout will be applied during deployment. All system data will be reset upon reboot or shutdown.
Customise options
Register as MAAS KVM host. KVM docs
Cloud-init user-data… Cloud-init docs
Upload script
#cloud-config
Cancel
Deploy 3 machines


MAAS
HARDWARE
Machines
Devices
Controllers
KVM
LXD
ORGANISATION
Tags
AZs
Pools
CONFIGURATION
3 machines in 1 pool
Filters
Search
Group by status
Add hardware
Columns
Showing 3 out of 3 machines
Page 1 of 1
50/page
FQDN / IP | MAC
POWER
STATUS
OWNER | NAME / TAGS
POOL / NOTE
ZONE / SPACES
FABRIC / VLAN
CORES / ARCH
RAM
DISKS
STORAGE
Deployed
3 machines
master1.dpf.clx.labs.mlnx
10.0.110.1 (PXE)
On
Ipmi
Ubuntu 24.04 LTS
admin
default
default
mgmt
Default VLAN
20
amd64
130 GiB
1
450.1 GB
master2.dpf.clx.labs.mlnx
10.0.110.2 (PXE)
master3.dpf.clx.labs.mlnx
10.0.110.3 (PXE)

Verify Deployment

- SSH into the Master VMs from the Jump node:

Jump Node Console — Collapse Source

```
depuser@jump:~$ ssh master1
depuser@master1:~$
```

- Run `sudo` without a password:

Master1 Console — Collapse Source

```
depuser@master1:~$ sudo -i
root@master1:~#
```

- Verify installed packages:

Master1 Console — Collapse Source

```
root@master1:~# apt list --installed | egrep 'nfs-common'
nfs-common/noble,now 1:2.6.4-3ubuntu5 amd64 [installed]
```

- Reboot the Master VMs to complete the provisioning.

Master1 Console — Collapse Source

```
root@master1:~# reboot
```

Repeat the verification commands for `master2` and `master3.`

K8s Cluster Deployment and Configuration

Kubespray Deployment and Configuration

In this solution, the Kubernetes (K8s) cluster is deployed using a **modified Kubespray** (based on release `v2.28.1`) with a non-root `depuser` account from the Jump Node. The modifications in Kubespray are designed to meet the DPF prerequisites as described in the User Manual and facilitate cluster deployment and scaling.

1. Download the modified Kubespray archive: **modified_kubespray_v2.28.1.tar.gz**.
2. Extract the contents and navigate to the extracted directory:

Jump Node Console — Collapse Source

```
$ tar -xzf /home/depuser/modified_kubespray_v2.28.1.tar.gz
$ cd kubespray/
depuser@jump:~/kubespray$
```

3. Set the K8s API VIP address and DNS record. Replace it with your own IP address and DNS record if different:

Jump Node Console — Collapse Source

```
depuser@jump:~/kubespray$ sed -i '/# kube_vip_address:/s/.*/kube_v
depuser@jump:~/kubespray$ sed -i '/apiserver_loadbalancer_domain_r
```

4. Install the necessary dependencies and set up the Python virtual environment:

Jump Node Console — Collapse Source

```
depuser@jump:~/kubespray$ sudo apt -y install python3-pip jq pytho
depuser@jump:~/kubespray$ python3 -m venv .venv
```

```
depuser@jump:~/kubespray$ source .venv/bin/activate
(.venv) depuser@jump:~/kubespray$ python3 -m pip install --upgrade
(.venv) depuser@jump:~/kubespray$ pip install -U -r requirements.t
(.venv) depuser@jump:~/kubespray$ pip install ruamel-yaml
```

5. Verify that the `kube_network_plugin` is set to `flannel` and that `kube_proxy_remove` is set to `false` in the `inventory/mycluster/group_vars/k8s_cluster/k8s-cluster.yml` file.

inventory/mycluster/group_vars/k8s_cluster/k8s-cluster.yml Collapse Source

```
[depuser@jump kubespray-2.28.0]$ vim inventory/mycluster/group_var
.....
## Change this to use another Kubernetes version, e.g. a current b
kube_version: 1.32.8
.....
# Choose network plugin (cilium, calico, kube-ovn, weave or flanne
# Can also be set to 'cloud', which lets the cloud provider setup
kube_network_plugin: flannel
....
# Kube-proxy proxyMode configuration.
# Can be ipvs, iptables
kube_proxy_remove: false
kube_proxy_mode: ipvs
.....
```

6. Review and edit the `inventory/mycluster/hosts.yaml` file to define the cluster nodes. The following is the configuration for this deployment:

inventory/mycluster/hosts.yaml Collapse Source

```
all:
  hosts:
    master1:
      ansible_host: 10.0.110.1
      ip: 10.0.110.1
      access_ip: 10.0.110.1
      node_labels:
```

```
        "k8s.ovn.org/zone-name": "master1"
    master2:
      ansible_host: 10.0.110.2
      ip: 10.0.110.2
      access_ip: 10.0.110.2
      node_labels:
        "k8s.ovn.org/zone-name": "master2"
    master3:
      ansible_host: 10.0.110.3
      ip: 10.0.110.3
      access_ip: 10.0.110.3
      node_labels:
        "k8s.ovn.org/zone-name": "master3"
  children:
    kube_control_plane:
      hosts:
        master1:
        master2:
        master3:
    kube_node:
      hosts:
    etcd:
      hosts:
        master1:
        master2:
        master3:
    k8s_cluster:
      children:
        kube_control_plane:
```

