Juniper Networks | Engineering Simplicity

AI Data Center Network with Juniper Apstra, NVIDIA GPUs, and WEKA Storage—Juniper Validated Design (JVD)

Published
2025-12-23

Table of Contents

AI Data Center Network with Juniper Apstra, NVIDIA GPUs, and WEKA Storage—Juniper Validated Design (JVD)

Juniper Networks Validated Designs provide a comprehensive, end-to-end blueprint for deploying Juniper solutions in your network. These designs are created by Juniper's expert engineers and tested to ensure they meet your requirements. Using a validated design, you can reduce the risk of costly mistakes, save time and money, and ensure that your network is optimized for maximum performance.

About this Document

This document describes the design requirements and implementation details of an AI cluster infrastructure with a GPU backend IP Fabric. This fabric is built based on AI-optimized Juniper Data Center QFX series switches, and PTX Series Routers, which are configured and managed by Juniper Apstra and Terraform automation. The cluster includes Nvidia H100 DGX as well as AMD MI300X GPU servers, and WEKA Storage systems.

All validation tests were conducted in Juniper's AI Innovation Lab in Sunnyvale, CA, USA. In this open lab, Juniper collaborates closely with customers and technology partners to develop AI solutions and test deployments for a range of AI applications and models.

The AI Innovation Lab allows customers to see AI training and inference in action, running on an NVIDIA GPU and WEKA Storage cluster. Juniper performs these tests running both customer-specific models as well as those from MLCommons for MLPerf performance benchmarking and comparisons.

Solution Benefits

IN THIS SECTION

Juniper Networks has excelled in building and supporting AI networks following a scalable, robust, and automated approach suitable for a range of cluster sizes. Unlike proprietary solutions that lock in enterprises and can stifle AI innovation, Juniper's standards-based solution assures the fastest innovation, maximizes design flexibility, and prevents vendor lock-in on the Frontend, GPU Backend, and Storage Backend AI fabric networks.

The Juniper Validated Design (JVD) for AI describes a structured approach for deploying high-performance AI training and inference networks that minimize job completion time and maximize GPU performance. Additionally, it incorporates industry best practices, and leverages Juniper's extensive expertise in building high-performance data center networks.

This design in this JVD employs a 3-stage Clos IP fabric architecture utilizing Juniper QFX and PTX switches. It integrates NVIDIA GPUs and WEKA storage and is deployed and managed using Juniper's Apstra software and Terraform Automation, incorporating best practices and Juniper's extensive experience in building Data Center networks.

The integration with Juniper's Apstra software and Terraform enables customers to orchestrate the network infrastructure systematically, without requiring in-depth knowledge of the products and technologies involved. This allows customers to easily build high-capacity, easy-to-operate network fabrics that deliver high performance, increased reliability, which result in optimal JCT (Job Completion Time) and maximized GPU utilization in the AI cluster.

The solution has been extensively tested and thoroughly documented by Juniper subject matter experts, resulting in a validated design that is easy to follow, guarantees successful implementation, and simplified management and troubleshooting tasks. This document provides comprehensive guidance on how to deploy this solution, with clear descriptions of its components and step by step instructions to connect and configure them.

Juniper Validated Design Benefits

JVDs are prescriptive blueprints for building data center fabrics using repeatable, validated, predictable, and well documented network architecture solutions with guidelines for a successful deployment. Each solution has been designed, fully tested, and documented by Juniper Networks experts with all the necessary implementation details, including hardware components, software versions, connectivity, and configuration steps.

To become a validated solution (JVD) and be approved for release, a solution must pass rigorous testing with real-world workloads and applications. All features must satisfy operational and performance criteria in real-world scenarios. Testing not only includes validating the design topology and configuration steps, but also that all products in the JVD work together as expected, thereby mitigating potential risks while deploying the solution.

The core benefits of JVDs solutions can be summarized as:

- Qualified Deployments—Qualified network design blueprints for data center fabrics, that follow best practices and meet the requirements of each specific use case, and make the solution deployment quicker, simpler, and more reliable.
- Scalable—Solutions that can scale beyond the initial design and support the adoption of different hardware platforms based on customer requirements, and customers' feedback can meet the needs of most Juniper's data center customers.
- Risk Mitigation— Prescriptive implementation guidelines guarantee that you have the right products, right software versions, optimal architecture, and deployment steps.
- Systematically Verified—Tested solutions using a suite of automated testing tools validate the performance and reliability of all the components.
- Predictability— Detailed testing and careful documentation of the solution, including the capabilities and limitations of its components, guarantees that the solution will operate as expected when implemented according to the JVD guidelines.
- Repeatability— Unlocked value with repeatable network designs due to the prescriptive nature of JVD designs as well as their applicability to common use cases in the data center environment. All JVD customers benefit from lessons learned through lab testing and real-world deployments.
- Reliability— Tested with real traffic, JVD solutions are qualified to operate as designed after deployment and with real-world traffic.
- Accelerated Deployment— Ease installation with step-by-step guidance automation, and prebuilt integrations simplifies, and accelerates deployment, while reducing risks.
- Accelerated Decision-Making— Predefined combination of products, software, and architecture removes the need to spend time comparing products, and deciding how the network should be built, allowing to bridge business and technology requirements faster and also reducing risks.
- Best Practice Networks— Better outcomes for a better experience. Juniper Validated Designs have known characteristics and performance profiles to help you make informed decisions about your network.

Juniper Apstra Benefits

Juniper Validated Designs in the data center start with the Apstra software, a multi-vendor, intent-based networking system (IBNS) that provides closed-loop automation and assurance. Apstra translates vendor-agnostic business intent and technical objectives to essential policy and device-specific configurations. The system also validates user intent, as part of the initial deployment and continuously thereafter, to ensure that the network state does not deviate from the intended state. Any anomaly or deviation can be flagged, and remediation actions can be taken directly from Apstra.

The core benefits of Apstra are:

- Intent-based networking—Apstra automates configuration creation to realize the intent, deploys the configuration to appropriate devices, and continuously validates the operating state against intended state.
- Network Automation—Apstra is a multi-vendor network automation platform that is continuously updated to work with the latest hardware and is extensively tested using modern DevOps practices.
- Recoverability—The Built-in rollback capability of Apstra allows to quickly restore the system to a known-working configuration if needed.
- Day 2+ Management—Apstra's rich data analysis capabilities, including Flow Data, reduce Mean Time to Resolution (MTTR).
- Simplicity—Apstra simplifies network deployment and management. As an example, using Apstra to implement a Data Center Interconnection (DCI), reduces complexity and makes it easy to unify multiple data centers, while isolating failure domains for high availability and resilience.

AI Use Case and Reference Design

IN THIS SECTION

The **AI JVD Reference Design** covers a complete end-to-end ethernet-based AI infrastructure, which includes the Frontend fabric, GPU Backend fabric and Storage Backend fabric. These three fabrics have a symbiotic relationship, while each provides unique functions to support AI training and inference tasks. The use of Ethernet Networking in AI Fabrics enables our customers to build high-capacity, easy-to-operate network fabrics that deliver the fastest job completion times, maximize GPU utilization, and use limited IT resources.

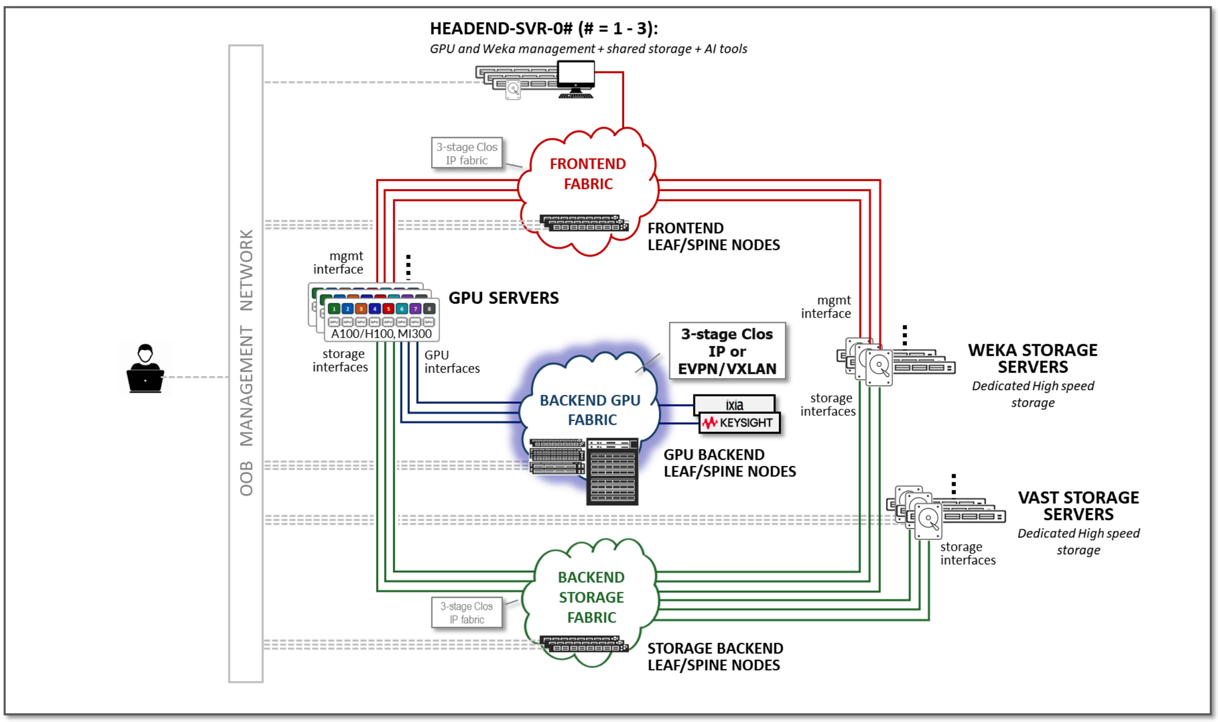
The AI JVD reference design shown in No Link Title includes:

- **Frontend Fabric**: This fabric is the gateway network to the GPU nodes and storage nodes from the AI tools residing in the headend servers. The Frontend GPU fabric allows users to interact with the GPU and storage nodes to initiate training or inference workloads and to visualize their progress and results. It also provides an out-of-band path for NCCL (NVIDIA Collective Communications Library) collective communication.

- **GPU Backend Fabric**: This fabric connects the GPU nodes (which perform the computations tasks for AI workflows). The GPU Backend fabric transfers high-speed information between GPUs during training jobs, in a lossless matter. Traffic generated by the GPUs is transferred using RoCEv2 (RDMA over Ethernet v2).

- **Storage Backend Fabric**: This fabric connects the high-availability storage systems (which hold the large model training data) and the GPUs (which consume this data during training or inference jobs). The Storage Backend fabric transfers high volumes of data in a seamless and reliable matter.

Figure 1: AI JVD Reference Design



Frontend Overview

The AI Frontend for AI encompasses the interface, tools, and methods that enable users to interact with the AI systems, and the infrastructure that allows these interactions. The Frontend gives users the ability to initiate training or inference tasks, and to visualize the results, while hiding the underlying technical complexities.

The key components of the Frontend systems include:

- **Model Scheduling**: Tools and methods for managing scripted AI model jobs and commonly based on SLURM (Simple Linux Utility for Resource Management) Workload Manager. These tools enable users to send instructions, commands, and queries, either through a shell CLI or through a graphical web-based interface to orchestrate learning and inference jobs running on the GPUs. Users can configure model parameters, input data, and interpret results as well as initiate or terminate jobs interactively. In the AI JVD, these tools are hosted on the *Headend Servers* connected to the AI Frontend fabric.

- **Management of AI Systems**: Tools for managing (configuring, monitoring and performing maintenance tasks) the AI storage and processing components. These tools facilitate building, running, training, and utilizing AI models efficiently. Examples include SLURM, TensorFlow, PyTorch, and Scikit-learn.

- **Management of Fabric Components**: Mechanisms and workflows designed to help users effortlessly deploy and manage fabric devices according to their requirements and goals. It includes tasks such as device onboarding, configuration management, and fabric deployment orchestration. This functionality is provided by ***Juniper Apstra*** .

- **Performance Monitoring and Error Analysis**: Telemetry systems tracking key performance metrics related to AI models, such as accuracy, precision, recall, and computational resource utilization (e.g. CPU, GPU usage) which are essential for evaluating model effectiveness during training and inference jobs. These systems also provide insights into error rates and failure patterns during training and inference operations, and help identify issues such as model drift, data quality problems, or algorithmic errors that may affect AI performance. Examples of these systems include Juniper Apstra dashboards, TIG Stack, and Elasticsearch.

- **Data Visualization**: Applications and tools that allow users to visually comprehend insights generated by AI models and workloads. They provide effective visualization that enhances understanding and decision-making based on AI outputs. The same telemetry systems used to monitor and measure System and Network level performance usually provide this visualization as well. Examples of these tools include Juniper Apstra dashboards, TensorFlow, and TIG stack.

- **User Interface**: routing and switching infrastructure that allows communication between the user interface applications and tools and the AI systems executing the jobs, including GPUs and storage devices. This infrastructure ensures seamless interaction between users and the computational resources needed to leverage AI capabilities effectively.

- **GPU-to-GPU control**: communication establishment, information exchange including, QP GIDs (Global IDs), Local and remote buffer addresses, and RDMA keys (RKEYs for memory access permissions)

GPU Backend Overview

The GPU Backend for AI encompasses the devices that execute learning and inference jobs or computational tasks, that is the GPU servers where the data processing occurs, and the infrastructure that allows the GPUs to communicate with each other to complete the jobs.

The key components of the GPU Backend systems include:

- **AI Systems**: Specialized hardware such as GPUs (Graphics Processing Units) and TPUs (Tensor Processing Units) that can execute numerous calculations concurrently. GPUs are particularly adept at handling AI workloads, including complex matrix multiplications and convolutions required to complete learning and inference tasks. The selection and number of GPU systems significantly impacts the speed and efficiency of these tasks.

- **AI Software**: Operating systems, libraries, and frameworks essential for developing and executing AI models. These tools provide the environment necessary for coding, training, and deploying AI algorithms effectively. The functions of these tools include:

  - **Data Management**: preprocessing, and transformation of data utilized in training and executing AI models. This encompasses tasks such as cleaning, normalization, and feature extraction. Given the volume and complexity of AI datasets, efficient data management strategies like parallel processing and distributed computing are crucial.

  - **Model Management**: tasks related to the AI models themselves, including evaluation (e.g., cross-validation), selection (choosing the optimal model based on performance metrics), and deployment (making the model accessible for real-world applications).

- **GPU Backend Fabric**: routing and switching infrastructure that allows GPU-to-GPU communication for workload distribution, memory sharing, synchronization of model parameters, exchange of results, etc. The design of this fabric can significantly impact the speed and efficiency of AI/ML model training and inference jobs and in most cases shall provide lossless connectivity for GPU-to-GPU traffic.

Storage Backend Overview

The AI storage backend for AI encompasses the hardware and software components for storing, retrieving, and managing the vast amounts of data involved in AI workloads, and the infrastructure that allows the GPUs to communicate with these storage components.

The key aspects of the storage backend include:

- **High-Performance Storage Devices**: optimized for high I/O throughput, which is essential for handling the intensive data processing requirements of the AI tasks such as deep learning. This includes high-performance storage devices designed to facilitate fast access to data during model training and to accommodate the storage needs of large datasets. These storage devices must provide:

- **Data Management Capabilities**: which support efficient data querying, indexing, and retrieval and are crucial for minimizing preprocessing and feature extraction times in AI workflows, as well as for facilitating quick data access during inference.
- **Scalability**: which accommodates growing data volumes and efficiently manages and stores massive amounts of data over time, to support AI workloads often involving large-scale datasets.

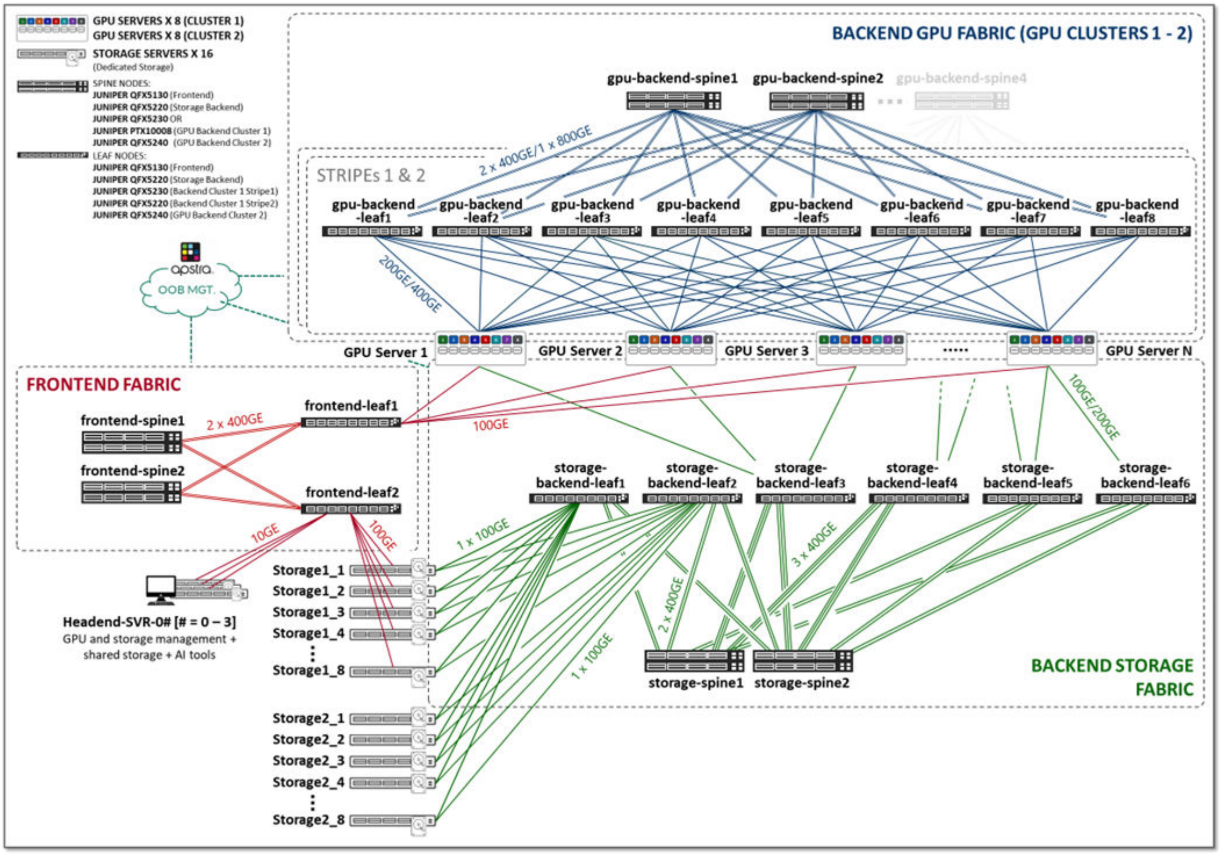
- **Storage Backend Fabric**: routing and switching infrastructure that provides the connectivity between the GPU and the storage devices. This integration ensures that data can be efficiently transferred between storage and computational resources, optimizing overall AI workflow performance. The performance of the storage backend significantly impacts the efficiency and JCT of AI/ML workflows. A storage backend that provides quick access to data can significantly reduce the amount of time for training AI/ML models.

Solution Architecture

IN THIS SECTION

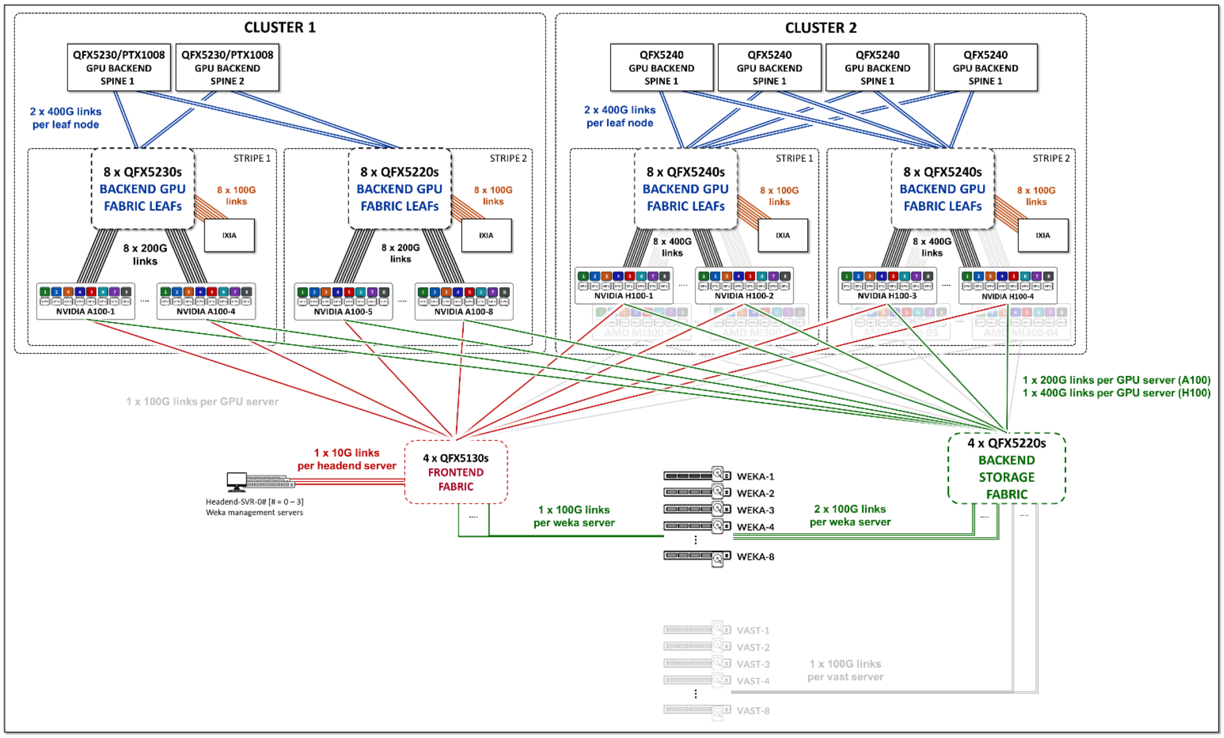
The three fabrics described in the previous section (Frontend, GPU Backend, and Storage Backend), are interconnected together in the overall AI JVD solution architecture as shown in Figure 2.