Deploying Cluster Using Kubespray Ansible Playbook

1. Run the following command from the Jump Node to initiate the deployment process:

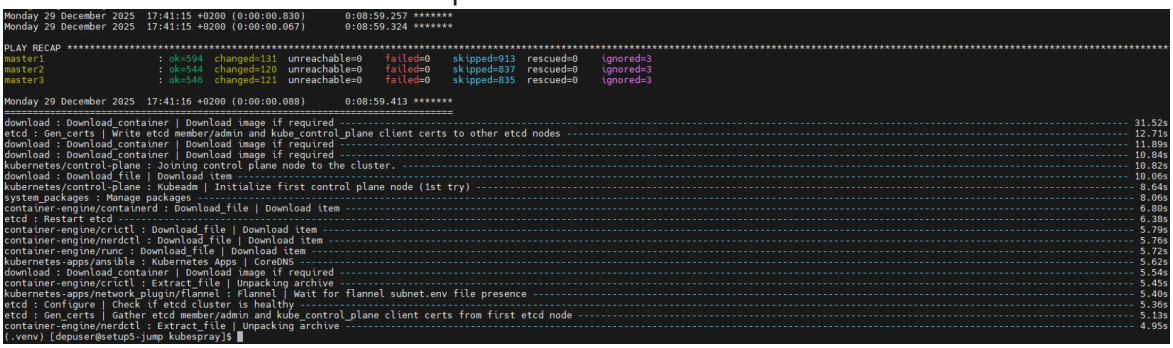
> **Note**
>
> Ensure you are in the Python virtual environment (`.venv`) when running the command.

Collapse Source

Jump Node Console

```
(.venv) depuser@jump:~/kubespray$ ansible-playbook -i inventory/my
```

2. It takes a while for this deployment to complete. Make sure there are no errors. Successful result example:



> **Tip**
>
> It is recommended to **keep the shell** from which **Kubespray has been running open**, later on it will be useful when performing cluster scale out to add the worker nodes.

K8s Deployment Verification

To simplify managing the K8s cluster from the Jump Host, set up `kubectl` with bash auto-completion.

1. Copy `kubectl` and the kubeconfig file from `master1` to the Jump Host:

Jump Node Console — Collapse Source

```
## Connect to master1
depuser@jump:~$ ssh master1
depuser@master1:~$ cp /usr/local/bin/kubectl /tmp/
depuser@master1:~$ sudo cp /root/.kube/config /tmp/kube-config
depuser@master1:~$ sudo chmod 644 /tmp/kube-config
```

2. In another terminal tab, copy the files to the Jump Host:

Jump Node Console Collapse Source

```
depuser@jump:~$ scp master1:/tmp/kubectl /tmp/
depuser@jump:~$ sudo chown root:root /tmp/kubectl
depuser@jump:~$ sudo mv /tmp/kubectl /usr/local/bin/
depuser@jump:~$ mkdir -p ~/.kube
depuser@jump:~$ scp master1:/tmp/kube-config ~/.kube/config
depuser@jump:~$ chmod 600 ~/.kube/config
```

3. Enable bash auto-completion for `kubectl`:
   a. Verify if bash-completion is installed:

Jump Node Console Collapse Source

```
depuser@jump:~$ type _init_completion
```

If installed, the output includes:

Jump Node Console Collapse Source

```
_init_completion is a function
```

   b. If not installed, install it:

Jump Node Console Collapse Source

```
depuser@jump:~$ sudo apt install -y bash-completion
```

   c. Set up the `kubectl` completion script:

Jump Node Console Collapse Source

```
depuser@jump:~$ kubectl completion bash | sudo tee /etc/bash_co
depuser@jump:~$ bash
```

4. Check the status of the nodes in the cluster:

Jump Node Console — Collapse Source

```
depuser@jump:~$ kubectl get nodes
```

Expected output:

Jump Node Console — Collapse Source

```
NAME      STATUS   ROLES           AGE     VERSION
master1   Ready    control-plane   3m59s   v1.32.8
master2   Ready    control-plane   3m51s   v1.32.8
master3   Ready    control-plane   3m48s   v1.32.8
```

5. Check the pods in all namespaces:

Jump Node Console — Collapse Source

```
depuser@jump:~$ kubectl get pods -A
```

Expected output:

Jump Node Console — Collapse Source

```
[depuser@setup5-jump ~]$ kubectl get pods -A
NAMESPACE     NAME                              READY   STATUS
kube-system   coredns-5c54f84c97-7245f          1/1     Running
kube-system   coredns-5c54f84c97-gk4pb          1/1     Running
kube-system   dns-autoscaler-56cb45595c-bv28g   1/1     Running
kube-system   kube-apiserver-master1            1/1     Running
```

```
kube-system   kube-apiserver-master2            1/1     Running
kube-system   kube-apiserver-master3            1/1     Running
kube-system   kube-controller-manager-master1   1/1     Running
kube-system   kube-controller-manager-master2   1/1     Running
kube-system   kube-controller-manager-master3   1/1     Running
kube-system   kube-flannel-cptzm                1/1     Running
kube-system   kube-flannel-m2gw4                1/1     Running
kube-system   kube-flannel-ql46d                1/1     Running
kube-system   kube-proxy-62dwr                  1/1     Running
kube-system   kube-proxy-dc4sb                  1/1     Running
kube-system   kube-proxy-mhbtb                  1/1     Running
kube-system   kube-scheduler-master1            1/1     Running
kube-system   kube-scheduler-master2            1/1     Running
kube-system   kube-scheduler-master3            1/1     Running
kube-system   kube-vip-master1                  1/1     Running
kube-system   kube-vip-master2                  1/1     Running
kube-system   kube-vip-master3                  1/1     Running
```

DPF Installation

Software Prerequisites and Required Variables

1. Start by installing the remaining **software perquisites**.

Jump Node Console — Collapse Source

```
## Connect to master1 to copy helm client utility that was install
$ depuser@jump:~$ ssh master1
depuser@master1:~$ cp /usr/local/bin/helm /tmp/

## In another tab
depuser@jump:~$ scp master1:/tmp/helm /tmp/
depuser@jump:~$ sudo chown root:root /tmp/helm
depuser@jump:~$ sudo mv /tmp/helm /usr/local/bin/

## Verify that envsubst utility is installed
depuser@jump:~$ which envsubst
/usr/bin/envsubst
```