Figure 2: AI JVD Solution Architecture



We have built two different Clusters, as shown in Figure 3, which share the " **Frontend fabric** " on page 11 and " **Storage Backend fabric** " on page 26 but have separate " **GPU Backend fabrics** " on page 12. Each cluster is made of two stripes following the " **Rail Optimized Stripe Architecture** " on page 17, but include different switch models as Leaf and Spine nodes, as well as GPU server models.

Figure 3: AI JVD Lab Clusters



The GPU Backend in Cluster 1 consists of Juniper QFX5220, and QFX5230 switches as leaf nodes and either QFX5230s switches or PTX10008 routers acting as spine nodes and includes Nvidia A100 GPU servers. The QFX5230s and PTX10008 acting as spine nodes have been validated separately, while maintaining the leaf nodes the same. Apstra blueprints are used to switch between the setups with QFX5230s acting as spine nodes and the one with PTX10008 acting as spine.

The GPU Backend in Cluster 2 consists of Juniper QFX5240 switches acting as both leaf nodes and spine nodes and includes AMD MI300X GPU servers and Nvidia H100 GPU servers.

The rest of this document focuses on the Nvidia servers and Weka storage and includes server and storage configurations, specific for these systems.

It is important to notice that the type of switch and the number of switches acting as leaf and spine nodes, as well as the number and speed of the links between them, is determined by the type of fabric (Frontend, GPU Backend or Storage Backend) as they present different requirements. More details will be included in the respective fabric description sections.

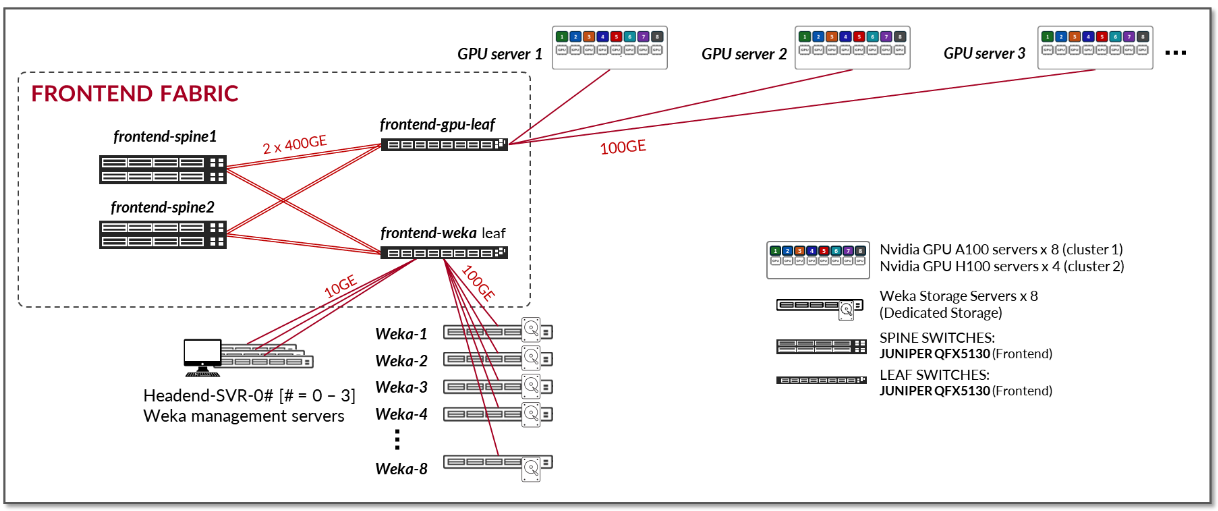
In the case of the GPU Backend fabric, the number of GPU servers, as well as the number of GPUs per server, are also factors determining the number and switch type of the leaf and spine nodes.

Frontend Fabric

The **Frontend Fabric** provides the infrastructure for users to interact with the AI systems to orchestrate training and inference tasks workflows using tools such as SLURM. These interactions do not generate heavy data flows nor have rigorous requirements regarding latency or packet drops; thus, they do not impose rigorous demands on the fabric.

The **Frontend Fabric** design described in this JVD follows a traditional 3-stage IP Fabric architecture without HA, as shown in Figure 4. This architecture provides a simple and effective solution for the connectivity required in the Frontend. However, any fabric architecture including EVPN/VXLAN, could be used. If an HA-capable Frontend Fabric is required we recommend following the 3-Stage with Juniper Apstra JVD.

Figure 4: Frontend Fabric Architecture



The devices included in the Frontend fabric, and the connections between them, are summarized in the following table:

Table 1: Frontend devices

Nvidia DGX GPU Servers	Weka Storage Servers	Headend Servers	Frontend Leaf Nodes switch model frontend-leaf# (#=1-2)	Frontend Spine Nodes switch model frontend-spine# (#=1-2)

A100 x 8 H100 x 4	Weka Storage Server x 8	Headend-SVR x 3	QFX5130-32CD x 2	QFX5130-32CD x 2

Table 2: Connections between servers, leaf and spine nodes per cluster and stripe in the Frontend

GPU Servers to <=> Frontend Leaf Nodes	Weka Storage Servers <=> Frontend Leaf Nodes	Headend Servers <=> Frontend Leaf Nodes	Frontend Spine Nodes <=> Frontend Leaf Nodes
1 x 100GE links between each GPU server A100-0#, H100-01# (#=1-8 for A100 and 1-4 for H100) and *frontend-leaf1*	1 x 100GE links between each storage server weka# (#=1-8) and *frontend-leaf2*	1 x 10GE links between each headend server *Headend-SVR-0# (#=1-3)* and *frontend-leaf2*	2 x 400GE links between each leaf node and each spine node.

This fabric is a pure L3 IP fabric using EBGP for route advertisement. The IP addressing and EBGP configuration details are described in the networking section on this document.

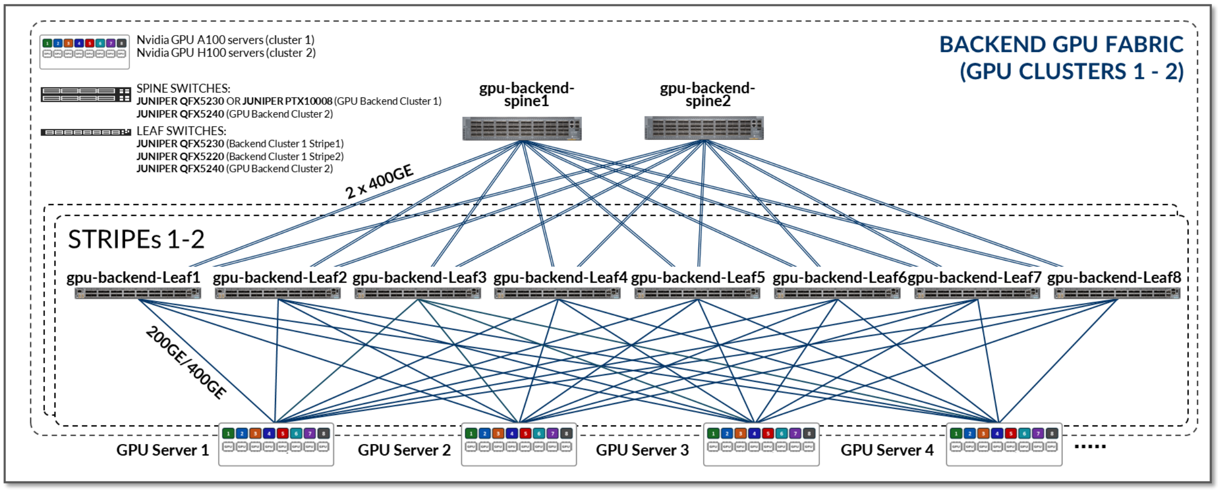
GPU Backend Fabric

The **GPU Backend fabric** provides the infrastructure for GPUs to communicate with each other within a cluster, using RDMA over Converged Ethernet (RoCEv2). ROCEv2 boosts data center efficiency, reduces overall complexity, and increases data delivery performance by enabling the GPUs to communicate as they would with the InfiniBand protocol.

Packet loss can impact job completion times and should be avoided. Therefore, when designing the network infrastructure to support RoCEv2 for an AI cluster, one of the key objectives is to provide a lossless fabric, while also achieving maximum throughput, minimal latency, and minimal network interference for the AI traffic flows. ROCEv2 is more efficient over lossless networks, resulting in optimum job completion times.

The **GPU Backend fabric** in this JVD was designed with these goals in mind and follows a 3-stage IP clos architecture combined with NVIDIA's Rail Optimized Stripe Architecture (discussed in the next section), as shown in Figure 5.

Figure 5: GPU Backend Fabric Architecture



The **GPU Backend** devices included in this fabric, and the connections between them, are summarized in the following table:

Table 3: GPU Backend devices per cluster and stripe

Cluster	Stripe	Nvidia DGX GPU Servers	GPU Backend Leaf Nodes switch model (gpu-backend-leaf#)	GPU Backend Spine Nodes switch model (gpu-backend-spine#)
1	1	*A100-01* to *A100-04*	QFX5230-64CD x 8	QFX5230-64CD x 2 OR PTX10008 w/ JNP10K-LC1201
1	2	*A100-05* to *A100-08*	QFX5220-32CD x 8	
2	1	*H100-01* to *H100-02*	QFX5240-64OD x 8	QFX5230-64OD x 4
2	2	*H100-03* to *H100-04*	QFX5240-64OD x 8	

Table 4: Connections between servers, leaf and spine nodes per cluster and stripe in the GPU Backend

Cluster	Stripe	GPU Servers <=> GPU Backend Leaf Nodes	GPU Backend Spine Nodes <=> GPU Backend Leaf Nodes

1	1	1 x 200GE links between each A100 server and each leaf node (200GE x 8 links per server)	2 x 400GE links between each leaf node and each spines node (2 x 400GE x 2 links per leaf node)
1	2	1 x 200GE links between each A100 server and each leaf nodes (200GE x 8 links per server)	2 x 400GE links between each leaf node and each spines node (2 x 400GE x 2 links per leaf node)
2	1	1 x 400GE links between each H100 server and each leaf nodes (400GE x 8 links per server)	2 x 400GE links between each leaf node and each spines node (2 x 400GE x 4 links per leaf node)
2	2	1 x 400GE links between each H100 server and each leaf nodes (400GE x 8 links per server)	2 x 400GE links between each leaf node and each spines node (2 x 400GE x 4 links per leaf node)

- All the Nvidia A100 servers in the lab are connected to the QFX5220 and QFX5230 leaf nodes in cluster 1 using 200GE interfaces, while the H100 servers are connected to the QFX5240 leaf nodes in cluster 2 using 400GE interfaces.
- This fabric is a pure L3 IP fabric (either IPv4 or IPV6) that uses EBGP for route advertisement (described in the networking section).
- Connectivity between the servers and the leaf nodes is L2 vlan-based with an IRB on the leaf nodes acting as default gateway for the servers (described in the networking section).

The speed and number of links between the GPU servers and leaf nodes and between the leaf and spine nodes determines the oversubscription factor. As an example, consider the number of GPU servers available in the lab, and how they are connected to the GPU backend fabric as described above.

Table 5: Per cluster, per stripe Server to Leaf Bandwidth

Server to Leaf Bandwidth per Stripe (per Cluster)					
Cluster	AI Systems (server type)	Servers per Stripe	Server <=> Leaf Links per Server	Bandwidth of Server <=> Leaf Links [Gbps]	Total Bandwidth Servers <=> Leaf per stripe [Tbps}

1	A100	4	8	200	4 x 8 x 200/1000 = 6.4
2	H100	2	8	400	2 x 8 x 400/1000 = 6.4

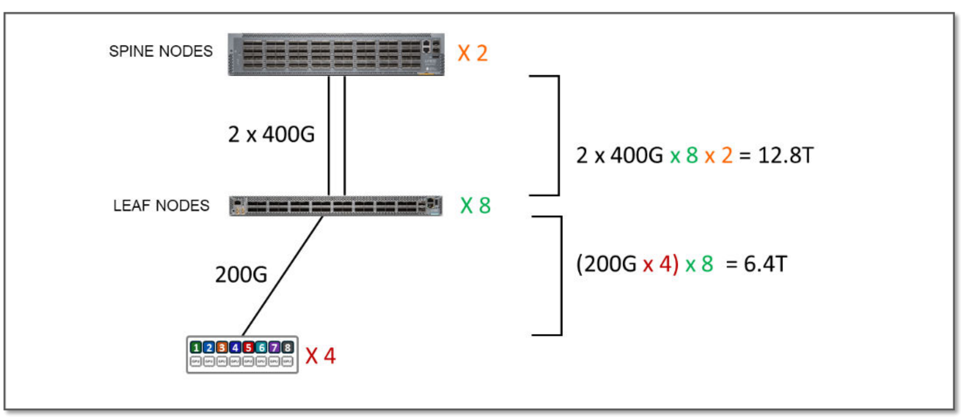
Table 6: Per cluster, per stripe Leaf to Spine Bandwidth

Leaf to Spine Bandwidth per Stripe			
Leaf <=> Spine Links Per Spine Node & Per Stripe	**Speed Of Leaf <=> Spine Links [Gbps]**	**Number of Spine Nodes**	**Total Bandwidth Leaf <=> Spine Per Stripe [Tbps]**
8	2 x 400	2	12.8

The (over)subscription rate is simply calculated by comparing the numbers from the two tables above:

In cluster 1, the bandwidth between the servers and the leaf nodes is 6.4 Tbps per stripe, while the bandwidth available between the leaf and spine nodes is 12.8 Tbps per stripe. This means that the fabric has enough capacity to process all traffic between the GPUs even when this traffic was 100% inter-stripe, while still having extra capacity to accommodate additional servers without becoming oversubscribed.

Figure 6: Extra Capacity Example



We also tested connecting the H100 GPU servers along the A100 servers to the stripes in Cluster 1 as follows:

Figure 7: 1:1 Subscription Example

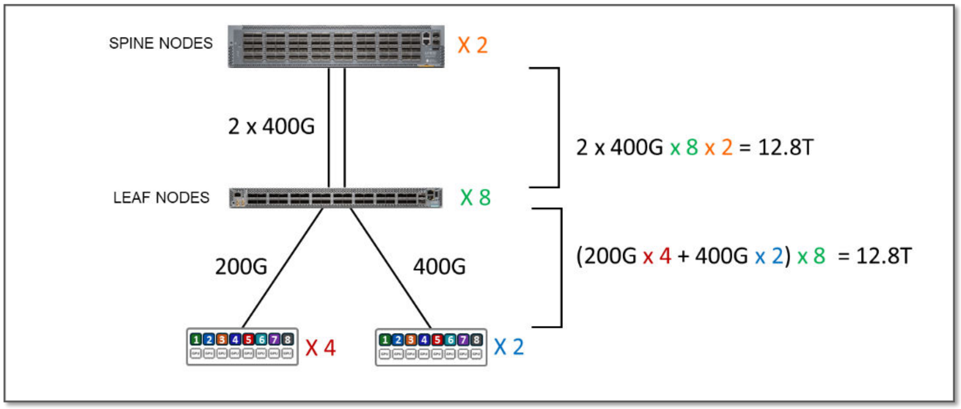


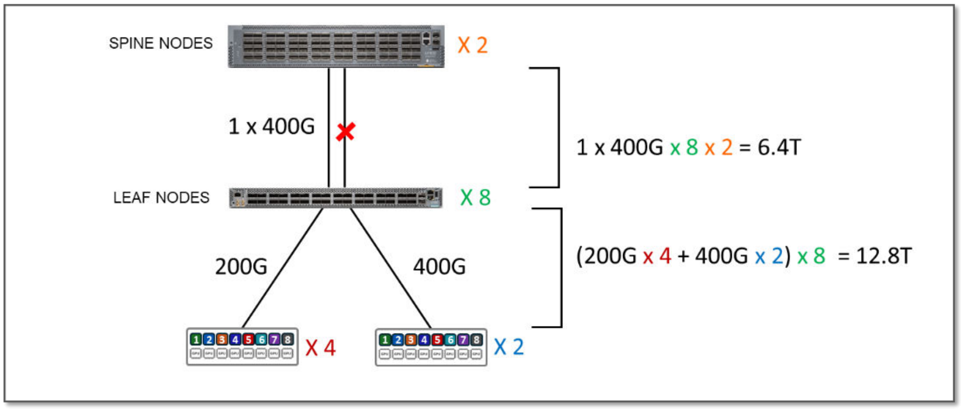
Table 7: Per cluster, per stripe Server to Leaf Bandwidth with all servers connected to same cluster

Server to Leaf Bandwidth per Stripe					
Cluster	AI Systems	Servers per Stripe	Server <=> Leaf Links per Server	Server <=> Leaf Links Bandwidth [Gbps]	Total Servers <=> Leaf Links Bandwidth per stripe [Tbps]
1	A100	4	8	200	4 x 8 x 200/1000 = 6.4
	H100	2	8	400	2 x 8 x 400/1000 = 6.4
				Total Bandwidth of Server <=> Leaf Links	12.8

The bandwidth between the servers and the leaf nodes is now 12.8 Tbps per stripe, while the bandwidth available between the leaf and spine nodes is also 12.8 Tbps per stripe (as shown in table above). This means that the fabric has enough capacity to process all traffic between the GPUs even when this traffic was 100% inter-stripe, but now there is no extra capacity to accommodate additional servers. The subscription factor in this case is 1:1 (no subscription).

To run oversubscription testing, we disabled some of the interfaces between the leaf and spines to reduce the available bandwidth as shown in the example in Figure 8:

Figure 8: 2:1 Oversubscription Example



The total Servers to Leaf Links bandwidth per stripe has not changed. It is still 12.8 Tbps as shown in table 3 in the previous scenario.

However, the bandwidth available between the leaf and spine nodes is now only 6.4 Tbps per stripe.

Table 8: Per Stripe Leaf to Spine Bandwidth

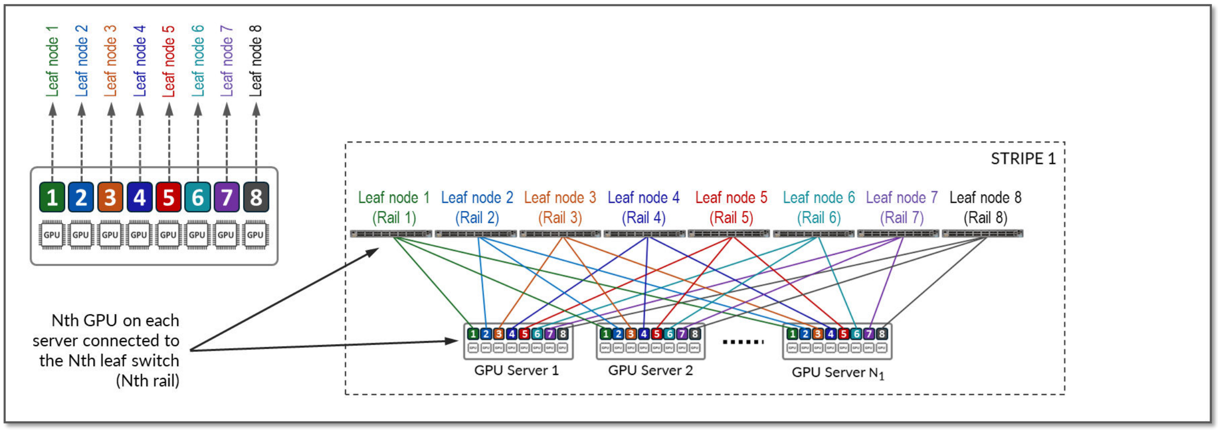
Leaf to Spine Bandwidth per Stripe			
Leaf <=> Spine Links Per Spine Node & Per Stripe	Speed Of Leaf <=> Spine Links [Gbps]	Number of Spine Nodes	Total Bandwidth Leaf <=> Spine Per Stripe [Tbps]
8	1 x 400	2	6.4

This means that the fabric no longer has enough capacity to process all traffic between the GPUs even if this traffic was 100% inter-stripe, potentially causing congestion and traffic loss. The oversubscription factor in this case is 2:1.

Rail Optimized Fabric

The GPUs on each server are numbered 1-8, where the number represents the GPU's position in the server, as shown in Figure 11.

Figure 11: Rail Optimized Connections Between GPUs and Leaf Nodes



Communication between GPUs in the same server happens internally via high throughput NV-Links (Nvidia links) channels attached to internal NV-Switches, while communication between GPUs in different servers happens across the QFX fabric, which provides 400Gbps GPU-to-GPU bandwidth. Communication across the fabric occurs between GPUs on the same rail, which is the basis of the Rail-optimized architecture: **Rails** connect GPUs of the same order across one of the leaf nodes; that is, rail N connects GPUs in position N in all the servers across leaf switch N.

Figure 12 represents a topology with one **stripe** and 8 **rails** connecting GPUs 1-8 across leaf switches 1-8 respectively.

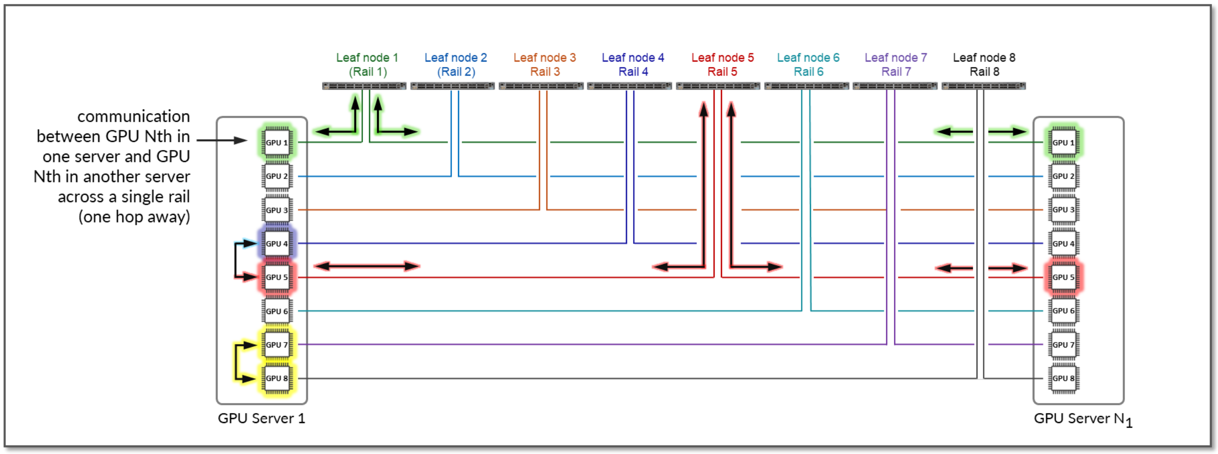
The example shows that communication between GPU 7 and GPU 8 in Server 1 happens internally across Nvidia's NVlinks/NV-switch (not shown), while communication between GPU 1 in Server 1 and GPU 1 in Server N1 happens across Leaf switch 1 (within the same rail).

Notice that if any communication between GPUs in different stripes and different servers is required (e.g. GPU 4 in server 1 communicating with GPU 5 in Server N1), data is first moved to a GPU interface in the same rail as the destination GPU, thus sending data to the destination GPU without crossing rails.

Following this design, data between GPUs on different servers (but in the same stripe) is always moved on the same rail and across one single switch, which guarantees GPUs are 1 hop away from each other and creates separate independent high-bandwidth channels, which minimize contention and maximize performance.

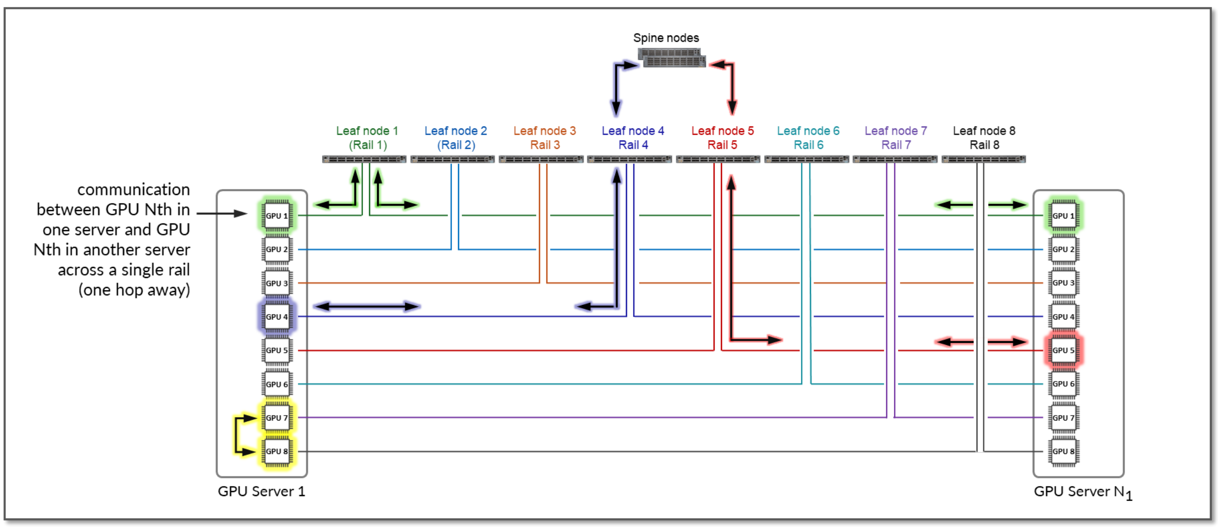
Notice that this example is presuming Nvidia's PXN feature is enabled. PXN can be easily enabled/disabled before a training or inference job in initiated.

Figure 12: GPU to GPU Communication Between Two Servers with PXN Enabled



For reference, Figure 13 shows an example with PXN disabled.

Figure 13: GPU to GPU Communication Between Two Servers Without PXN Enabled



The example shows that communication between GPU 4 in Server 1 and GPU 5 in Server N1 goes across Leaf switch 1, the Spine nodes, and Leaf switch 5 (between two different rails).

Backend GPU Rail Optimized Stripe Architecture

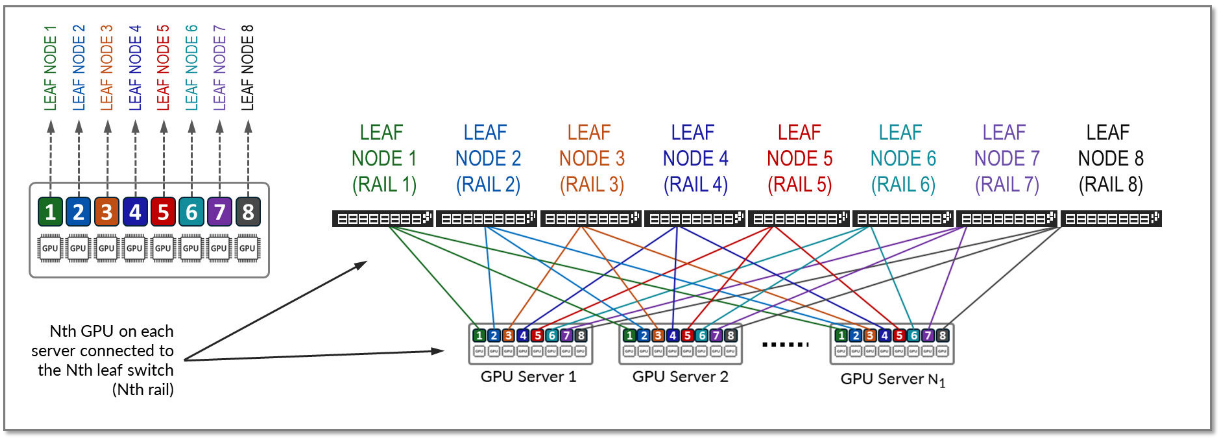
As previously described a **Rail Optimized Stripe Architecture** provides efficient data transfer between GPUs, especially during computationally intensive tasks such as AI Large Language Models (LLM) training workloads, where seamless data transfer is necessary to complete the tasks within a reasonable timeframe. A Rail Optimized topology aims to maximize performance by providing minimal bandwidth contention, minimal latency, and minimal network interference, ensuring that data can be transmitted efficiently and reliably across the network.

In a Rail Optimized Stripe Architecture there are two important concepts: **rail** and **stripe**.

The GPUs on a server are numbered 1-8, where the number represents the GPU's position in the server, as shown in Figure 6. This number is sometimes called **rank** or more specifically "**local rank**" in relationship to the GPUs in the server where the GPU sits, or "**global rank**" in relationship to all the GPUs (in multiple servers) assigned to a single job.

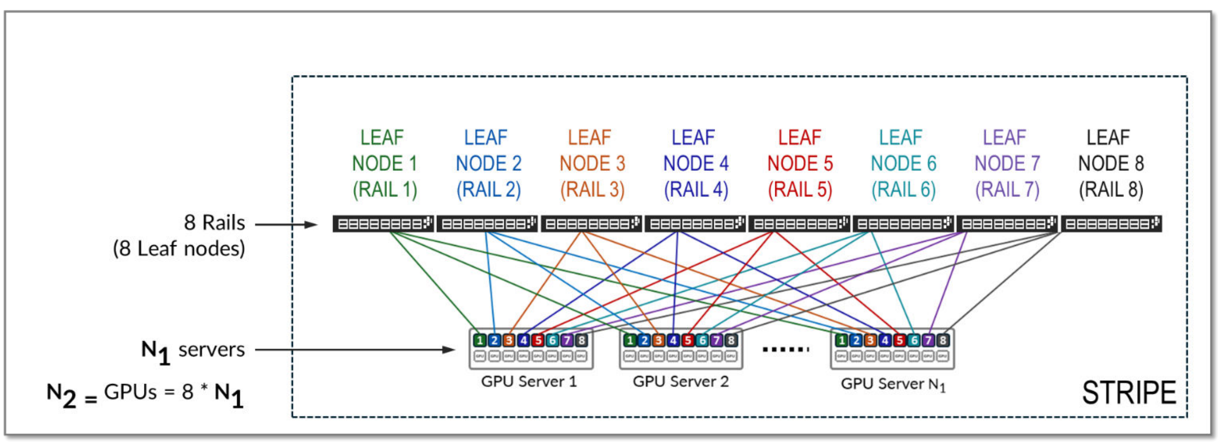
A **rail** connects GPUs of the same order across one of the leaf nodes in the fabric; that is, rail Nth connects all GPUs in position Nth on all the servers, to leaf node Nth, as shown in Figure 9.

Figure 9: Rails in a Rail Optimized Architecture



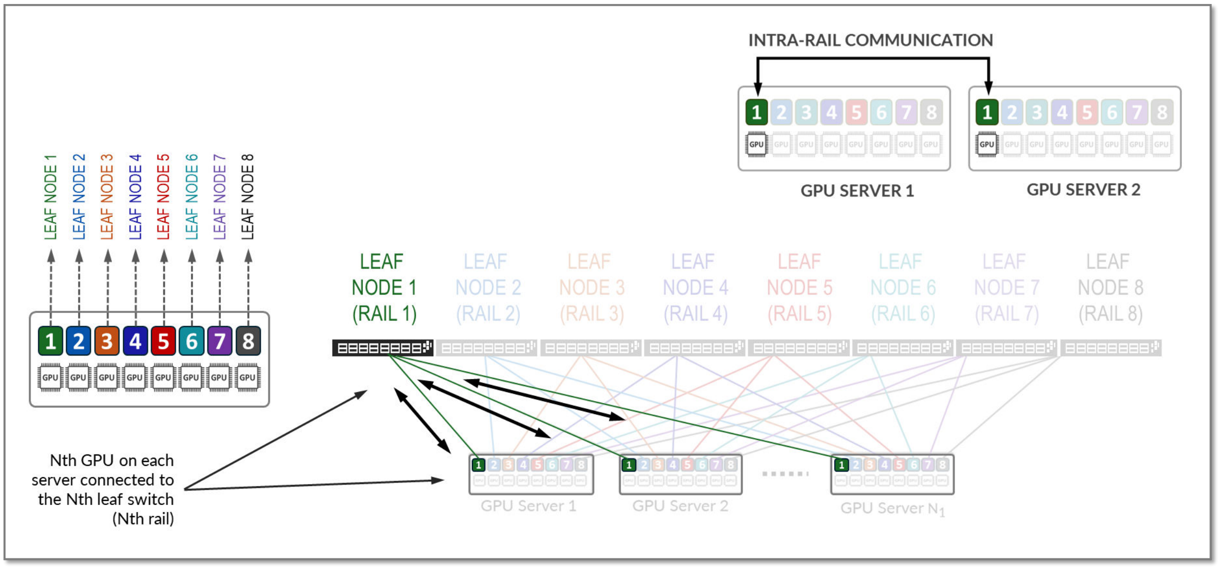
A **stripe** refers to a design module or building block, comprised of multiple **rails**, and that includes a number of Leaf nodes and GPU servers.

Figure 10: Stripes in a Rail Optimized Architecture



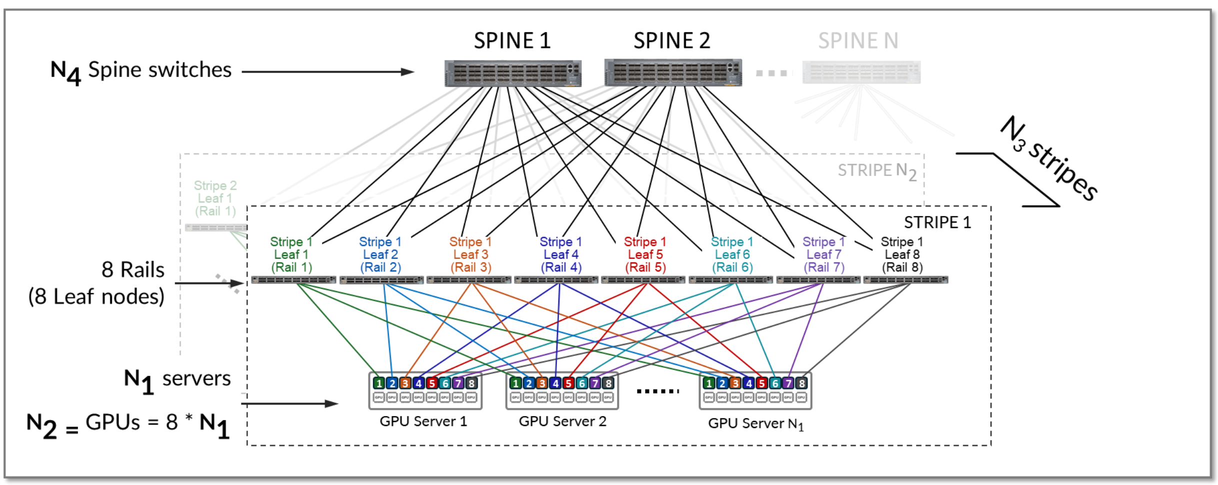
All traffic between GPUs of the same rank (intra-rail traffic) is forwarded at the leaf node level as shown in Figure 11.

Figure 11: Intra-rail GPU to GPU traffic example.



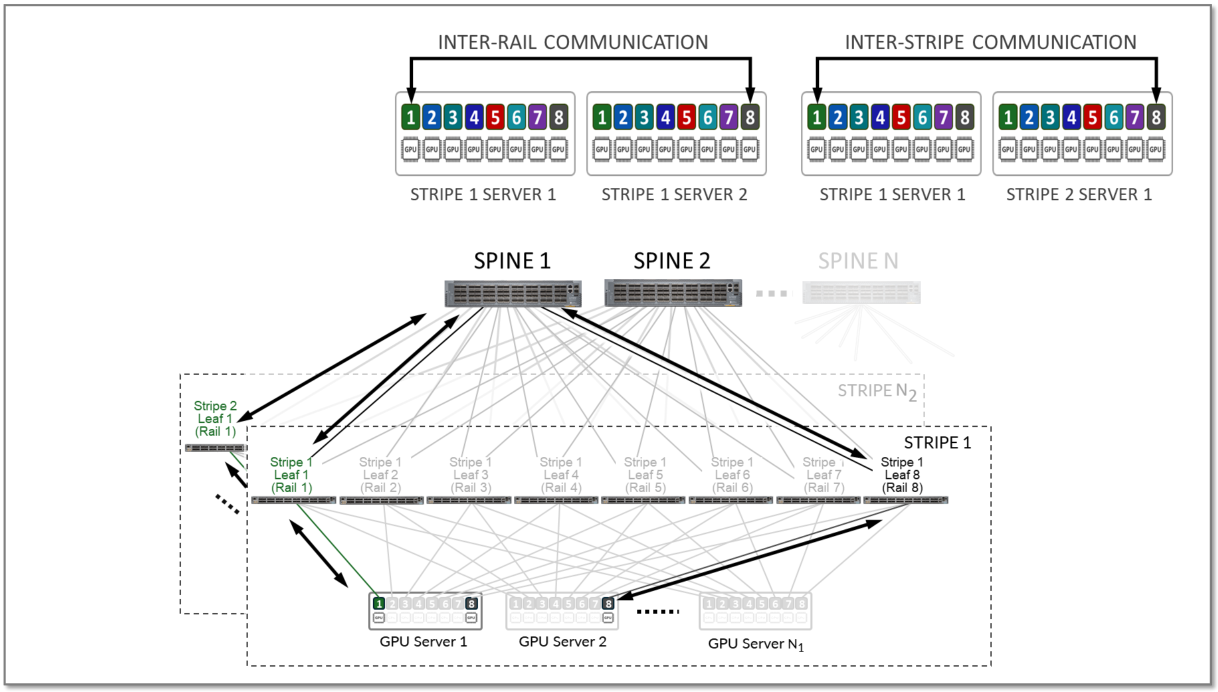
A stripe can be replicated to scale up the number of servers (N_1) and GPUs (N_2) in an AI cluster. Multiple stripes (N_3) are then connected across Spine switches as shown in Figure 12.

Figure 12: Multiple stripes connected via Spine nodes



Both Inter-rail and inter-stripe traffic will be forwarded across the spines nodes as shown in figure 13.

Figure 13. Inter-rail, and Inter-stripe GPU to GPU traffic example.



Calculating the number of leaf and spine nodes, Servers, and GPUs in a rail optimized architecture

The **number of leaf nodes** in a single stripe in a rail optimized architecture is defined by the number of GPUs per server (number of rails). Each NVIDIA DGX H100 GPU server includes 8 NVIDIA H100 Tensor core GPUs. Therefore, a single stripe includes 8 leaf nodes (8 rails).

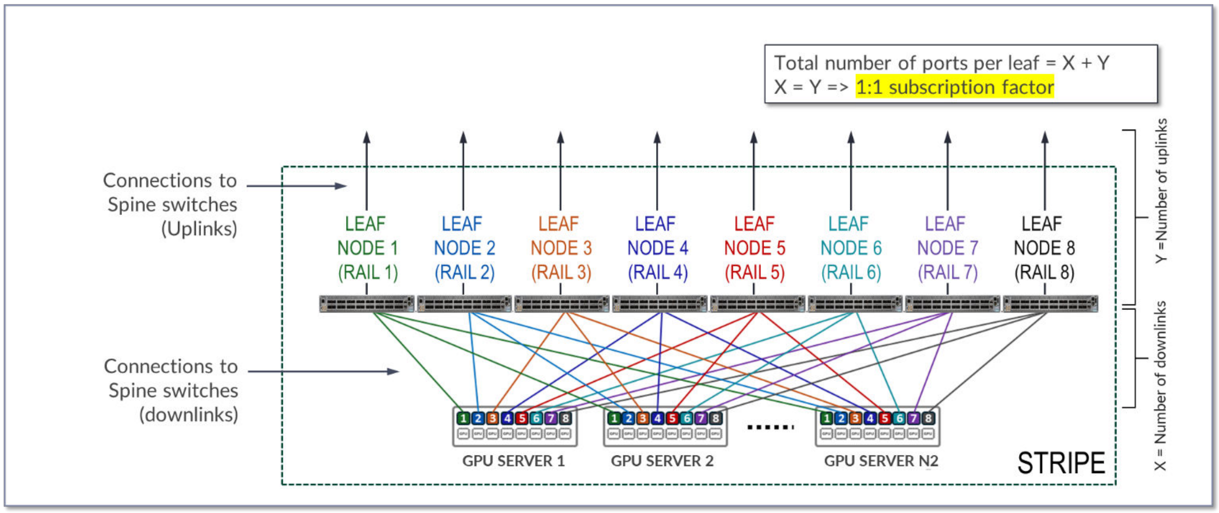
Number of leaf nodes = number of GPUs per server = 8

The **maximum number of servers** supported in a single stripe (N1) is defined by the **number of available ports** on the Leaf node which depends on the switches model.

The total bandwidth between the GPU servers and leaf nodes must match the total bandwidth between leaf and spine nodes to maintain a 1:1 subscription ratio.

Assuming all the interfaces on the leaf node operate at the same speed, half of the interfaces will be used to connect to the GPU servers, and the other half to connect to the spines. Thus, the **maximum number of servers** in a stripe is calculated as half the **number of available ports** on each leaf node. Some examples are included in Table 14.

Figure 14. Number of uplinks and downlinks for 1:1 subscription factor



I the diagram X represents the number of downlink (links between leaf nodes and the GPU servers), while Y represents the number of uplinks (links between the leaf nodes and the spine nodes). To allow for a 1:1 subscription factor, X must be equal to Y.

The **number of available ports** on each leaf node is equal to X + Y or 2 * X.

Because all servers in a stripe have one port connected to every leaf in the stripe the maximum number of servers in the stripe (N_1) is equal X.

N_1 (maximum number of servers per stripe) = number of available ports ÷ 2

The **maximum number of GPUs** in the stripe is calculated by simply multiplying by the number of GPUs per server.

N2 (maximum number of GPUs) = N1 (maximum number of servers per stripe) * 8

The **total number of available ports** is dependent on the switch model used for the leaf node. Table 9 shows some examples.