2. Proceed to clone the **doca-platform Git repository**:

Jump Node Console — Collapse Source

```
$ git clone https://github.com/NVIDIA/doca-platform.git
```

3. Change directory to **doca-platform** and checkout to **tag v25.10.0**:

Jump Node Console — Collapse Source

```
$ cd doca-platform/
$ git checkout v25.10.0
```

4. Change directory to **readme.md** from where all the commands will be run:

Jump Node Console — Collapse Source

```
$ cd docs/public/user-guides/zero-trust/use-cases/passthrough/
```

5. Change the BMC root's password.
   In Zero Trust mode, provisioning DPUs requires authentication with Redfish. In order to do that, you must set the same root password to access the BMC for all DPUs DPF is going to manage.For more information on how to set the BMC root password refer to **BlueField DPU Administrator Quick Start Guide**. Connect to the first DPU BMC over SSH to change the BMC root's password:

Jump Node Console — Collapse Source

```
$ ssh root@10.0.110.201
root@10.0.110.201's password: <BMC Root Password. Default root/0pe
```

6. Modify the variables in `manifests/00-env-vars/envvars.env` to fit your environment, then source the file:

Warning

Replace the values for the variables in the following file with the values that fit your setup. Specifically, pay attention to `DPUCLUSTER_INTERFACE`, `BMC_ROOT_PASSWORD`, and `DPU's serial number`.
To get a `DPU's serial number` you can use following command. Sample:
$ curl -k -u root:'BMC root password'
https://10.0.110.201/redfish/v1/Systems/Bluefield | jq -r '.SerialNumber | ascii_downcase'
% Total % Received % Xferd Average Speed Time Time Time Current
Dload Upload Total Spent Left Speed
100 4970 100 4970 0 0 4211 0 0:00:01 0:00:01 --:--:-- 4211
mt2402xz0f7x

manifests/00-env-vars/envvars.env — Collapse Source

```
## IP Address for the Kubernetes API server of the target cluster
## This should never include a scheme or a port.
## e.g. 10.10.10.10
export TARGETCLUSTER_API_SERVER_HOST=10.0.110.10

## Virtual IP used by the load balancer for the DPU Cluster. Must
## allocated by DHCP.
export DPUCLUSTER_VIP=10.0.110.200

## Interface on which the DPUCluster load balancer will listen. Sh
export DPUCLUSTER_INTERFACE=ens160

## IP address to the NFS server used as storage for the BFB.
export NFS_SERVER_IP=10.0.110.253

## The DPF REGISTRY is the Helm repository URL where the DPF Opera
## Usually this is the NVIDIA Helm NGC registry. For development p
export REGISTRY=https://helm.ngc.nvidia.com/nvidia/doca

## The DPF TAG is the version of the DPF components which will be
export TAG=v25.10.0

## URL to the BFB used in the `bfb.yaml` and linked by the DPUSet.
export BFB_URL="https://content.mellanox.com/BlueField/BFBs/Ubuntu

## IP_RANGE_START and IP_RANGE_END
## These define the IP range for DPU discovery via Redfish/BMC int
```

```
## Example: If your DPUs have BMC IPs in range 10.0.110.201-240
## export IP_RANGE_START=10.0.110.201
## export IP_RANGE_END=10.0.110.204

## Start of DPUDiscovery IpRange
export IP_RANGE_START=10.0.110.201

## End of DPUDiscovery IpRange
export IP_RANGE_END=10.0.110.204

# The password used for DPU BMC root login, must be the same for a
# For more information on how to set the BMC root password refer t
export BMC_ROOT_PASSWORD=<set your BMC_ROOT_PASSWORD>
```

7. Export environment variables for the installation:

Jump Node Console — Collapse Source

```
$ source manifests/00-env-vars/envvars.env
```

DPF Operator Installation

Create Storage Required by the DPF Operator

1. Create the NS for the operator:

Jump Node Console — Collapse Source

```
$ kubectl create ns dpf-operator-system
```

2. The following YAML file defines storage (for the BFB image) that is required by the DPF operator.

manifests/01-dpf-operator-installation/nfs-storage-for-bfb-dpf-ga.yaml Collapse Source

```
---
apiVersion: v1
kind: PersistentVolume
metadata:
  name: bfb-pv
spec:
  capacity:
    storage: 10Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteMany
  nfs:
    path: /mnt/dpf_share/bfb
    server: $NFS_SERVER_IP
  persistentVolumeReclaimPolicy: Delete
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: bfb-pvc
  namespace: dpf-operator-system
spec:
  accessModes:
  - ReadWriteMany
  resources:
    requests:
      storage: 10Gi
  volumeMode: Filesystem
  storageClassName: ""
```

3. Run the following command to substitute the environment variables using `envsubst` and apply the yaml file:

Jump Node Console Collapse Source

```
$ cat manifests/01-dpf-operator-installation/*.yaml | envsubst | k
```

Verify the rshim service

Verify that the `rshim` service is runnging.

1. Connect to the first DPU BMC over SSH:

Jump Node Console — Collapse Source

```
$ ssh root@10.0.110.201
```

2. Verify that the `rshim` service is runnging or run if not.

Jump Node Console — Collapse Source

```
root@dpu-bmc:~# systemctl status rshim
* rshim.service - rshim driver for BlueField SoC
     Loaded: loaded (/usr/lib/systemd/system/rshim.service; enable
     Active: active (running) since Mon 2025-07-14 13:21:34 UTC; 1
       Docs: man:rshim(8)
    Process: 866 ExecStart=/usr/sbin/rshim $OPTIONS (code=exited,
   Main PID: 874 (rshim)
        CPU: 2h 43min 44.730s
     CGroup: /system.slice/rshim.service
             `-874 /usr/sbin/rshim

Jul 14 13:21:34 dpu-bmc (rshim)[866]: rshim.service: Referenced bu
Jul 14 13:21:34 dpu-bmc rshim[874]: Created PID file: /var/run/rsh
Jul 14 13:21:34 dpu-bmc rshim[874]: USB device detected
Jul 14 13:21:38 dpu-bmc rshim[874]: Probing usb-2.1
Jul 14 13:21:38 dpu-bmc rshim[874]: create rshim usb-2.1
Jul 14 13:21:39 dpu-bmc rshim[874]: rshim0 attached

root@dpu-bmc:~# ls /dev/rshim0
boot     console  misc     rshim

# To start the rshim, if not runnging
root@dpu-bmc:~# systemctl enable rshim
root@dpu-bmc:~# systemctl start rshim
root@dpu-bmc:~# systemctl status rshim
root@dpu-bmc:~# ls /dev/rshim0
```

```
root@dpu-bmc:~# exit
$ curl -k -u root:'set your BMC_ROOT_PASSWORD' https://10.0.110.20
```

3. **Repeat** the **step 1-2** on all your **DPUs**.

The password is provided to DPF by creating the following secret:

> **Note**
>
> It is necessary to set several **environment variables** before running this command.
>
> *$ source manifests/00-env-vars/envvars.env*

Jump Node Console — Collapse Source

```
$ kubectl create secret generic -n dpf-operator-system bmc-shared-passw
```

Additional Dependencies

The DPF Operator requires several prerequisite components to function properly in a Kubernetes environment. Starting with DPF v25.7, all Helm dependencies have been removed from the DPF chart. This means that **all dependencies must be installed manually** before installing the DPF chart itself. The following commands describe an opiniated approach to install those dependencies (for more information, check: **Helm Prerequisites - NVIDIA Docs**).

1. Install `helmfile` binary:

Jump Node Console — Collapse Source

```
$ wget https://github.com/helmfile/helmfile/releases/download/v1.1
$ tar  -xvf helmfile_1.1.2_linux_amd64.tar.gz
$ sudo mv ./helmfile /usr/local/bin/
```

2. Change directory to **doca-platform**:

> **Tip**
>
> Use another shell from the one where you run all the other installation commands for DPF.

Jump Node Console — Collapse Source

```
$ cd /home/depuser/doca-platform
```

3. Install Helm dependencies using the following command:

Jump Node Console — Collapse Source

```
$ make HELMFILE_FILE=deploy/helmfiles/prereqs.yaml test-deploy-hel
.....
UPDATED RELEASES:
NAME                     NAMESPACE                CHART
node-feature-discovery   dpf-operator-system      node-feature-dis
local-path-provisioner   local-path-provisioner   local-storage/lo
cert-manager             cert-manager             jetstack/cert-ma
argo-cd                  dpf-operator-system      argoproj/argo-cd
maintenance-operator     dpf-operator-system      oci://ghcr.io/me
kamaji                   dpf-operator-system      oci://ghcr.io/nv
```

DPF Operator Deployment

1. Run the following commands to substitute the environment variables and install the DPF Operator:

Jump Node Console — Collapse Source

```
$ helm repo add --force-update dpf-repository ${REGISTRY}
$ helm repo update
$ helm upgrade --install -n dpf-operator-system dpf-operator dpf-r
```

```
Release "dpf-operator" does not exist. Installing it now.
I1231 10:01:11.907400 2304105 warnings.go:110] "Warning: spec.temp
I1231 10:01:11.919779 2304105 warnings.go:110] "Warning: spec.jobT
NAME: dpf-operator
LAST DEPLOYED: Wed Dec 31 10:01:10 2025
NAMESPACE: dpf-operator-system
STATUS: deployed
REVISION: 1
TEST SUITE: None
```

2. Verify the DPF Operator installation by ensuring the deployment is available and all the pods are ready:

> **Note**
>
> The following verification commands may need to be run multiple times to ensure the conditions are met.

Jump Node Console — Collapse Source

```
## Ensure the DPF Operator deployment is available.
$ kubectl rollout status deployment --namespace dpf-operator-syste
deployment "dpf-operator-controller-manager" successfully rolled c

## Ensure all pods in the DPF Operator system are ready.
$ kubectl wait --for=condition=ready --namespace dpf-operator-syst
pod/argo-cd-argocd-application-controller-0 condition met
pod/argo-cd-argocd-redis-77dfd8fcb4-nq545 condition met
pod/argo-cd-argocd-repo-server-7b6c5b8cdb-mct95 condition met
pod/argo-cd-argocd-server-744d5f9c7c-4knwj condition met
pod/dpf-operator-controller-manager-645467745b-gqqsg condition met
pod/kamaji-556cb86895-99lsh condition met
pod/kamaji-etcd-0 condition met
pod/kamaji-etcd-1 condition met
pod/kamaji-etcd-2 condition met
pod/maintenance-operator-585767f779-qrfvg condition met
pod/node-feature-discovery-gc-7f64f764f8-gvdgh condition met
pod/node-feature-discovery-master-6fbc95665c-4njbd condition met
```

DPF System Installation

This section involves creating the DPF system components and some basic infrastructure required for a functioning DPF-enabled cluster.

1. Change directory back to :

Jump Node Console — Collapse Source

```
$ cd /home/depuser/doca-platform/docs/public/user-guides/zero-trus
```

2. The following YAML files define the **DPFOperatorConfig** to install the DPF System components, the **DPUCluster** to serve as the Kubernetes control plane for DPU nodes, and **DPUDiscovery** to discover **DPUDevices** and **DPUNodes**.