Table 9: Maximum number of GPUs supported per stripe

Leaf Node QFX switch Model	total number of available 400 GE ports per switch	**Maximum number of servers supported per stripe for 1:1 Subscription (N_1)**	**GPUs per server**	**Maximum number of GPUs supported per stripe (N_2)**
QFX5220-32CD	32	32 ÷ 2 = 16	8	16 servers x 8 GPUs/ server = 128 GPUs

(Continued)

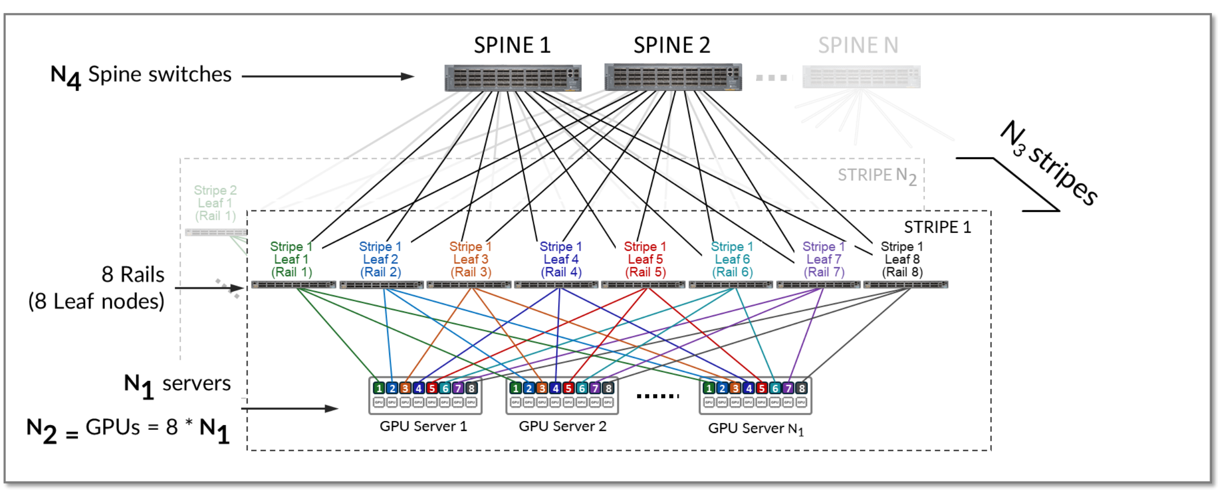
Leaf Node QFX switch Model	total number of available 400 GE ports per switch	Maximum number of servers supported per stripe for 1:1 Subscription (N_1)	GPUs per server	Maximum number of GPUs supported per stripe (N_2)
QFX5230-64CD	64	64 ÷ 2 = 32	8	32 servers x 8 GPUs/ server = 256 GPUs
QFX5240-64OD	128	128 ÷ 2 = 64	8	64 servers x 8 GPUs/ server = 512 GPUs

- QFX5220-32CD switches provide 32 x 400 GE ports (16 will be used to connect to the servers and 16 will be used to connect to the spine nodes)
- QFX5230-64CD switches provide up to 64 x 400 GE ports (32 will be used to connect to the servers and 32 will be used to connect to the spine nodes).
- QFX5240-64OD switches provide up to 128 x 400 GE ports (64 will be used to connect to the servers and 64 will be used to connect to the spine nodes).

NOTE: QFX5240-64OD switches come with 64 x 800GE ports which can break out into 2x400GE ports, for a maximum of 128 400GE interfaces was shown in table 7.

- To achieve larger scales, multiple stripes (N_3) can be connected using a set of Spine nodes (N_4), as shown in Figure 10.

Figure 10: Multiple Stripes connected across Spine nodes.



The number of stripes required is calculated based on the number of GPUs required, and the number of GPUs supported per stripe.

For example, assume that the required number of GPUs (GPUs) is 16,000 and the fabric is using QFX5240-64OD as leaf nodes.

The number of available 400G ports is 128, which means that:

- the maximum number of servers per stripe (N_1) = 64
- the maximum number of GPUs per stripe (N_2) = 512

To **number of stripes** (N_3) required is calculated by diving the number of GPUs required, and the number of GPUs per stripe as shown:

N_3 (number of stripes) = GPUs/ N_2 (maximum number of GPUs per stripe) = 16000/256 ≈ 64 stripes

- with 64 stripes & 256 servers per stripe the cluster can provide 16,384 GPUs.
- with N2 = 72 & N1 servers = 32 the cluster can provide 18432 GPUs.
- With 64 stripes & 256 servers per stripe the cluster can provide 16,384 GPUs.

Knowing the **number of stripes required** (N_3) and the **number of uplinks ports per leaf node** (**Y**) you can calculate how many spine nodes are required.

Remember X = Y = N1

First the **total number of leaf nodes** can be calculated by multiplying the **number of stripes required** by 8 (number of leaf nodes per stripe).

Total number of leaf nodes = N_3 x 8 = 64 x 8 = 512

Then the **total number of uplinks** can be obtained multiplying the **number of uplinks per leaf node (N_1),** and the **total number of leaf nodes**.

Total number of uplinks = N_1 x N_3 = 64 x 512 = 32768

The **number of spines required** (N_4) can then be determined by dividing the **total number of uplinks** by the **number of available ports on each spine node**, which as for the leaf nodes, depends on the switch model used for the spine role.

Number of spines required (N_4) = **32768 / number of available ports on each spine node**

For example, if the spine nodes are QFX5240, the **number of available ports on each spine node** is 128.

Table 8: Number of spines nodes for two stripes.

Spine Node QFX switch Model	Maximum number of 400 GE interfaces per switch	Number of spines required (N_4) with 64 stripes
QFX5240-64OD	128	32768 ÷ 128 = 256
PTX10008	288	32768 ÷ 288 ~ 128

Storage Backend Fabric

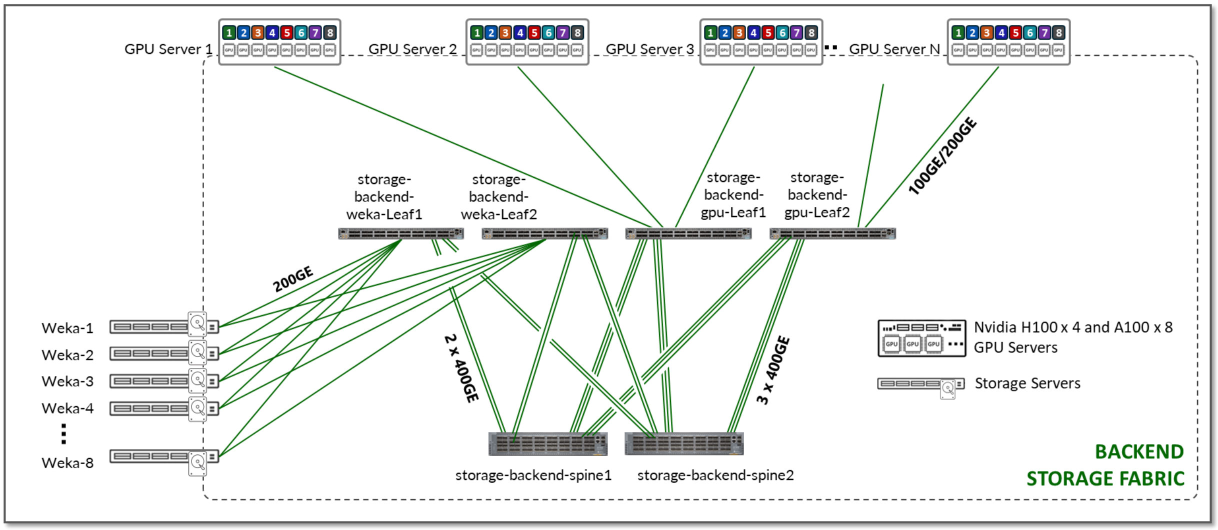
IN THIS SECTION

The **Storage Backend fabric** provides the connectivity infrastructure for storage devices to be accessible from the GPU servers.

The performance of the storage infrastructure significantly impacts the efficiency of AI workflows. A storage system that provides quick access to data can significantly reduce the amount of time for training AI models. Similarly, a storage system that supports efficient data querying and indexing can minimize the completion time of preprocessing and feature extraction in an AI workflow.

The **Storage Backend fabric** design in the JVD also follows a 3-stage IP clos architecture as shown in Figure 16. There is no concept of rail-optimization in a storage cluster. Each GPU server has a single connection to the leaf nodes, instead of 8.

Figure 16: Storage Backend Fabric Architecture



The Storage Backend devices included in this fabric, and the connections between them, are summarized in the following table:

Table 16: Storage Backend devices

Nvidia DGX GPU Servers	Weka Storage Servers	Storage Backend Leaf Nodes switch model (*storage-backend-gpu-leaf & storage-backend-weka-leaf*)	Storage Backend Spine Nodes switch model (*storage-backend-spine#)*
A100 x 8 H100 x 4	Weka storage server x 8	QFX5130-32CD x 4 (2 ***storage-backend-gpu-leaf*** nodes, and 2 ***storage-backend-weka-leaf*** nodes)	QFX5130-32CD x 2

QFX5230 and QFX5240 were also validated for the Storage Backend Leaf and Spine roles.

Table 17: Connections between servers, leaf and spine nodes in the Storage Backend

GPU Servers <=> Storage Backend GPU Leaf Nodes	Weka Storage Servers <=> Storage Backend Weka Leaf Nodes	Storage Backend Spine Nodes <=> Storage Backend Leaf nodes

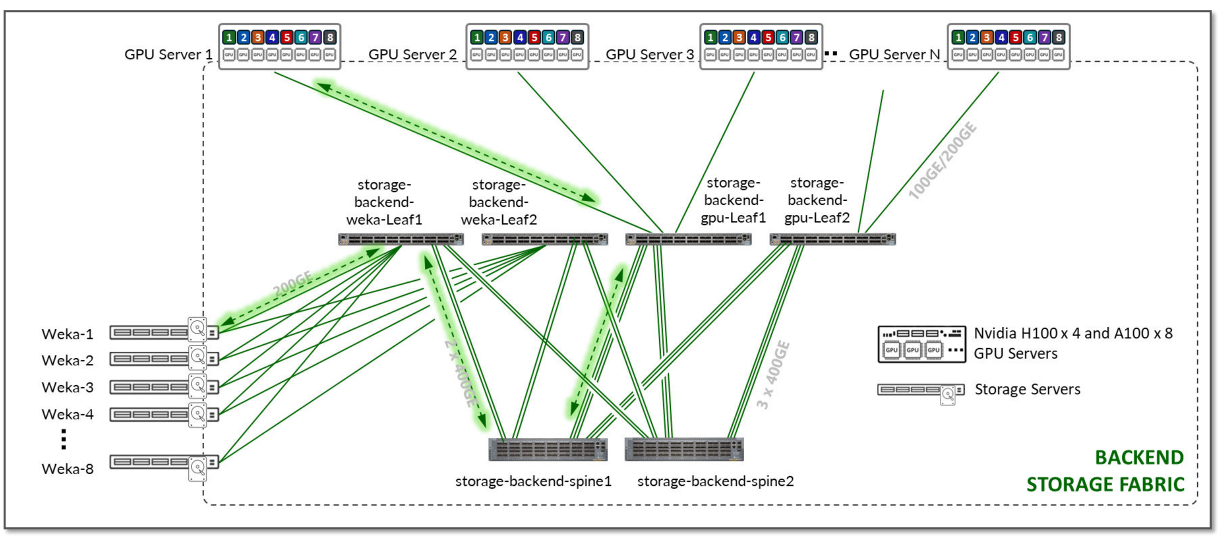
1 x 100GE links between each H100 server and the *storage-backend-gpu-leaf* switch 1 x 200GE links between each A100 server and the *storage-backend-gpu-leaf* switch	1 x 100GE links between each storage server (weka-1 to weka-8) and the *storage-backend-weka-leaf* switch	2 x 400GE links between each leaf and spine nodes and the *storage-backend-weka-leaf* switch 3 x 400GE links between each leaf and spine nodes and the *storage-backend-gpu-leaf* switch

The NVIDIA servers hosting the GPUs have dedicated storage network adapters (NVIDIA ConnectX) that support both the Ethernet and InfiniBand protocols and provide connectivity to external storage arrays.

Communications between GPUs and the storage devices leverage the WEKA distributed POSIX client which enables multiple data paths for transfer of stored data from the WEKA nodes to the GPU client servers. The WEKA client leverages the Data Plane Development Kit (DPDK) to offload TCP packet processing from the Operating System Kernel to achieve higher throughput.

This communication is supported by the Storage Backend fabric described in the previous section and exemplified in Figure 17.

Figure 17: GPU Backend to Storage Backend Communication



WEKA Storage Solution

In small clusters, it may be sufficient to use the local storage on each GPU server, or to aggregate this storage together using open-source or commercial software. In larger clusters with heavier workloads, an external dedicated storage system is required to provide dataset staging for ingest, and for cluster checkpointing during training. This JVD describes the infrastructure for dedicated storage using WEKA storage.

WEKA is a distributed data platform that allows high performance and concurrent access and allows all GPU Servers in the cluster to efficiently utilize a shared storage resource. With extreme I/O capabilities, the WEKA system can service the needs of all servers and scale to support hundreds or even thousands of GPUs.

Toward the end of this document, you can find more details on the WEKA storage system, including configuration settings, driver details, and more.

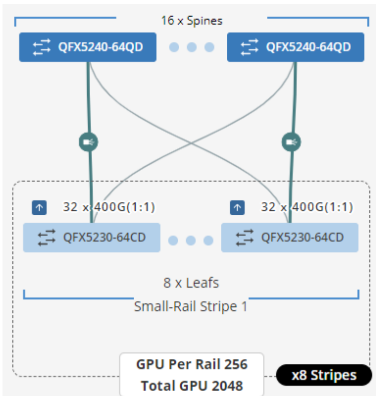
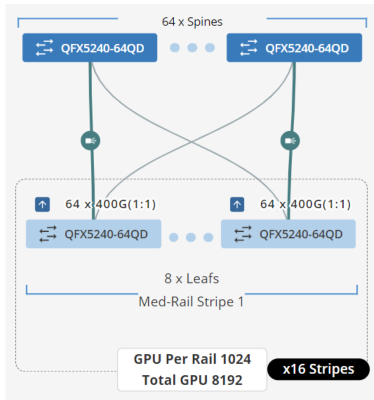
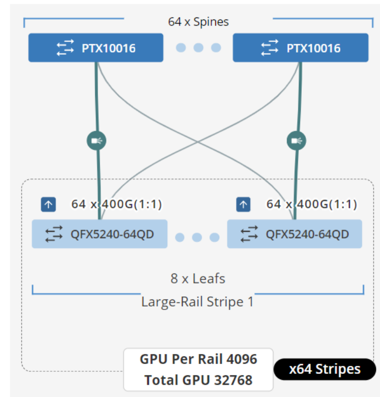
Scaling

The size of an AI cluster varies significantly depending on the specific requirements of the workload. The number of nodes in an AI cluster is influenced by factors such as the complexity of the machine learning models, the size of the datasets, the desired training speed, and the available budget. The number varies from a small cluster with less than 100 nodes to a data center-wide cluster comprising 10000s of compute, storage, and networking nodes. A minimum of 4 spines must always be deployed for path diversity and reduction of PFC failure paths.

Table 18: Fabric Scaling - Devices and Positioning

Small	Medium	Large
64 – 2048 GPU	2048 – 8192 GPU	8192 – 32768 GPU
With support for up to 2048 GPUs, the Juniper QFX5240-64CD/ QFX5241-64CD or QFX5230-64CD can be used as Spine and leaf devices to support single or dual-stripe applications. To follow best practice recommendations, a minimum of 4 Spines should be deployed, even in a single-stripe fabric.	With support for 2048 – 8192 GPUs, the Juniper QFX5240-64CD/QFX5241-64CD can be used as Spine and leaf devices to achieve appropriate scale. This 3-stage, rail-based fabric design provides physical connectivity to 16 Stripes from 64 Spines and 1024 leaf nodes, maintaining a 1:1 subscription throughput model.	For infrastructures supporting more than 8192 GPUs, the Juniper PTX1000x Chassis spine and QFX5240-64CD/QFX5241-64CD leaf nodes can support up to 32768 GPUs. This 3-stage, rail-based fabric design provides physical connectivity to 64 Stripes from 64 Spines and 4096 leaf nodes, maintaining a 1:1 subscription throughput model.

(Continued)

<table>
<tr><th>Small</th><th>Medium</th><th>Large</th></tr>
<tr>
<td>
</td>
<td>
</td>
<td>
</td>
</tr>
</table>

Juniper Hardware and Software Components

IN THIS SECTION

For this solution design, the Juniper products and software versions are below. The design documented in this JVD is considered the baseline representation for the validated solution. As part of a complete solutions suite, we routinely swap hardware devices with other models during iterative use case testing. Each switch platform validated in this document goes through the same rigorous role-based testing using specified versions of Junos OS and Apstra management software.

Juniper Hardware Components

The following table summarizes the validated Juniper devices for this JVD, and includes devices tested for AI Data Center Network with Juniper Apstra, AMD GPUs, and Vast Storage—Juniper Validated Design (JVD)

Table 19: Validated Devices and Positioning

Solution	Leaf Switches	Spine Switches
Frontend Fabric	QFX5130-32CD	QFX5130-32CD
GPU Backend Fabric	QFX5230-64CD (CLUSTER 1-STRIPE 1) QFX5220-32CD (CLUSTER 1-STRIPE 2) QFX5240-64CD/QFX5241-64CD (CLUSTER 2)	QFX5230-64CD (CLUSTER 1) PTX10008 JNP10K-LC1201 (CLUSTER 1) QFX5240-64CD/QFX5241-64CD (CLUSTER 2)
Storage Backend Fabric	QFX5220-32CD QFX5230-64CD QFX5240-64CD/QFX5241-64CD	QFX5220-32CD QFX5230-64CD QFX5240-64CD/QFX5241-64CD

Juniper Software Components

The following table summarizes the software versions tested and validated by role.

Table 20: Platform Recommended Release

Platform	Role	Junos OS Release
QFX5240-64CD	GPU Backend Leaf	23.4X100-D20
QFX5241-64CD	GPU Backend Spine	23.4X100-D42
QFX5220-32CD	GPU Backend Leaf	23.4X100-D20
QFX5230-64CD	GPU Backend Leaf	23.4X100-D20
QFX5240-64CD	GPU Backend Spine	23.4X100-D20
QFX5241-64CD	GPU Backend Spine	23.4X100-D42
QFX5230-64CD	GPU Backend Spine	23.4X100-D20
PTX10008 with LC1201	GPU Backend Spine	23.4R2-S3

(Continued)

Platform	Role	Junos OS Release
QFX5130-32CD	Frontend Leaf	23.43R2-S3
QFX5130-32CD	Frontend Spine	23.43R2-S3
QFX5220-32CD	Storage Backend Leaf	23.4X100-D20
QFX5230-64CD	Storage Backend Leaf	23.4X100-D20
QFX5240-64CD	Storage Backend Leaf	23.4X100-D20
QFX5241-64CD	Storage Backend Leaf	23.4X100-D42
QFX5220-32CD	Storage Backend Spine	23.4X100-D20
QFX5230-64CD	Storage Backend Spine	23.4X100-D20
QFX5240-64CD	Storage Backend Spine	23.4X100-D20
QFX5241-64CD	Storage Backend Spine	23.4X100-D42

IP Services for AI Networks

IN THIS SECTION

Congestion Management

AI clusters pose unique demands on network infrastructure due to their high-density, and low-entropy traffic patterns, characterized by frequent elephant flows with minimal flow variation. Additionally, most AI modes require uninterrupted packet flow with no packet loss for training jobs to be completed.

For these reasons, when designing a network infrastructure for AI traffic flows, the key objectives include maximum throughput, minimal latency, and minimal network interference over a lossless fabric, resulting in the need to configure effective congestion control methods.

Data Center Quantized Congestion Notification (DCQCN), has become the industry-standard for end-to-end congestion control for RDMA over Converged Ethernet (RoCEv2) traffic. DCQCN congestion control methods offer techniques to strike a balance between reducing traffic rates and stopping traffic all together to alleviate congestion, without resorting to packet drops.

DCQCN combines two different mechanisms for flow and congestion control:

- Priority-Based Flow Control (PFC), and
- Explicit Congestion Notification (ECN).

Priority-Based Flow Control (PFC) helps relieve congestion by halting traffic flow for individual traffic priorities (IEEE 802.1p or DSCP markings) mapped to specific queues or ports. The goal of PFC is to stop a neighbor from sending traffic for an amount of time (PAUSE time), or until the congestion clears. This process consists of sending **PAUSE control frames** upstream requesting the sender to halt transmission of all traffic for a specific class or priority while congestion is ongoing. The sender completely stops sending traffic to the requesting device for the specific priority.

While PFC mitigates data loss and allows the receiver to catch up processing packets already in the queue, it impacts performance of applications using the assigned queues during the congestion period. Additionally, resuming traffic transmission post-congestion often triggers a surge, potentially exacerbating or reinstating the congestion scenario.

We recommend configuring PFC only on the QFX devices acting as spine nodes.

Explicit Congestion Notification (ECN), on the other hand, curtails transmit rates during congestion while enabling traffic to persist, albeit at reduced rates, until congestion subsides. The goal of ECN is to reduce packet loss and delay by making the traffic source decrease the transmission rate until the congestion clears. This process entails marking packets with ECN bits at congestion points by setting the ECN bits to 11 in the IP header. The presence of this ECN marking prompts receivers to generate Congestion Notification Packets (CNPs) sent back to source, which signal the source to throttle traffic rates.

Combining PFC and ECN offers the most effective congestion relief in a lossless IP fabric supporting RoCEv2, while safeguarding against packet loss. To achieve this, when implementing PFC and ECN together, their parameters should be carefully selected so that ECN is triggered before PFC.

Load Balancing

The fabric architecture used in this JVD for both the Frontend and backend follows the 2-stage clos design, with every leaf node connected to all the available spine nodes, and via multiple interfaces. As a result, multiple paths are available between the leaf and spine nodes to reach other devices.

AI traffic characteristics may impede optimal link utilization when implementing traditional Equal Cost Multiple Path (ECMP) Static Load Balancing (SLB) over these paths. This is because the hashing algorithm which looks at specific fields in the packet headers will result in multiple flows mapped to the same link due to their similarities. Consequently, certain links will be favored, and their high utilization may impede the transmission of smaller low-bandwidth flows, leading to potential collisions, congestion and packet drops. To improve the distribution of traffic across all the available paths either Dynamic Load Balancing (DLB) or Global Load Balancing (GLB) can be implemented instead.

For this JVD Dynamic Load Balancing flowlet-mode was implemented on all the QFX leaf and spines nodes. Global Load Balancing is also included as an alternative solution.

Additional testing was conducted on the QFX5240-64OD/QFX5241-64OD in the "GPU Backend Fabric cluster 2" on page 12, to evaluate the benefits of Selective Dynamic Load Balancing, and Reactive path rebalancing. Notice that these load balancing mechanisms are only available on QFX devices.

Dynamic Load Balancing (DLB)

DLB ensures that all paths are utilized more fairly, by not only looking at the packet headers, but also considering real-time link quality based on port load (link utilization) and port queue depth, when selecting a path. This method provides better results when multiple long-lived flows moving large amounts of data need to be load balanced.

DLB can be configured in two different modes:

- **Per packet mode**: packets from the same flow are sprayed across link members of an IP ECMP group, which can cause packets to arrive out of order.

- **Flowlet Mode**: packets from the same flow are sent across a link member of an IP ECMP group. A flowlet is defined as bursts of the same flow separated by periods of inactivity. If a flow pauses for longer than the configured inactivity timer, it is possible to reevaluate the link members quality, and for the flow to be reassigned to a different.

Some enhancements have been introduced for the QFX5230s and QFX5240s in recent versions of Junos OS.

- Selective Dynamic Load Balancing (SDLB): allows implementing DLB only to certain traffic. This feature is only supported on QFX5230-64CD, QFX5240-64OD, and QFX5240-64QD, starting in Junos OS Evolved Release 23.4R2, at the time this document publication.

- Reactive path rebalancing : allows a flow to be reassigned to a different (better) link, when the current link quality deteriorates, even if no pause in the traffic flow has exceeded the configured inactivity timer. This feature is only supported on QFX5240-64OD, and QFX5240-64QD, starting in Junos OS Evolved Release 23.4R2, at the time this document publication.

Global load balancing (GLB):

GLB is an improvement on DLB which only considers the local link bandwidth utilization. GLB on the other hand, has visibility into the bandwidth utilization of links at the next-to-next-hop (NNH) level. As a result, GLB can reroute traffic flows to avoid traffic congestion farther out in the network than DLB can detect.

AI-ML data centers have less entropy and larger data flows than other networks. Because hash-based load balancing does not always effectively load-balance large data flows of traffic with less entropy, dynamic load balancing (DLB) is often used instead. However, DLB considers only the local link bandwidth utilization. For this reason, DLB can effectively mitigate traffic congestion only on the immediate next hop. GLB more effectively load-balances large data flows by taking traffic congestion on remote links into account.

GLB is only supported for QFX-5240 (TH5) starting on 23.4R2 and 24.4R1, requires a full 3-tier CLOS architecture, and is limited to only one link between each spine and leaf. When there is more than one interface or a bundle between a pair of leaf and spine, GLB won't work. Also, GLB supports 64 profiles in the table. This means there can be 64 leaves in the 3-stage Clos topology where GLB is running.

For additional details on the operation and configuration of GLB refer to Avoiding AI/ML traffic congestion with global load balancing | HPE Juniper Networking Blogs

ADDITIONAL REFERENCES:

Introduction to Congestion Control in Juniper AI Networks explores how to build a lossless fabric for AI workloads using DCQCN (ECN and PFC) congestion control methods and DLB. The document was based on DLRM training model as a reference and demonstrates how different congestion parameters such as ECN and PFC counters, input drops and tail drops can be monitored to adjust configuration and build a lossless fabric infrastructure for RoCEv2 traffic.

Load Balancing in the Data Center provides a comprehensive deep dive into the various load-balancing mechanisms and their evolution to suit the needs of the data center.

Configuration Walkthrough

IN THIS SECTION

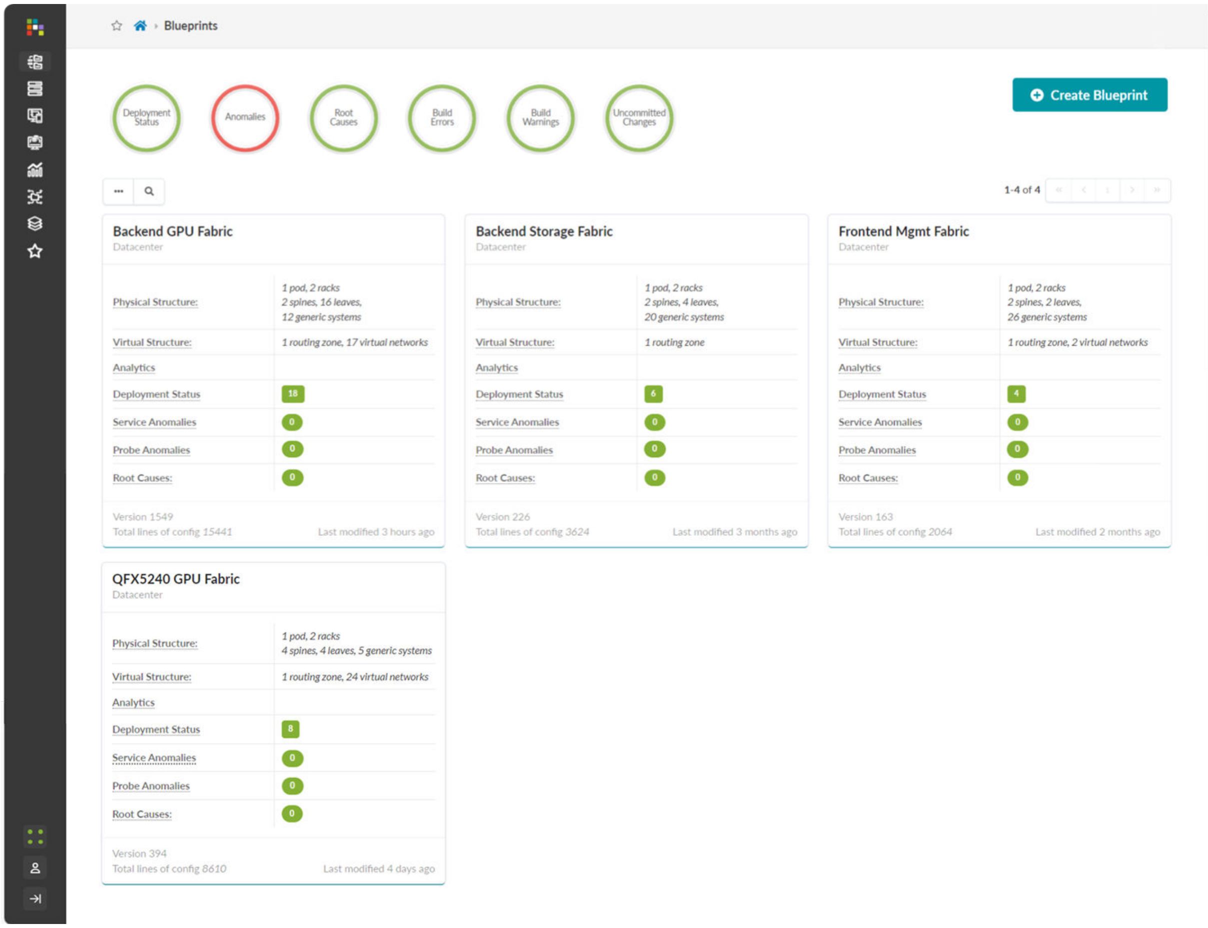
This section describes the steps to deploy one of the AI GPU Backend IP fabrics in the AI JVD lab, as an example of how to deploy new fabrics, using Juniper Apstra.

These steps will cover the AI GPU Backend IP fabric is Cluster 1 which consists of QFX5230-64CD switches in the spine role and QFX5230-64CD (stripe 1) and QFX5220-32CD (stripe 2) switches in the GPU Backend leaf role along with associated NVIDIA GPU servers and WEKA storage devices.

Similar steps should be followed to set up the Frontend and Storage Backend fabrics, as well as the AI GPU Backend IP fabric. The configurations for these are included in the Terraform repository described in the next section.

The Apstra Blueprints for all the fabrics have been created in the JVD AI lab, as shown in Figure 18.

Figure 18: AI Fabric Blueprints in Apstra



For more detailed information about installation and step-by-step configuration with Apstra, refer to the Juniper Apstra User Guide. Additional guidance in this walkthrough is provided in the form of notes.

Apstra: Configure Apstra Server and Apstra ZTP Server

A configuration wizard launches upon connecting to the Apstra server VM for the first time. At this point, passwords for the Apstra server, Apstra UI, and network configuration can be configured.

Apstra: Onboard the devices into Apstra

There are two methods for adding Juniper devices into Apstra for management: manually or in bulk using ZTP.

To add devices manually (recommended):

- In the Apstra UI navigate to Devices >> Agents >> Create Offbox Agents:
- This requires that the devices are preconfigured with a root password, a management IP and proper static routing if needed, as well as ssh Netconf, so that they can be accessed and configured by Apstra.

To add devices via ZTP:

- From the Apstra ZTP server, follow the ZTP steps described in the Juniper Apstra User Guide.

To add the QFX switches into Apstra, first log into the Apstra Web UI, choose the manual method of device addition as per above, and provide the appropriate username and password matching those preconfigured on the devices. Make sure the routers are configured accordingly.

NOTE: Apstra imports the configuration from the devices into a baseline configuration called **pristine configuration**, which is a clean, minimal, and free of any pre-existing settings that could interfere with the intended network design managed by Apstra.

Apstra ignores the Junos configuration ‘groups’ stanza and does not validate any group configuration listed in the inheritance model, refer to the configuration groups usage guide.

It is best practice to avoid setting loopbacks, interfaces (except management interface), routing-instances (except management-instance) or any other settings as part of this baseline configuration.

Apstra sets the protocols LLDP and RSTP when the device is successfully Acknowledged.

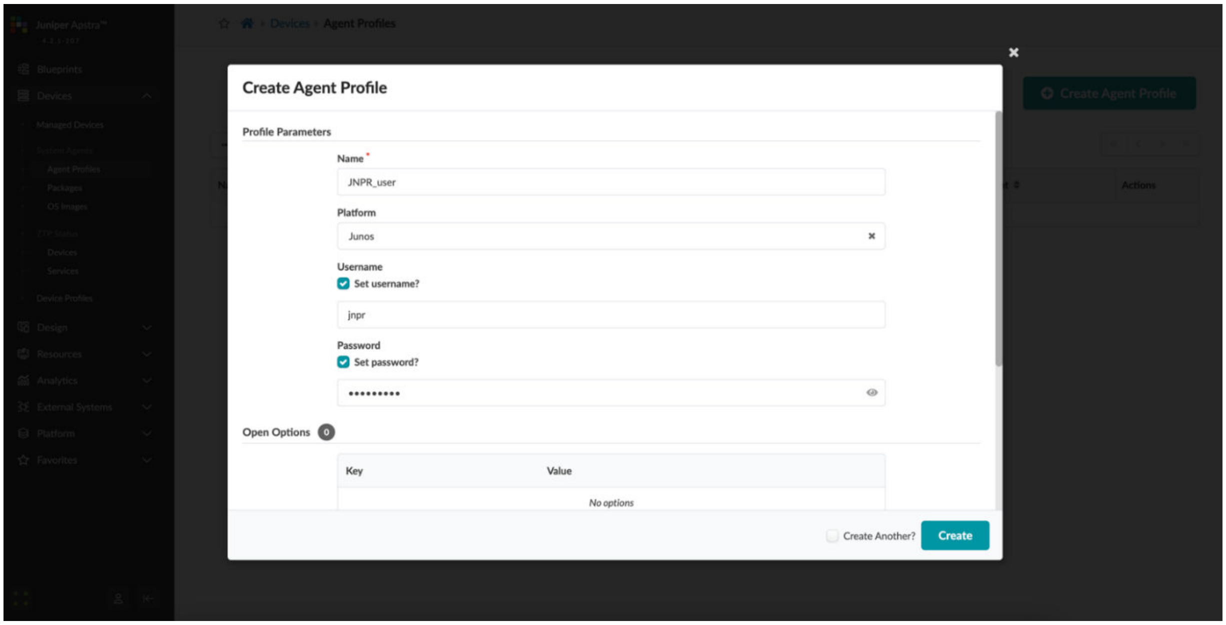
Onboarding Devices

1) Apstra Web UI: Create Agent Profile

For the purposes of this JVD, the same username and password are used across all devices. Thus, only one Apstra Agent Profile is needed to onboard all the devices, making the process more efficient.

To create an Agent Profile, navigate to **Devices** >> **Agent Profiles** and then click on **Create Agent Profile**.

Figure 19: Creating an Agent Profile in Apstra

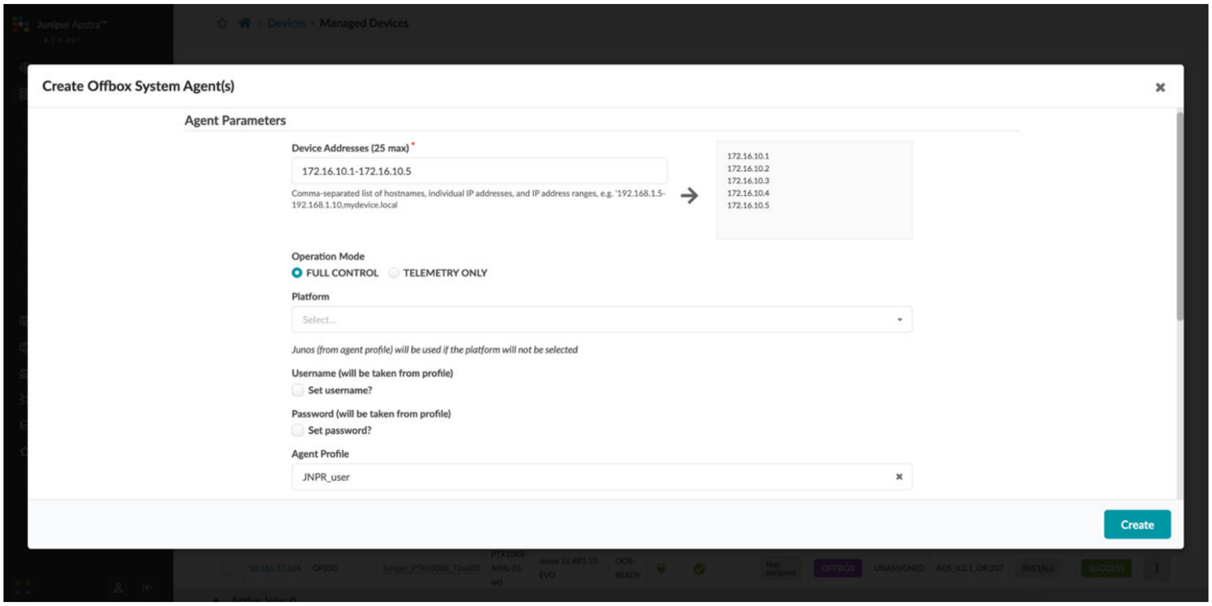


2) Apstra Web UI: Add Range of IP Addresses for Onboarding Devices

An IP address range can be provided to bulk onboard devices in Apstra. The ranges shown in the example below are shown for demonstration purposes only.

To onboard devices, navigate to **Devices** >> **Agents** and then click on **Create Offbox Agents**.

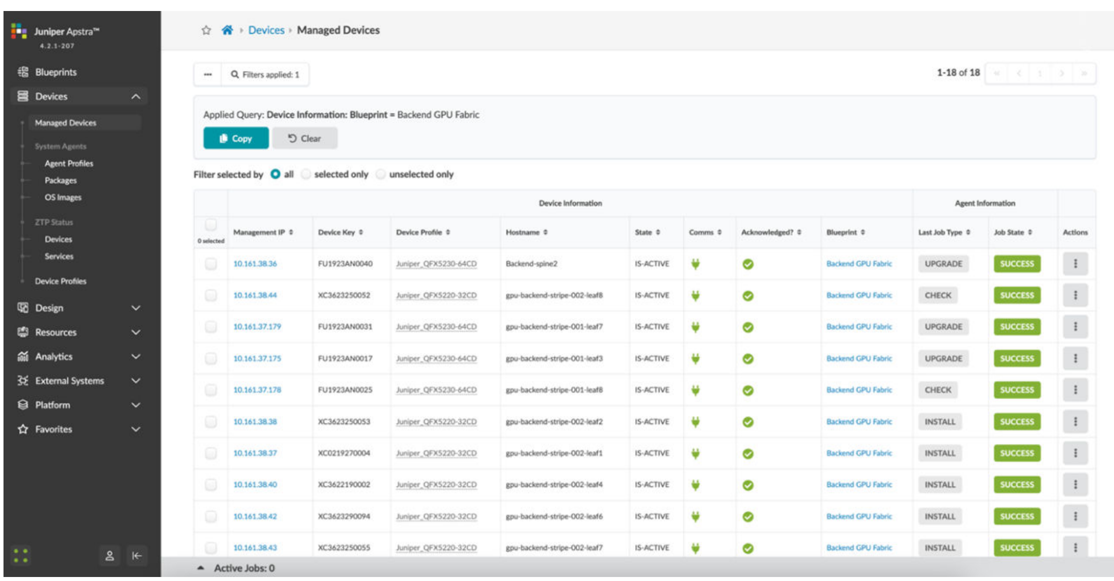
Figure 20: Adding a Range of IP Addresses in Apstra



3) Apstra Web UI: Acknowledge Managed Devices for Use in Apstra Blueprints

Once the offbox agent creation has been successfully executed for each device, the devices must be acknowledged by the user to complete the onboarding and make them part of the Apstra Blueprints. This moves the device state from OOS-QUARANTINE to OOS-READY.

Figure 21: Acknowledging Managed Devices in Apstra Blueprints



Apstra: Fabric Provisioning

The following steps outline the provisioning of the GPU Backend Fabric with Apstra.

1) Apstra Web UI: Create Logical Devices and Interface Maps with Device Profiles

The GPU Backend fabric in Apstra uses a combination of QFX5230-64CD's (stripe-1) and QFX5220-32CD's (stripe-2) for the leaf nodes and QFX5230-64CD's for the spines. Logical Devices and Interface Maps must be created for the two types of switches.