> **Note**
>
> Note that to achieve high performance results you need to adjust the `operatorconfig.yaml` to support MTU 9000.

manifests/02-dpf-system-installation/operatorconfig.yaml — Collapse Source

```
---
apiVersion: operator.dpu.nvidia.com/v1alpha1
kind: DPFOperatorConfig
metadata:
  name: dpfoperatorconfig
  namespace: dpf-operator-system
spec:
  dpuDetector:
    disable: true
  provisioningController:
    bfbPVCName: "bfb-pvc"
    dmsTimeout: 900
    installInterface:
      installViaRedfish:
        # Set this to the IP of one of your control plane nodes +
        bfbRegistryAddress: "$TARGETCLUSTER_API_SERVER_HOST:8080"
```

```
        skipDpuNodeDiscovery: false
  kamajiClusterManager:
    disable: false
  networking:
    highSpeedMTU: 9000
```

manifests/02-dpf-system-installation/dpucluster.yaml Collapse Source

```
---
apiVersion: provisioning.dpu.nvidia.com/v1alpha1
kind: DPUCluster
metadata:
  name: dpu-cplane-tenant1
  namespace: dpu-cplane-tenant1
spec:
  type: kamaji
  maxNodes: 10
  clusterEndpoint:
    # deploy keepalived instances on the nodes that match the give
    keepalived:
      # interface on which keepalived will listen. Should be the c
      interface: $DPUCLUSTER_INTERFACE
      # Virtual IP reserved for the DPU Cluster load balancer. Mus
      vip: $DPUCLUSTER_VIP
      # virtualRouterID must be in range [1,255], make sure the gi
      virtualRouterID: 126
      nodeSelector:
        node-role.kubernetes.io/control-plane: ""
```

manifests/02-dpf-system-installation/dpudiscovery.yaml Collapse Source

```
---
apiVersion: provisioning.dpu.nvidia.com/v1alpha1
kind: DPUDiscovery
metadata:
  name: dpu-discovery
  namespace: dpf-operator-system
spec:
  # Define the IP range to scan
```

```
  ipRangeSpec:
    ipRange:
      startIP: $IP_RANGE_START
      endIP:   $IP_RANGE_END
  # Optional: Set scan interval
  scanInterval: "60m"
```

3. Create NS for the Kubernetes control plane of the DPU nodes:

Jump Node Console — Collapse Source

```
$ kubectl create ns dpu-cplane-tenant1
```

4. Apply the previous YAML files:

Jump Node Console — Collapse Source

```
$ cat manifests/02-dpf-system-installation/*.yaml | envsubst | kub
```

5. Verify the DPF system by ensuring that the provisioning and DPUService controller manager deployments are available, that all other deployments in the DPF Operator system are available, and that the DPUCluster is ready for nodes to join.

Jump Node Console — Collapse Source

```
## Ensure the provisioning and DPUService controller manager deplo
$ kubectl rollout status deployment --namespace dpf-operator-syste

deployment "dpf-provisioning-controller-manager" successfully roll
deployment "dpuservice-controller-manager" successfully rolled out

## Ensure all other deployments in the DPF Operator system are Ava
$ kubectl rollout status deployment --namespace dpf-operator-syste

deployment "argo-cd-argocd-applicationset-controller" successfully
```

```
deployment "argo-cd-argocd-redis" successfully rolled out
deployment "argo-cd-argocd-repo-server" successfully rolled out
deployment "argo-cd-argocd-server" successfully rolled out
deployment "dpf-operator-controller-manager" successfully rolled c
deployment "dpf-provisioning-controller-manager" successfully roll
deployment "dpuservice-controller-manager" successfully rolled out
deployment "kamaji" successfully rolled out
deployment "kamaji-cm-controller-manager" successfully rolled out
deployment "maintenance-operator" successfully rolled out
deployment "node-feature-discovery-gc" successfully rolled out
deployment "node-feature-discovery-master" successfully rolled out
deployment "servicechainset-controller-manager" successfully rolle

## Ensure bfb registry daemonset is available
$ kubectl rollout status daemonset --namespace dpf-operator-system

daemon set "bfb-registry" successfully rolled out

## Ensure the DPUCluster is ready for nodes to join.(Take time)
$ kubectl wait --for=condition=ready --namespace dpu-cplane-tenant

dpucluster.provisioning.dpu.nvidia.com/dpu-cplane-tenant1 conditio


$ kubectl get dpudiscoveries.provisioning.dpu.nvidia.com -A
NAMESPACE             NAME            LAST SCAN   FOUND DPUS
dpf-operator-system   dpu-discovery   92s         4
```

6. Verify the DPF system by ensuring that the `DPUDevices` exist:

Jump Node Console — Collapse Source

```
$ kubectl get dpudevices -n dpf-operator-system
NAME           READY
mt2402xz0f7x
mt2402xz0f80
mt2402xz0f8g
mt2402xz0f9n
```

7. Verify the DPF system by ensuring that the `DPUNodes` exist.

Jump Node Console Collapse Source

```
$ kubectl get dpunodes -n dpf-operator-system
NAME                    AGE
dpu-node-mt2402xz0f7x   3m17s
dpu-node-mt2402xz0f80   3m16s
dpu-node-mt2402xz0f8g   3m15s
dpu-node-mt2402xz0f9n   3m15s
```

Congratulations, the DPF system has been successfully installed!

(Optional) DPF system installation verification

For the verification phase, our focus is on provisioning NVIDIA® BlueField®-3 DPUs through DPF and configuring them to operate in passthrough mode.