For the QFX5230-64CD leaf nodes, the Logical Device and Interface Map are shown in Figures 22 and 23:

Figure 22: Apstra Logical Device for the QFX5230 Leaf Nodes

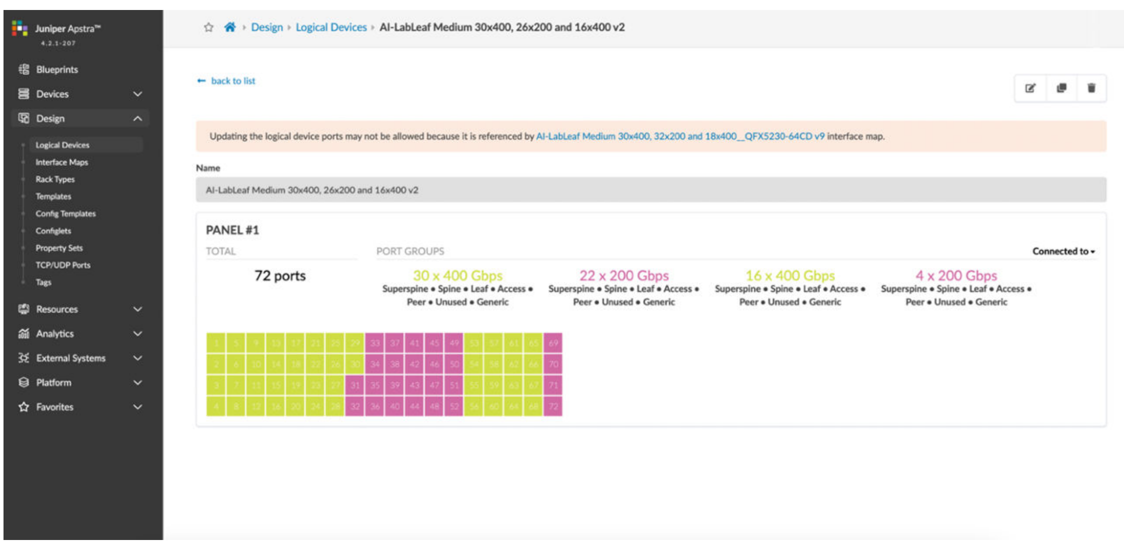
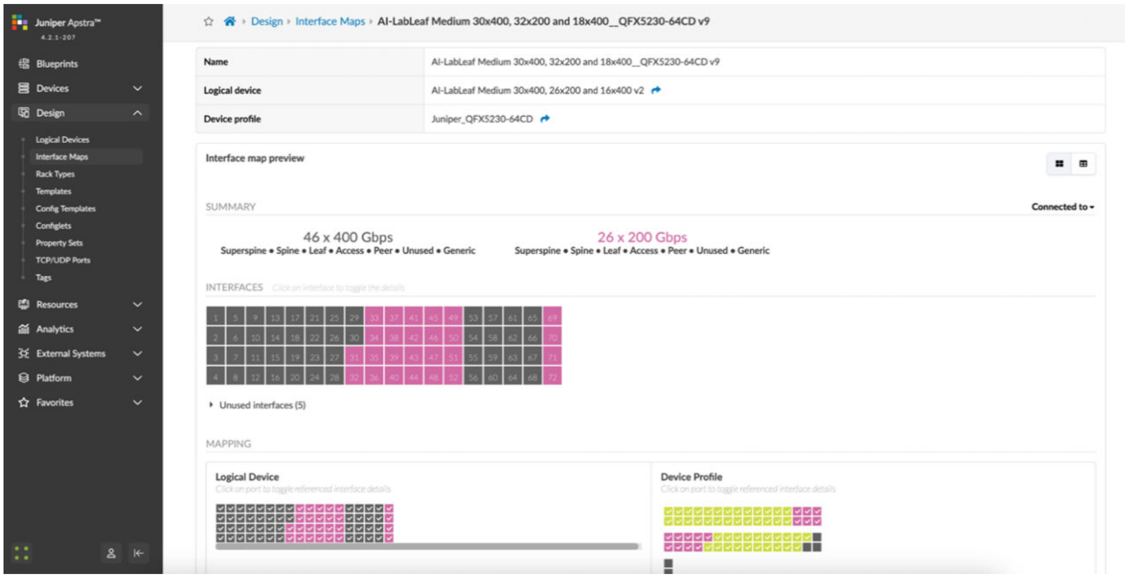


Figure 23: Apstra Interface Map for the QFX5230 Leaf Nodes



For the QFX5220 leaf nodes, the Logical Device and Interface Map are shown in Figures 24 and 25:

Figure 24: Apstra Logical Device for the QFX5220 Leaf Nodes

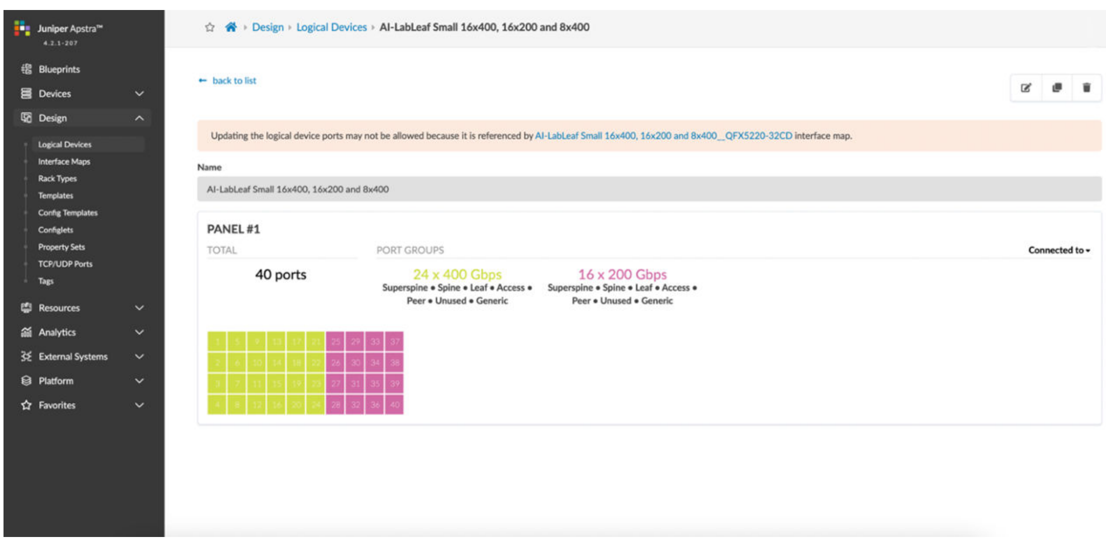
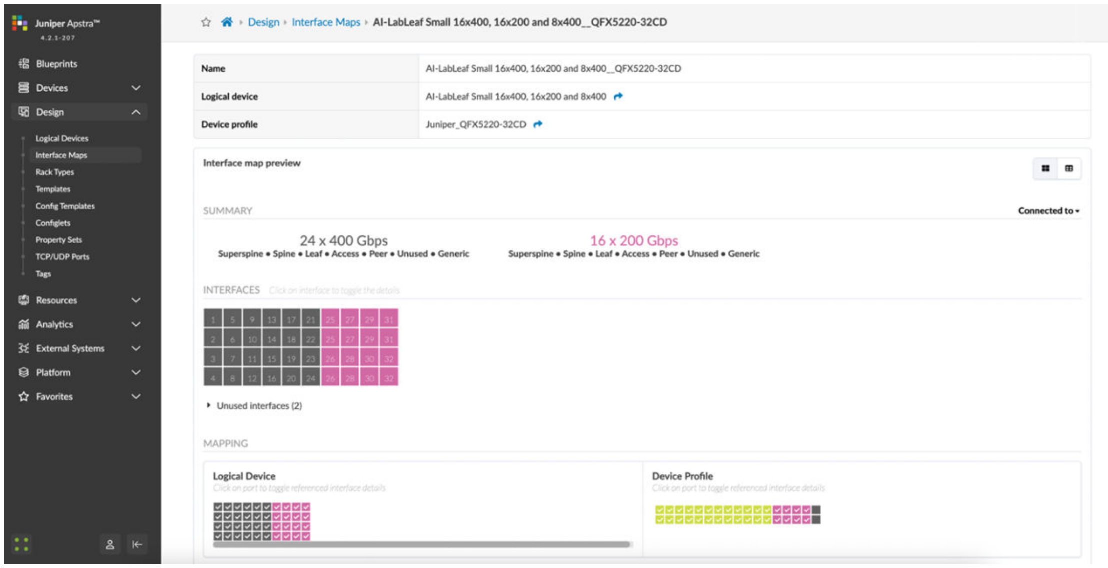


Figure 25: Apstra Interface Map for the QFX5220 Leaf Nodes



For the QFX5230 leaf nodes, the Logical Device and Interface Map are shown in Figures 26 and 27:

Figure 26: Apstra Logical Device for the QFX5230 Spine Nodes

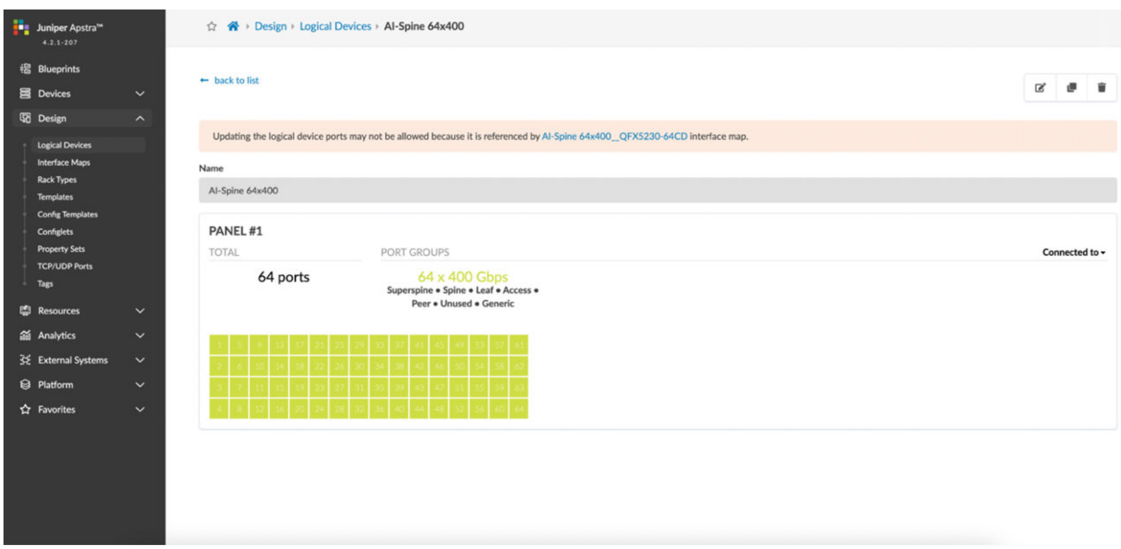
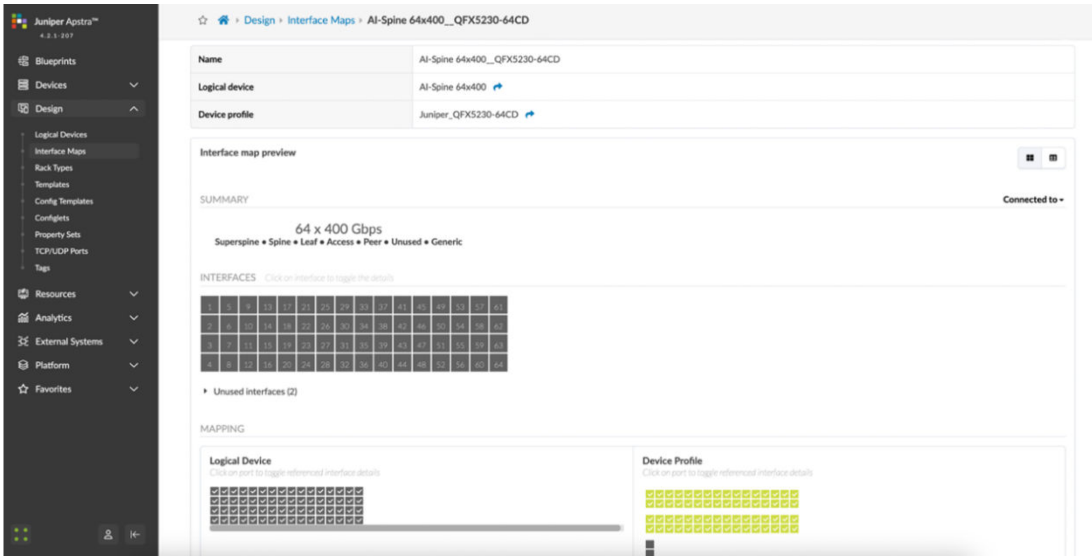


Figure 27: Apstra Interface Map for the QFX5230 Spine Nodes



For the QFX5240 spine and leaf nodes, the Logical Device and Interface Map are shown in Figures 28-29 and 30-31 respectively.

Even though the QFX5240s are not part of the fabric deployment example in this section, we are including the Logical Device and Interface Map creation for the QFX5240s to highlight the changes made to the port numbering in Junos OS Release 23.4R2, which requires completely different logical devices and interface maps.

The following table shows the differences between the old and the new port mappings.

OLD PORT MAPPING (22.2X100															
0 (8x100G)	2	4	6	8	10	12	14	16	18	20	22	24	26	28	30
1 (unused)	3	5	7	9	11	13	15	17	19	21	23	25	27	29	31
32 (8x100G)	34	36	38	40	42	44	46	48	50	52	54	56	58	60	62
33 (unused)	35	37	39	41	43	45	47	49	51	53	55	57	59	61	63
NEW PORT MAPPING (23.4R2)															
0 (8x100G)	4	8	12	16	20	24	28	32	36	40	44	48	52	56	60
1 (2x400G or 1x800G)	5	9	13	17	21	25	29	33	37	41	45	49	53	57	61
2 (8x100G)	6	10	14	18	22	26	30	34	38	42	46	50	54	58	62
3 (2x400G or 1x800G)	7	11	15	19	23	27	31	35	39	43	47	51	55	59	63

The Logical Device and Interface Map included below were created following the new port mapping.

Figure 28: Apstra Logical Device for the QFX5240 Spine Nodes

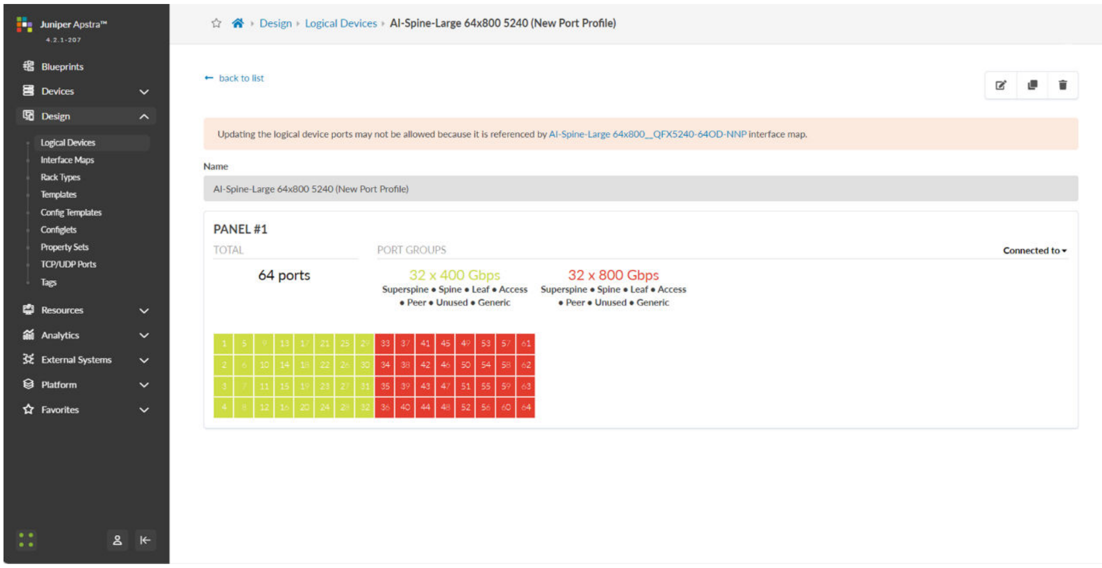


Figure 29: Apstra Interface Map for the QFX5240 Spine Nodes

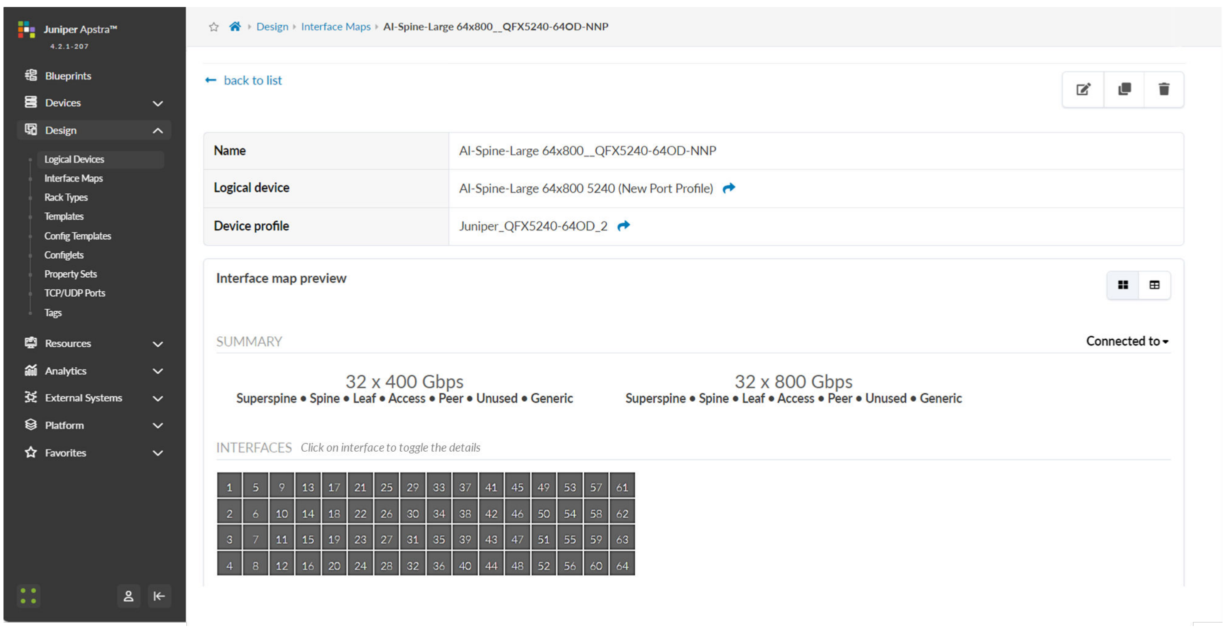


Figure 30: Apstra Interface Map for the QFX5240 Leaf Nodes

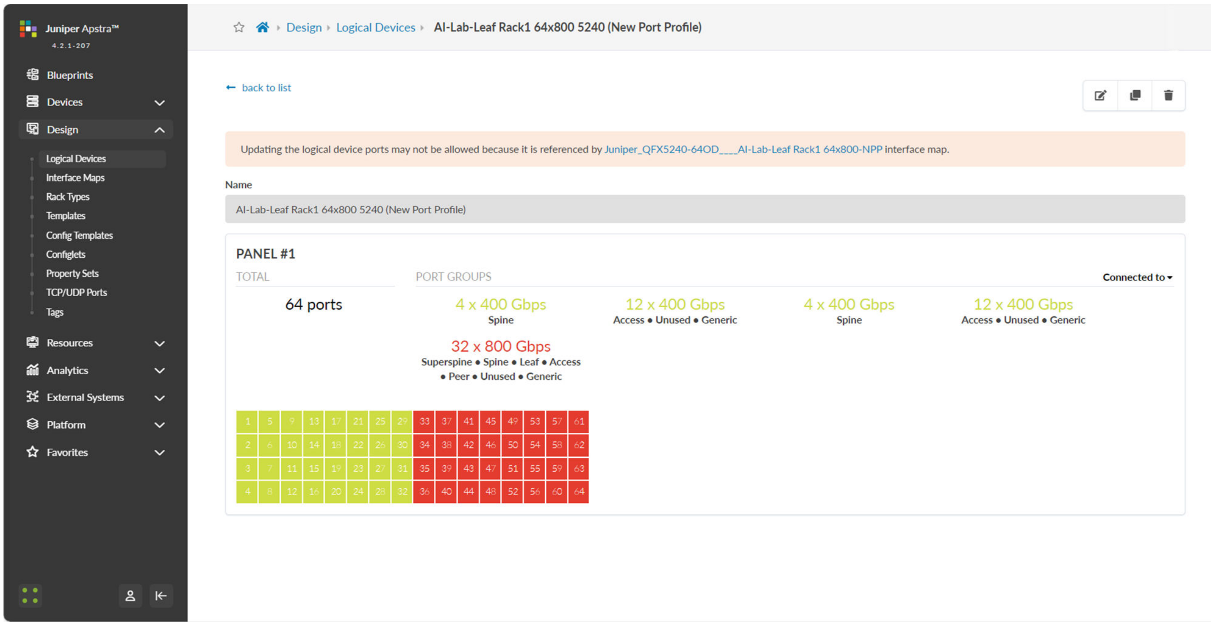
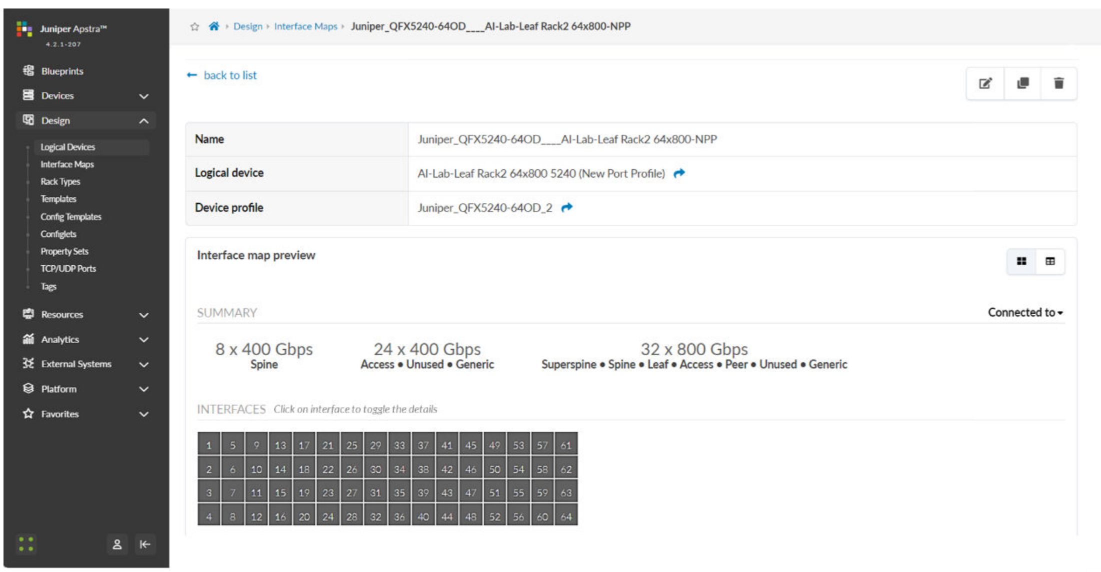


Figure 31: Apstra Logical Device for the QFX5240 Leaf Nodes



For the PTX10008 spine nodes also tested in cluster 1, the Logical Device and Interface Map are shown in Figures 32-33.

Figure 32: Apstra Interface Map for the PTX Spine Nodes

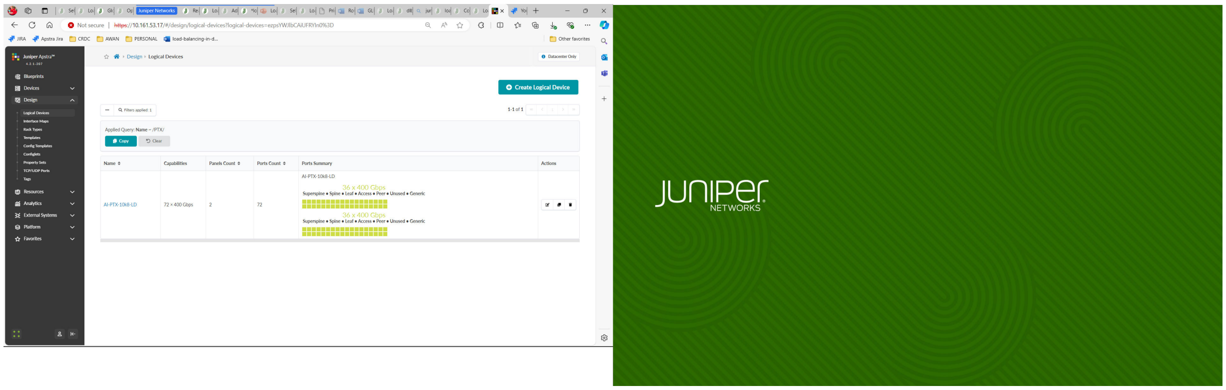
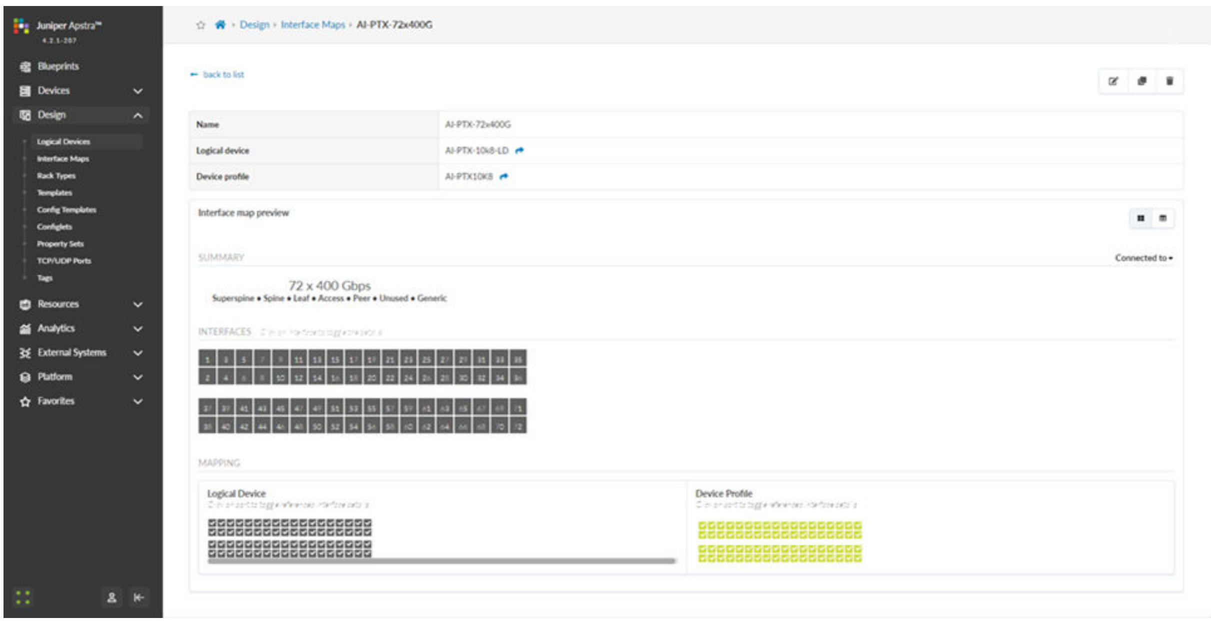


Figure 33: Apstra Logical Device for the PTX Spine Nodes



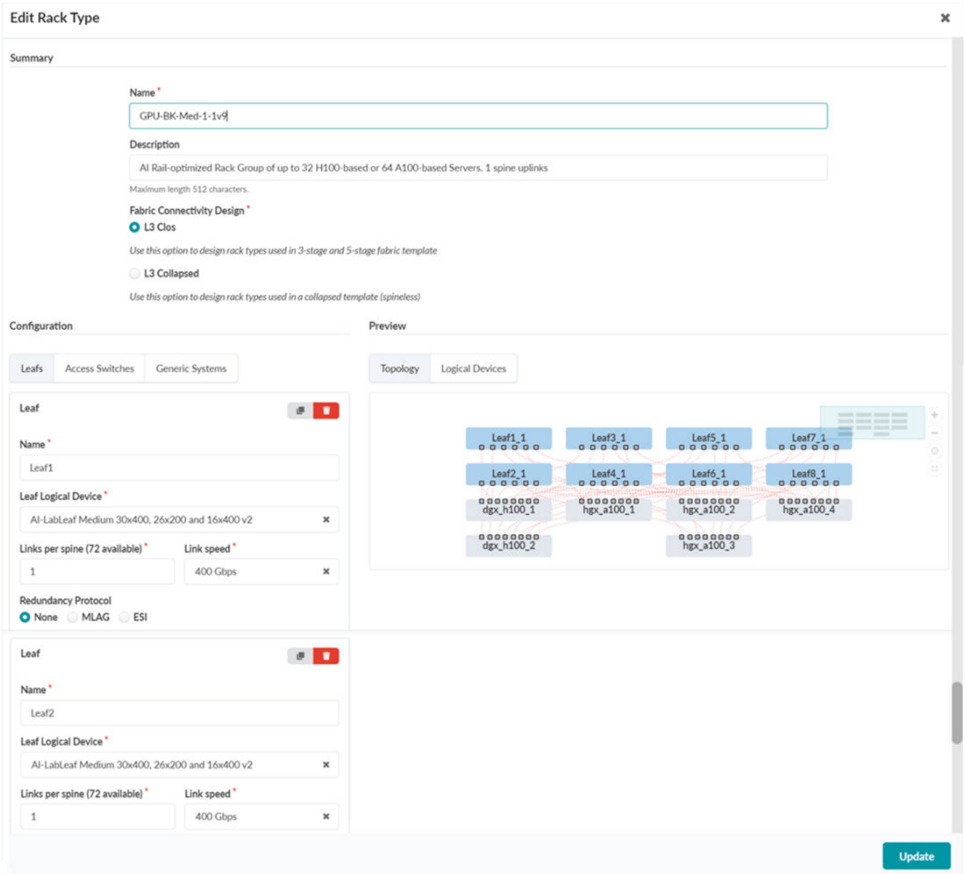
2) Apstra Web UI: Create Rack types and Template in Apstra for the GPU Backend Fabric

Once the Logical Devices and Interface Maps are created, create the necessary rack types for the GPU Backend fabric.

The design requires two rack types: one with the QFX5230 leaf nodes (stripe 1) and another with the QFX5220 leaf nodes (stripe 2).

For the sake of brevity, only the snippet of the QFX5230 rack type is shown in Figure 34.

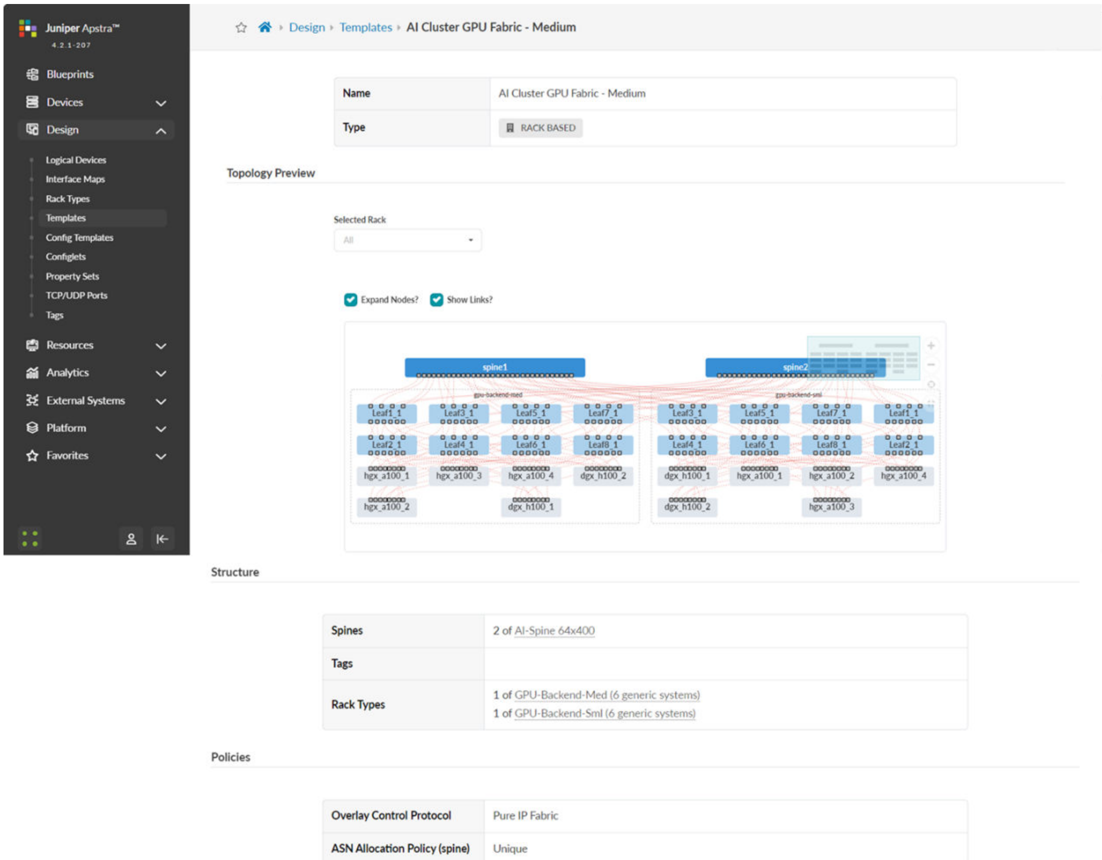
Figure 34: Creating a Rack in Apstra



Once both the racks are ready, a Template is created in Apstra by navigating to **Design** -> **Templates** -> **Create Template**.

The new Template references the QFX5230 and QFX5220 rack types created in the previous step, and is deployed as a pure IP fabric, as shown in Figure 35.

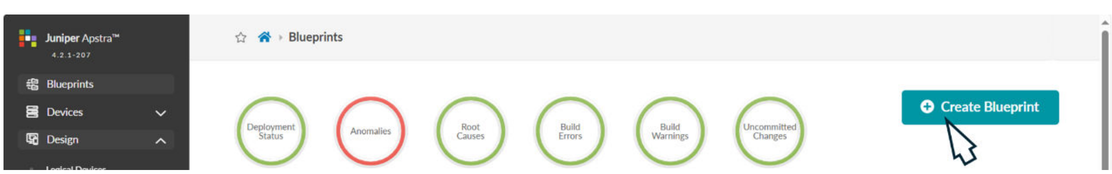
Figure 35: Creating a Template in Apstra



3) Apstra Web UI: Create a Blueprint for GPU Backend Fabric

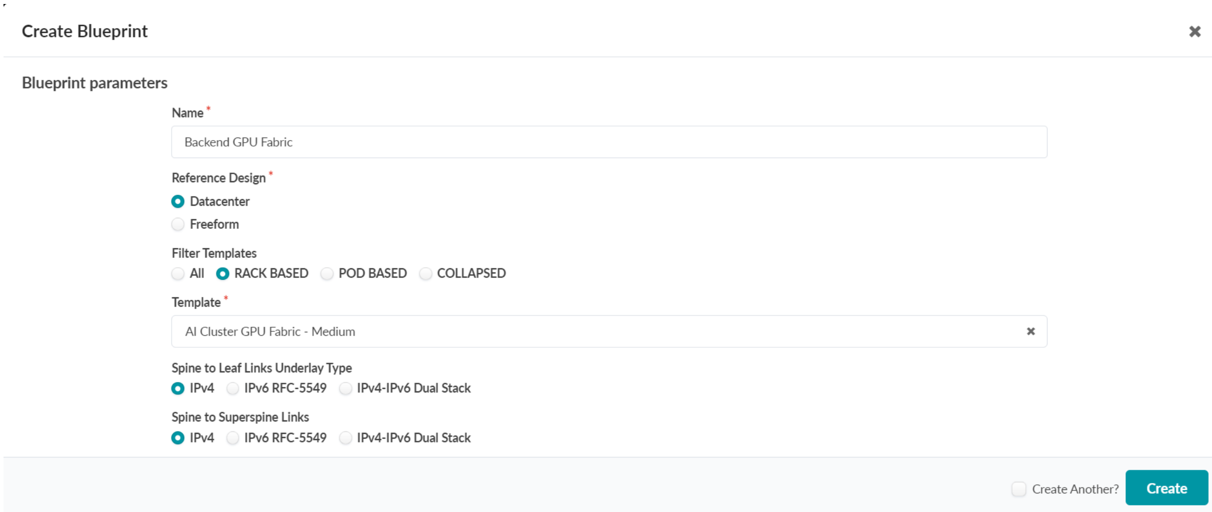
Once the Apstra Template is ready, create a Blueprint for the GPU Backend fabric by navigating to the Blueprints and clicking on Create Blueprint as shown in Figure 36.

Figure 36: Creating a Blueprint in Apstra



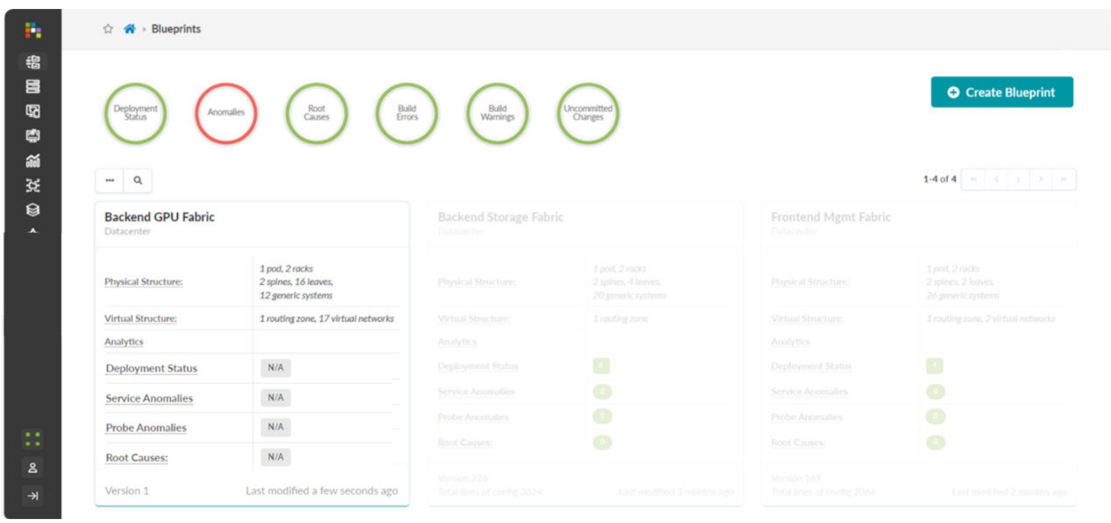
Provide a name for the new blueprint, select data center as the reference design, and select Rack-based. Then select the template that was created in the previous step which will include the two rack types that were created for the QFX5230 leaf nodes and the QFX5220 leaf nodes.

Figure 37: New Blueprint Attributes in Apstra



Once the blueprint is successfully initiated by Apstra, it will be included in the Blueprint dashboard as shown below.

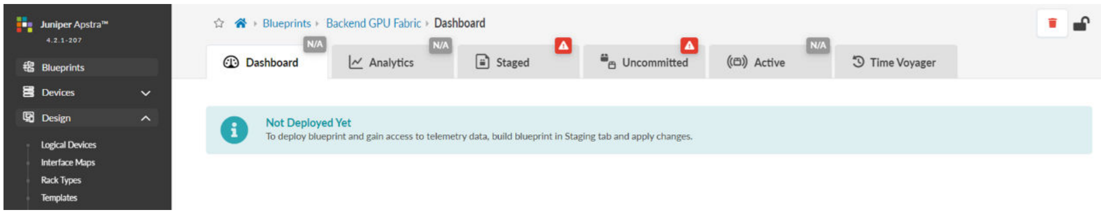
Figure 38: New Blueprint Added to Blueprint Dashboard



Notice that the Deployment Status, Service Anomalies, Probe Anomalies and Root Causes all shown as N/A. This is because you will need to complete additional steps that inlcudes mapping the different roles in the blueprint to the physical devices, defining which interfaces will be used, etc.

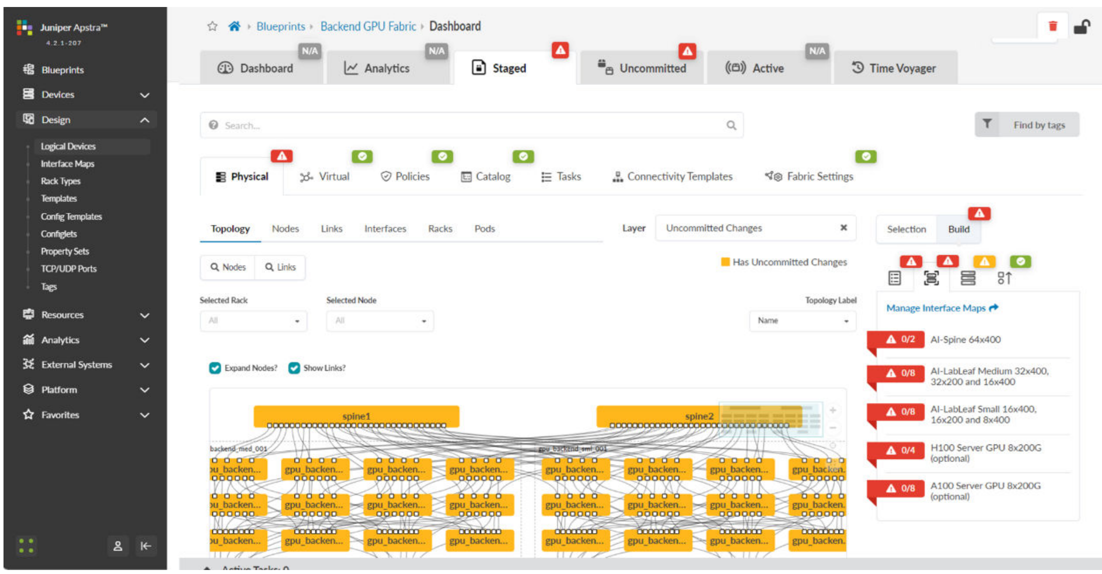
When you click on the blueprint name and enter the blueprint dashboard it will indicate that the blueprint has not been deployed yet.

Figure 39: New Blueprint's dashboard



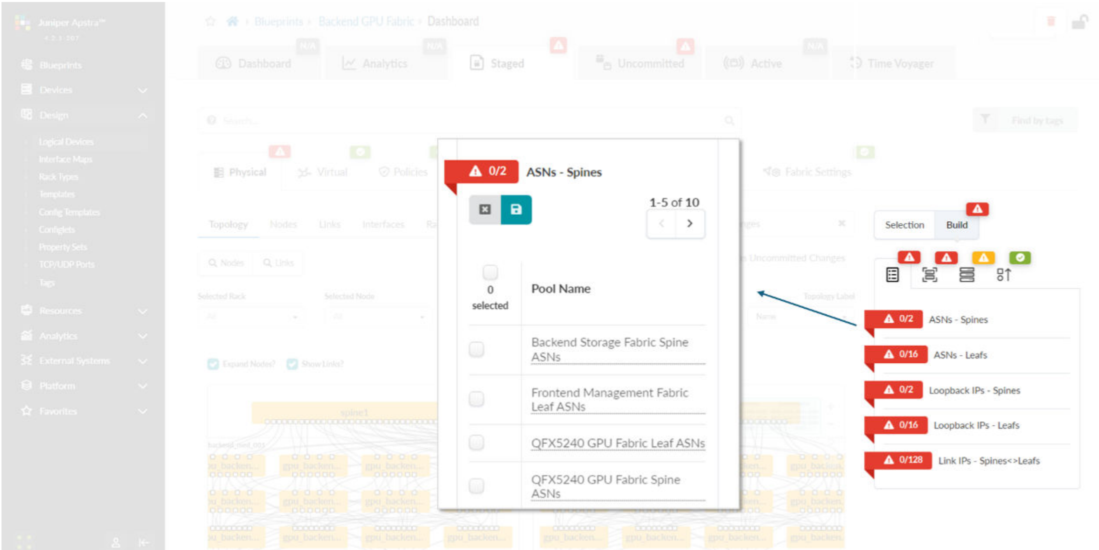
The Staged view as depicted in Figure 40 shows that the topology is correct, but attributes such as mandatory ASNs and loopback addressing for the spines and the leaf nodes, and the spine to leaf links addressing must be provided by the user.

Figure 40: Undeployed Blueprint Dashboard



You will need to edit each one of these attributes and select from predefined pools of addresses and ASNs, as shown in the example on Figure 41, to fix this issue.

Figure 41: Selecting ASN Pool for Spine Nodes



You will also need to select Interface Maps for each devices' role and along with assignment of system IDs as shown in Figures 42-43.