1. Export environment variables for the installation:

Jump Node Console Collapse Source

```
$ source manifests/00-env-vars/envvars.env
```

2. Use the following YAMLs to define a `BFB` resource, a `DPUFlavor`, a `DPUSet`, a `DPUServiceInterfaces`, and a `DPUServiceChain`:

manifests/03-dpf-object-installation/bfb.yaml Collapse Source

```
---
apiVersion: provisioning.dpu.nvidia.com/v1alpha1
kind: BFB
metadata:
  name: bf-bundle
  namespace: dpf-operator-system
spec:
  url: $BFB_URL
```

Expand

manifests/03-dpf-object-installation/dpuflavor.yaml Collapse Source

```
---
apiVersion: provisioning.dpu.nvidia.com/v1alpha1
kind: DPUFlavor
metadata:
  name: passthrough-$TAG
  namespace: dpf-operator-system
spec:
  dpuMode: zero-trust
  grub:
    kernelParameters:
    - console=hvc0
    - console=ttyAMA0
    - earlycon=pl011,0x13010000
    - fixrttc
    - net.ifnames=0
    - biosdevname=0
    - iommu.passthrough=1
    - cgroup_no_v1=net_prio,net_cls
    - hugepagesz=2048kB
    - hugepages=3072
  nvconfig:
  - device: "*"
    parameters:
    - PF_BAR2_ENABLE=0
    - PER_PF_NUM_SF=1
    - PF_TOTAL_SF=20
    - PF_SF_BAR_SIZE=10
    - NUM_PF_MSIX_VALID=0
    - PF_NUM_PF_MSIX_VALID=1
    - PF_NUM_PF_MSIX=228
    - INTERNAL_CPU_MODEL=1
    - INTERNAL_CPU_OFFLOAD_ENGINE=0
    - SRIOV_EN=1
    - NUM_OF_VFS=46
    - LAG_RESOURCE_ALLOCATION=1
    - LINK_TYPE_P1=ETH
    - LINK_TYPE_P2=ETH
    - EXP_ROM_UEFI_x86_ENABLE=1
  ovs:
```

manifests/03-dpf-object-installation/dpuset.yaml Collapse Source

```
---
apiVersion: provisioning.dpu.nvidia.com/v1alpha1
kind: DPUSet
metadata:
  name: passthrough
  namespace: dpf-operator-system
spec:
  dpuNodeSelector:
    matchLabels:
      feature.node.kubernetes.io/dpu-enabled: "true"
  dpuTemplate:
    spec:
      dpuFlavor: passthrough-$TAG
      bfb:
        name: bf-bundle-$TAG
      nodeEffect:
        hold: true
```

Note

Please notice that with default nodeEffect above, DPU provisioning workflow will be paused and wait for an external signal (annotation) in order to proceed, as demonstrated in upcoming steps.
To implement a fully automated process that won't require user intervention, see **customAction option**.

Expand

manifests/03-dpf-object-installation/ifaces.yaml Collapse Source

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceInterface
metadata:
```

```
  name: p0
  namespace: dpf-operator-system
spec:
  template:
    spec:
      template:
        metadata:
          labels:
            interface: "p0"
        spec:
          interfaceType: physical
          physical:
            interfaceName: p0
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceInterface
metadata:
  name: p1
  namespace: dpf-operator-system
spec:
  template:
    spec:
      template:
        metadata:
          labels:
            interface: "p1"
        spec:
          interfaceType: physical
          physical:
            interfaceName: p1
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceInterface
metadata:
  name: pf0hpf
  namespace: dpf-operator-system
```

manifests/03-dpf-object-installation/dpuservicechain.yaml Collapse Source

```
---
apiVersion: svc.dpu.nvidia.com/v1alpha1
kind: DPUServiceChain
metadata:
  name: passthrough
```

```
    namespace: dpf-operator-system
  spec:
    template:
      spec:
        template:
          spec:
            switches:
              - ports:
                - serviceInterface:
                    matchLabels:
                      interface: p0
                - serviceInterface:
                    matchLabels:
                      interface: pf0hpf
              - ports:
                - serviceInterface:
                    matchLabels:
                      interface: p1
                - serviceInterface:
                    matchLabels:
                      interface: pf1hpf
```

3. Run the command;

Jump Node Console — Collapse Source

```
$ cat manifests/03-dpf-object-installation/*.yaml | envsubst |kube
```

This will deploy the following objects:

a. BFB to download the BFB to a shared volume

b. DPUFlavor used for provisioning the DPUs

c. DPUSet to provision DPUs on worker nodes

d. DPUServiceInterfaces used by the DPUServiceChain

4. To follow the progress of DPU provisioning, run the following command to check its current phase:

Jump Node Console — Collapse Source

```
$ watch -n10 "kubectl describe dpu -n dpf-operator-system | grep '
```

5. Wait for the **NodeEffect stage** (at this point the provisioning is paused, waintig for external signal).
   Run following command on all/specific DPU nodemaintanace object/s to proceed with provisioning:

Jump Node Console — Collapse Source

```
$ kubectl annotate dpunodemaintenances -n dpf-operator-system --al
```

6. To follow the progress of DPU provisioning, run the following command to check its current phase:

Expand

Jump Node Console — Collapse Source

```
$ watch -n10 "kubectl describe dpu -n dpf-operator-system | grep '
Every 10.0s: kubectl describe dpu -n dpf-operator-system | grep 'N

  Dpu Node Name:                          dpu-node-mt2402xz0f7x
    Last Transition Time:  2025-12-29T17:48:17Z
    Type:                  BFBPrepared
    Last Transition Time:  2025-12-29T17:48:13Z
    Type:                  BFBReady
    Last Transition Time:  2025-12-29T17:52:56Z
    Type:                  BFBTransferred
    Last Transition Time:  2025-12-29T17:48:16Z
    Type:                  FWConfigured
    Last Transition Time:  2025-12-29T17:47:28Z
    Type:                  Initialized
    Last Transition Time:  2025-12-29T17:48:14Z
    Type:                  InterfaceInitialized
    Last Transition Time:  2025-12-29T17:48:13Z
    Type:                  NodeEffectReady
    Last Transition Time:  2025-12-29T18:31:22Z
    Reason:                OemLastState
```

```
    Type:                  OSInstalled
    Last Transition Time:  2025-12-29T18:34:23Z
    Type:                  Rebooted
  Phase:                Rebooting
  Dpu Node Name:                            dpu-node-mt2402xz0f80
    Last Transition Time:  2025-12-29T17:48:18Z
    Type:                  BFBPrepared
    Last Transition Time:  2025-12-29T17:48:13Z
    Type:                  BFBReady
    Last Transition Time:  2025-12-29T17:52:56Z
    Type:                  BFBTransferred
    Last Transition Time:  2025-12-29T17:48:16Z
    Type:                  FWConfigured
    Last Transition Time:  2025-12-29T17:47:28Z
    Type:                  Initialized
    Last Transition Time:  2025-12-29T17:48:15Z
    Type:                  InterfaceInitialized
    Last Transition Time:  2025-12-29T17:48:14Z
    Type:                  NodeEffectReady
    Last Transition Time:  2025-12-29T18:30:22Z
    Reason:                OomLastState
```

or with dpfctl:

Expand

Collapse Source

Jump Node Console

```
$ kubectl -n dpf-operator-system exec deploy/dpf-operator-controll
NAME                                          NAMESPACE
DPFOperatorConfig/dpfoperatorconfig           dpf-operator-syste
│           ├─Ready
│           │
│           └─SystemComponentsReady
│
├─DPUServiceChains
│ └─DPUServiceChain/passthrough               dpf-operator-syst
├─DPUServiceInterfaces
│ └─4 DPUServiceInterfaces...                 dpf-operator-syst
├─DPUServiceNADs
│ └─DPUServiceNAD/mybrsfc                     dpf-operator-syst
└─DPUSets
  └─DPUSet/passthrough                        dpf-operator-syst
    │           └─Ready
    ├─BFB/bf-bundle-v25.10.0                  dpf-operator-syst
```

```
        ├─DPUNodes
        │ ├─DPUNode/dpu-node-mt2402xz0f7x          dpf-operator-syst
        │ │ │           └─Ready
        │ │ └─DPUDevices
        │ │   └─DPUDevice/mt2402xz0f7x             dpf-operator-syst
        │ ├─DPUNode/dpu-node-mt2402xz0f80          dpf-operator-syst
        │ │ │           └─Ready
        │ │ └─DPUDevices
        │ │   └─DPUDevice/mt2402xz0f80            dpf-operator-syst
        │ ├─DPUNode/dpu-node-mt2402xz0f8g          dpf-operator-syst
        │ │ │           └─Ready
        │ │ └─DPUDevices
        │ │   └─DPUDevice/mt2402xz0f8g            dpf-operator-syst
        │ └─DPUNode/dpu-node-mt2402xz0f9n          dpf-operator-syst
        │   │           └─Ready
        │   └─DPUDevices
        │     └─DPUDevice/mt2402xz0f9n            dpf-operator-syst
        └─DPUs
          ├─DPU/dpu-node-mt2402xz0f7x-mt2402xz0f7x  dpf-operator-syst
          │             ├─Rebooted
          │             └─Ready
          ├─DPU/dpu-node-mt2402xz0f80-mt2402xz0f80  dpf-operator-syst
          │             ├─Rebooted
          │             └─Ready
```

7. Wait for the **Rebooted stage** and then **Power Cycle** the bare-metal host manual.
   After the DPU is up, run following command for all/each DPU worker:

Jump Node Console — Collapse Source

```
$ kubectl annotate dpunodes -n dpf-operator-system --all provisior
```

8. At this point, the DPU workers should be added to the cluster. As they being added to the cluster, the DPUs are provisioned.

Expand

Jump Node Console — Collapse Source

```
$ watch -n10 "kubectl describe dpu -n dpf-operator-system | grep '
Every 10.0s: kubectl describe dpu -n dpf-operator-system | grep 'N

  Dpu Node Name:                              dpu-node-mt2402xz0f7x
    Type:       InternalIP
    Type:       Hostname
    Last Transition Time:  2025-12-29T19:41:21Z
    Type:                  Ready
    Last Transition Time:  2025-12-29T17:48:17Z
    Type:                  BFBPrepared
    Last Transition Time:  2025-12-29T17:48:13Z
    Type:                  BFBReady
    Last Transition Time:  2025-12-29T17:52:56Z
    Type:                  BFBTransferred
    Last Transition Time:  2025-12-29T19:41:20Z
    Type:                  DPUClusterReady
    Last Transition Time:  2025-12-29T17:48:16Z
    Type:                  FWConfigured
    Last Transition Time:  2025-12-29T17:47:28Z
    Type:                  Initialized
    Last Transition Time:  2025-12-29T17:48:14Z
    Type:                  InterfaceInitialized
    Last Transition Time:  2025-12-29T17:48:13Z
    Type:                  NodeEffectReady
    Last Transition Time:  2025-12-29T19:41:20Z
    Type:                  NodeEffectRemoved
    Last Transition Time:  2025-12-29T18:31:22Z
    Reason:                OemLastState
    Type:                  OSInstalled
    Last Transition Time:  2025-12-29T19:41:20Z
    Type:                  Rebooted
  Phase:                Ready
  Dpu Node Name:                              dpu-node-mt2402xz0f80
    Type:       InternalIP
    Type:       Hostname
    Last Transition Time:  2025-12-29T19:41:21Z
    Type:                  Ready
    Last Transition Time:  2025-12-29T17:48:18Z
    Type:                  BFBPrepared
    Last Transition Time:  2025-12-29T17:48:13Z
```

9. Finally, validate that all the different DPU-related objects are now in the Ready state:

Collapse Source

Jump Node Console

```
$ kubectl get secrets -n dpu-cplane-tenant1 dpu-cplane-tenant1-adm

$ echo "alias ki='KUBECONFIG=/home/depuser/dpu-cluster.config kube

$ ki get node -A
dpu-node-mt2402xz0f7x-mt2402xz0f7x   Ready    <none>   2m34s   v1.
dpu-node-mt2402xz0f80-mt2402xz0f80   Ready    <none>   2m48s   v1.
dpu-node-mt2402xz0f8g-mt2402xz0f8g   Ready    <none>   2m43s   v1.
dpu-node-mt2402xz0f9n-mt2402xz0f9n   Ready    <none>   2m21s   v1.


$ kubectl get dpu -A
NAMESPACE             NAME                                 READY
dpf-operator-system   dpu-node-mt2402xz0f7x-mt2402xz0f7x   True
dpf-operator-system   dpu-node-mt2402xz0f80-mt2402xz0f80   True
dpf-operator-system   dpu-node-mt2402xz0f8g-mt2402xz0f8g   True
dpf-operator-system   dpu-node-mt2402xz0f9n-mt2402xz0f9n   True

$ kubectl wait --for=condition=ready --namespace dpf-operator-syst
dpu.provisioning.dpu.nvidia.com/dpu-node-mt2402xz0f7x-mt2402xz0f7x
dpu.provisioning.dpu.nvidia.com/dpu-node-mt2402xz0f80-mt2402xz0f80
dpu.provisioning.dpu.nvidia.com/dpu-node-mt2402xz0f8g-mt2402xz0f8g
dpu.provisioning.dpu.nvidia.com/dpu-node-mt2402xz0f9n-mt2402xz0f9n
```

Test Traffic

After the DPUs are provisioned and the rest of the objects are Ready, we can test traffic by assigning an IP on one of the PFs on the host for each DPU, and run a simple ping. This assumes that the high speed ports of the DPUs are connected and the DPUs can reach each other.

> ✓ **Note**
>
> Ubuntu 24.04 was installed on the servers.

10. Using console window , connect to the **First Worker Server**.

Jump Node Console — Collapse Source

```
$ ssh worker1
```

11. Bring up the network interface and assign it an IP address.

First Worker Server Console — Collapse Source

```
$ sudo -i
root@worker1:~# ip a s
.....
6: ens1f0np0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN
link/ether 58:a2:e1:73:69:e6 brd ff:ff:ff:ff:ff:ff
altname enp43s0f0np0
......

root@worker1:~# ip link set dev ens1f0np0 up
root@worker1:~# ip addr add 192.168.1.1/24 dev ens1f0np0
```

12. **Using another console window** , connect to the **Second Worker Server**.

Jump Node Console — Collapse Source

```
$ ssh worker2
```

13. Bring up the network interface and assign it an IP address. Run ping to the **First Worker Server**.

First Worker Server Console — Collapse Source

```
$ sudo -i
root@worker2:~# ip a s
.....
6: ens1f0np0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN
link/ether 58:a2:e1:73:6a:58 brd ff:ff:ff:ff:ff:ff
altname enp43s0f0np0
......

root@worker2:~# ip link set dev ens1f0np0 up
root@worker2:~# ip addr add 192.168.1.2/24 dev ens1f0np0
root@worker2:~# $ ping 192.168.1.1 -c3
PING 192.168.1.1 (192.168.1.1) 56(84) bytes of data.
64 bytes from 192.168.1.1: icmp_seq=1 ttl=64 time=0.377 ms
```

```
64 bytes from 192.168.1.1: icmp_seq=2 ttl=64 time=0.354 ms
64 bytes from 192.168.1.1: icmp_seq=3 ttl=64 time=0.393 ms

--- 192.168.1.1 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2053ms
rtt min/avg/max/mdev = 0.377/0.354/0.393/0.022 ms

root@worker2:~# exit
$ exit
```

Uninstall passthrough mode

14. Run the command to remove aremove `DPUServiceInterfaces`, `DPUServiceChain`, `DPUSet`, `DPUFlavor` and `BFB:`

Jump Node Console — Collapse Source

```
[depuser@setup5-jump passthrough]$ kubectl -n dpf-operator-system
dpuserviceinterface.svc.dpu.nvidia.com "p0" deleted
dpuserviceinterface.svc.dpu.nvidia.com "p1" deleted
dpuserviceinterface.svc.dpu.nvidia.com "pf0hpf" deleted
dpuserviceinterface.svc.dpu.nvidia.com "pf1hpf" deleted
[depuser@setup5-jump passthrough]$ kubectl -n dpf-operator-system
dpuservicechain.svc.dpu.nvidia.com "passthrough" deleted
[depuser@setup5-jump passthrough]$ kubectl -n dpf-operator-system
dpuset.provisioning.dpu.nvidia.com "passthrough" deleted
[depuser@setup5-jump passthrough]$ kubectl -n dpf-operator-system
dpuflavor.provisioning.dpu.nvidia.com "passthrough" deleted
[depuser@setup5-jump passthrough]$ kubectl -n dpf-operator-system
bfb.provisioning.dpu.nvidia.com "bf-bundle" deleted
```

Optional

For deploy:

RDG for DPF Zero Trust (DPF-ZT) with HBN DPU Service

RDG for DPF Zero Trust (DPF-ZT) with VPC OVN DPU service

RDG for DPF Zero Trust (DPF-ZT) with Argus DPU service

RDG for DPF Zero Trust (DPF-ZT) with DOCA Telemetry Service(DTS) and Blueman services

RDG for DPF Zero Trust (DPF-ZT) with HBN and Argus DPU Services

RDG for DPF Zero Trust (DPF-ZT) with HBN and SNAP Block Bootable Disk DPU Services

RDG for DPF Zero Trust (DPF-ZT) with VPC OVN and Argus DPU services

RDG for DPF Zero-Trust Multi-DPU: DPU1 with HBN and DPU2 with DTS/Blueman services

Authors

Boris Kovalev

Boris Kovalev has worked for the past several years as a Solutions Architect, focusing on NVIDIA Networking/Mellanox technology, and is responsible for complex machine learning, Big Data and advanced VMware-based cloud research and design. Boris previously spent more than 20 years as a senior consultant and solutions architect at multiple companies, most recently at VMware. He has written multiple reference designs covering VMware, machine learning, Kubernetes, and container solutions which are available at the NVIDIA Documents website.

Notice

Last updated on May 29, 2025