Figure 42: Mapping Interface Maps to Spine Nodes

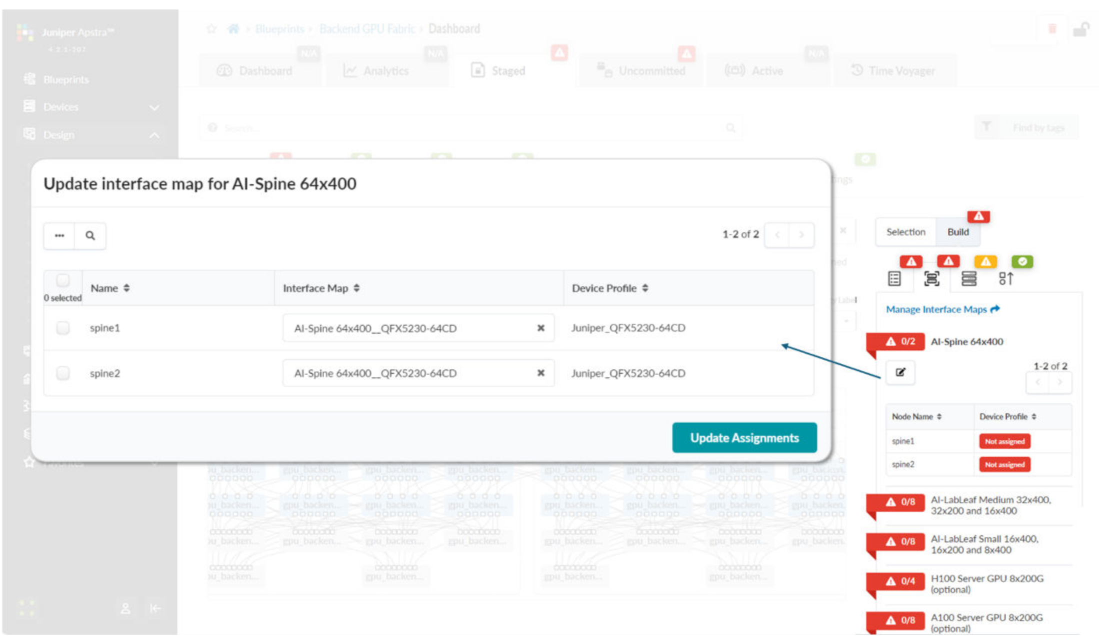
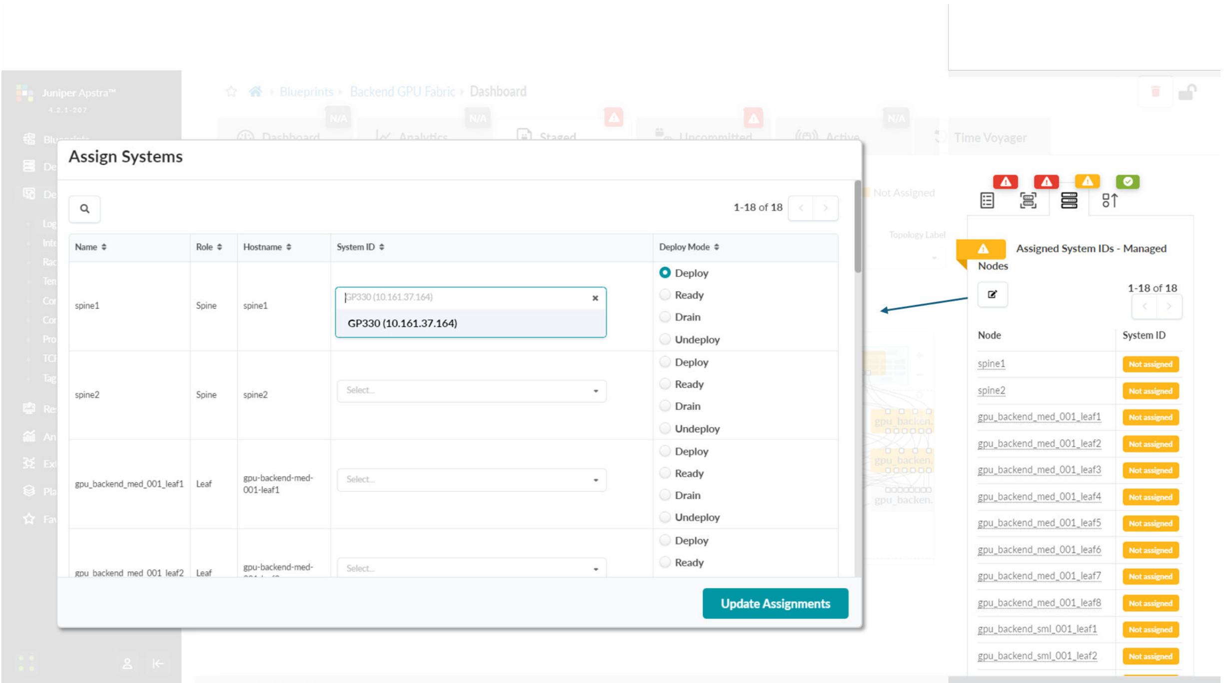
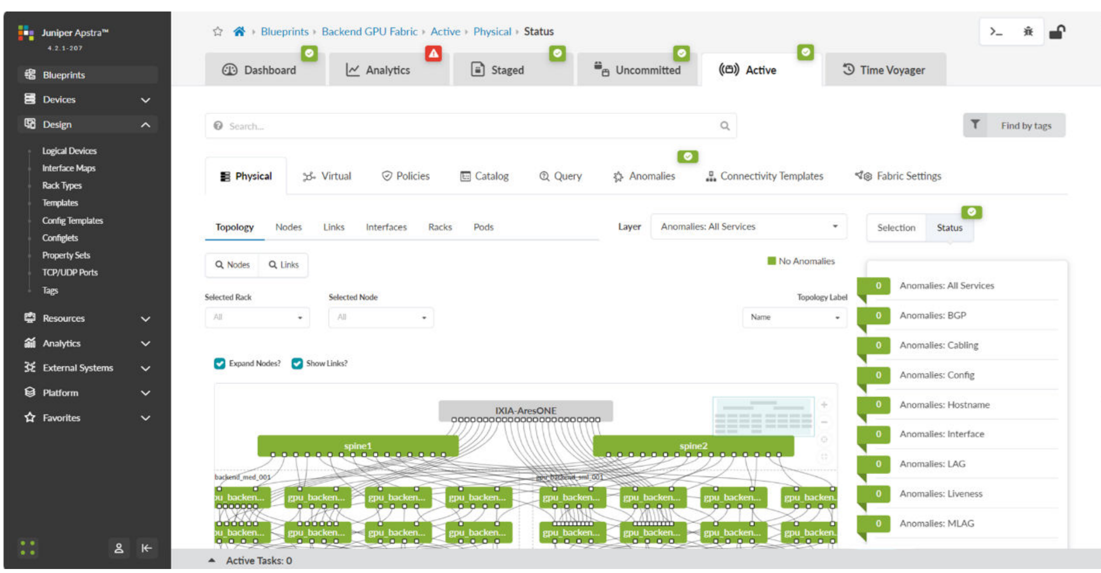


Figure 43: Mapping Spine Nodes to Physical Devices (System IDs)



Once all these steps are completed, you can commit all the changes and Apstra will generate and push all the necessary vendor-specific configuration to the nodes. Once this has been completed you should be able to view an active blueprint that represents the successfully deployed fabric as shown in Figure 44.

Figure 44: Mapping Spine Nodes to Physical Devices 2 (System IDs)



Apstra Web UI: Creating Configlets in Apstra for DCQCN and DLB

As of Apstra 4.2.1, features such as ECN and PFC (DCQCN), and DLB are not natively available. Thus, to deploy the necessary configuration to enable these features on the fabric devices, Apstra Configlets are used.

The configlet used for the DCQCN and DLB features on the QFX leaf nodes is as follows:

- ```
  /* DLB configuration */

  hash-key {

      family inet {

          layer-3;

          layer-4;

      }

  }

  enhanced-hash-key {

      ecmp-dlb {

          flowlet {

              inactivity-interval 128;

              flowset-table-size 2048;

          }

          ether-type {

              ipv4;

              ipv6;

          }
  ```
  ```

```
        sampling-rate 1000000;
    }
}
protocols {
    bgp {
        global-load-balancing {
            load-balancer-only;
        }
    }
}
/* DCQCN configuration */
class-of-service {
    classifiers {
        dscp mydscp {
            forwarding-class CNP {
                loss-priority low code-points 110000;
            }
            forwarding-class NO-LOSS {
                loss-priority low code-points 011010;
            }
        }
    }
```

```
drop-profiles {
    dp1 {
        interpolate {
            fill-level [ 55 90 ];
            drop-probability [ 0 100 ];
        }
    }
}
shared-buffer {
    ingress {
   buffer-partition lossless {
      percent 66;
      dynamic-threshold 10;
   }
   buffer-partition lossless-headroom {
      percent 24;
   }
   buffer-partition lossy {
      percent 10;
   }
  }
```

```
  egress {
    buffer-partition lossless {
       percent 66;
    }
    buffer-partition lossy {
       percent 10;
    }
  }
}
forwarding-classes {
    class CNP queue-num 3;
    class NO-LOSS queue-num 4 no-loss pfc-priority 3;
}
congestion-notification-profile {
    cnp {
        input {
            dscp {
                code-point 011010 {
                    pfc;
                }
            }
        }
```

```
        output {
            ieee-802.1 {
                code-point 011 {
                    flow-control-queue 4;
                }
            }
        }
    }
}
interfaces {
    et-* {
        congestion-notification-profile cnp;
        scheduler-map sm1;
        unit * {
            classifiers {
                dscp mydscp;
            }
        }
    }
}
scheduler-maps {
```

```
        sm1 {
            forwarding-class CNP scheduler s2-cnp;
            forwarding-class NO-LOSS scheduler s1;
        }
    }
    schedulers {
        s1 {
            drop-profile-map loss-priority any protocol any drop-profile dp1;
            explicit-congestion-notification;
        }
        s2-cnp {
            transmit-rate percent 5;
            priority strict-high;
        }
    }
}
```

The configlet used for the DCQCN and DLB features on the QFX spine nodes is as follows:

- ```
  /* DLB configuration */
  hash-key {
      family inet {
          layer-3;
  ```
  ```

```
        layer-4;
    }
}
enhanced-hash-key {
    ecmp-dlb {
        flowlet {
            inactivity-interval 128;
            flowset-table-size 2048;
        }
        ether-type {
            ipv4;
            ipv6;
        }
        sampling-rate 1000000;
    }
}
protocols {
    bgp {
        global-load-balancing {
            helper-only;
        }
    }
```

```
}
/* DCQCN configuration */
class-of-service {
    classifiers {
        dscp mydscp {
            forwarding-class CNP {
                loss-priority low code-points 110000;
            }
            forwarding-class NO-LOSS {
                loss-priority low code-points 011010;
            }
        }
    }
    drop-profiles {
        dp1 {
            interpolate {
                fill-level [ 55 90 ];
                drop-probability [ 0 100 ];
            }
        }
    }
```

```
shared-buffer {
    ingress {
   buffer-partition lossless {
      percent 66;
      dynamic-threshold 10;
   }
   buffer-partition lossless-headroom {
      percent 24;
   }
   buffer-partition lossy {
      percent 10;
   }
  }
  egress {
    buffer-partition lossless {
       percent 66;
    }
    buffer-partition lossy {
       percent 10;
    }
  }
}
```

```
forwarding-classes {
    class CNP queue-num 3;
    class NO-LOSS queue-num 4 no-loss pfc-priority 3;
}
congestion-notification-profile {
    cnp {
        input {
            dscp {
                code-point 011010 {
                    pfc;
                }
            }
        }
        output {
            ieee-802.1 {
                code-point 011 {
                    flow-control-queue 4;
                }
            }
        }
    }
```

```
}
interfaces {
    et-* {
        congestion-notification-profile cnp;
        scheduler-map sm1;
        unit * {
            classifiers {
                dscp mydscp;
            }
        }
    }
}
scheduler-maps {
    sm1 {
        forwarding-class CNP scheduler s2-cnp;
        forwarding-class NO-LOSS scheduler s1;
    }
}
schedulers {
    s1 {
        drop-profile-map loss-priority any protocol any drop-profile dp1;
        explicit-congestion-notification;
```

```
        }
        s2-cnp {
            transmit-rate percent 5;
            priority strict-high;
        }
    }
}
```

The configuration used for the DCQCN features on the PTX10008 as spine devices is as follows:

NOTE: when using PTX10008 as a spine node, GLB is not an option.

- ```
  /* DCQCN configuration */
  class-of-service {
          classifiers {
                  dscp mydscp {
                          forwarding-class rdma-cnp {
                                  loss-priority low code-points 110000;
                          }
                          forwarding-class rdma-ecn {
                                  loss-priority low code-points 011010;
                          }
                  }
          }
  ```
  ```

```
        drop-profiles {
        dp-ecn {
                fill-level 1 drop-probability 0;
                fill-level 3 drop-probability 100;
                }
        }
        forwarding-classes {
        class network-control queue-num 3;
        class other queue-num 2;
        class rdma-cnp queue-num 0;
        class rdma-ecn queue-num 1 no-loss;
        }
        monitoring-profile {
        mp1 {
                        export-filters filt1 {
                                peak-queue-length {
                                percent 0;
                        }
                        queue [ 0 1 ];
                }
        }
}
```

```
interfaces {
    et-* {
        scheduler-map sched-map-aiml;
        monitoring-profile mp1;
        unit * {
            classifiers {
                dscp mydscp;
            }
        }
    }
}
scheduler-maps {
    sched-map-aiml {
        forwarding-class network-control scheduler sched-nc;
        forwarding-class other scheduler sched-other;
        forwarding-class rdma-cnp scheduler sched-cnp;
        forwarding-class rdma-ecn scheduler sched-ecn;
    }
}
schedulers {
    sched-cnp {
```

```
        transmit-rate percent 1;

        priority high;

    }

    sched-ecn {

        transmit-rate percent 97;

        buffer-size temporal 4063;

        priority medium-high;

        drop-profile-map loss-priority any protocol any drop-profile dp-ecn;

        explicit-congestion-notification;

    }

    sched-nc {

        transmit-rate percent 1;

        priority medium-high;

    }

    sched-other {

        priority low;

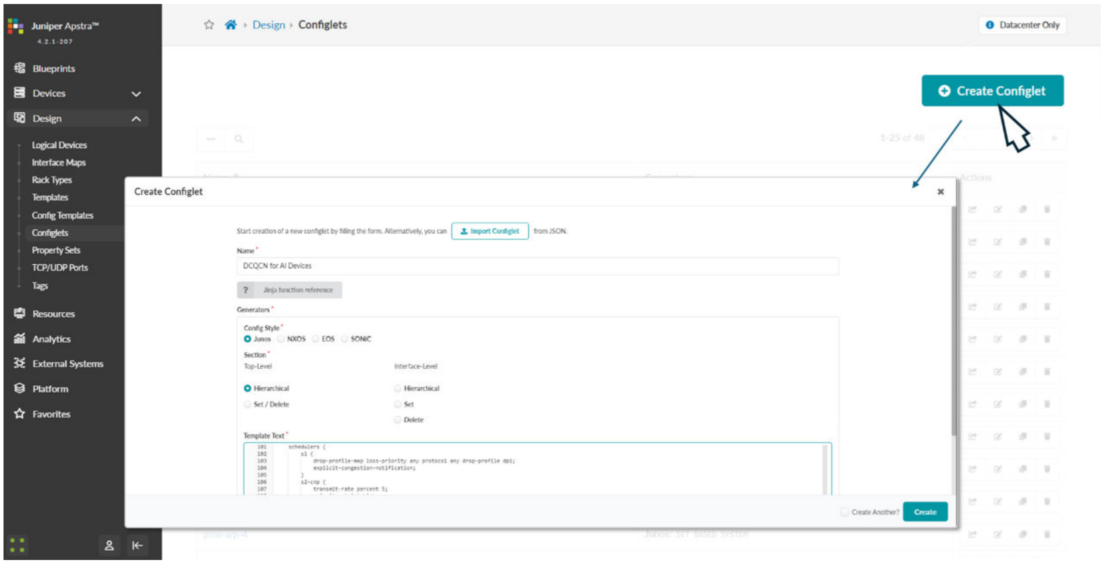
    }

}
```

To create the DCQCN configlets navigate to **Design** -> **Configlets** -> **Create Configlet**, and click on Create configlet.

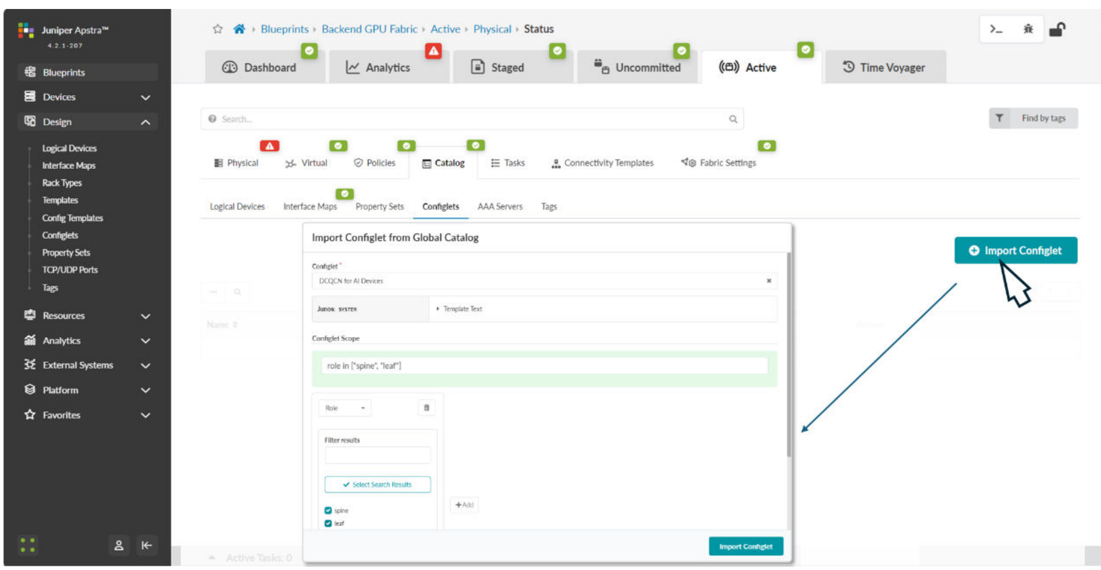
Provide a name for the config, select the operating system, vendor and configuration mode and paste the above configuration snippet on the template text box as shown below:

Figure 45: DCQCN Configlet Creation in Apstra



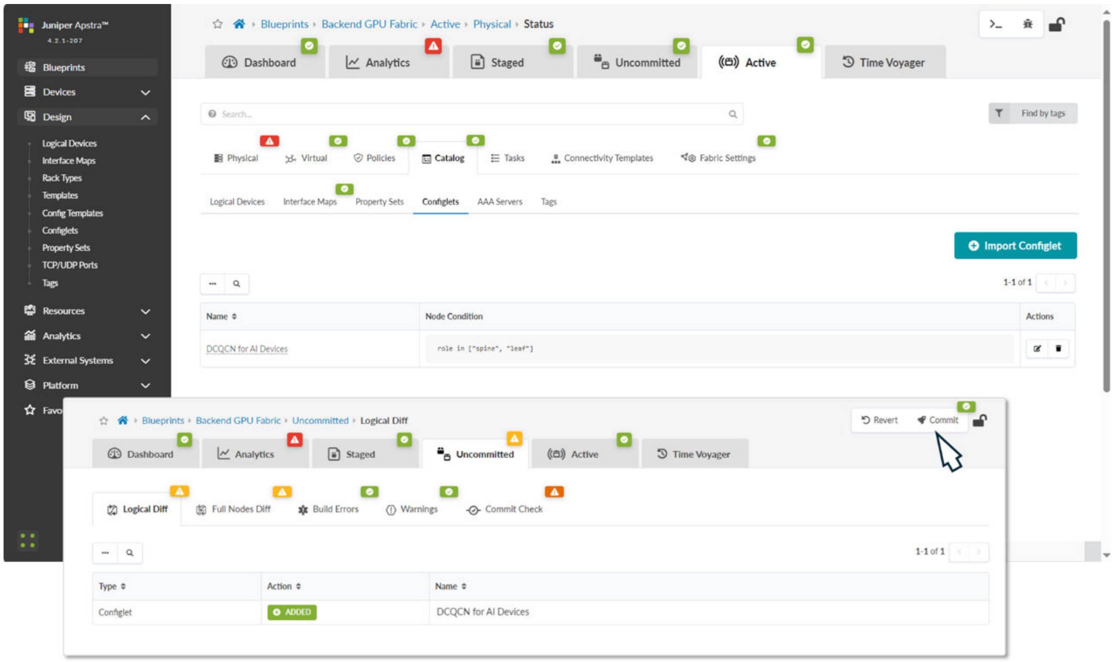
The configlet should be applied to the devices, both leaf and spine roles within the blueprint. Navigate back to the blueprint dashboard and the move to **Staged** -> **Catalog** -> **Import.** Select the configlet you want to apply, and the device role where you want to apply it.

Figure 46: Applying DCQCN Configlets to Devices in Apstra



After successfully importing the configlet into the blueprint it should be listed in the catalog. You need to commit the changes for the configuration to be deployed to the devices.

Figure 47: Applying DCQCN Configlets to Devices in Apstra



NVIDIA Configuration

IN THIS SECTION

NVIDIA® ConnectX® family of network interface cards (NICs) offer advanced hardware offload and acceleration features, and speeds up to 400G, supporting both Ethernet and Infiniband protocols.

Always refer to the official manufacturer documentation when making changes. This section provides some guidelines based on the AI JVD lab testing.

Converting NVIDIA ConnectX NICs from Infiniband to Ethernet

By default, the NVIDIA ConnectX NICs are set to operate as Infiniband interfaces and must be converted to Ethernet using the mlxconfig tool.

1) Check the status of the ConnectX NICs using sudo mst status.

NOTE: Mellanox Software Tools (MST) is part of the Mellanox firmware tools suite and can be used to manage and interact with Mellanox network adapters.

-
```
user@A100-01:/dev/mst$ sudo mst -h

Usage:

    /usr/bin/mst {start|stop|status|remote|server|restart|save|load|rm|add|help|version|
gearbox|cable} Type "/usr/bin/mst help" for detailed help

user@A100-01:/dev/mst$ sudo mst status | egrep "module|load"

MST modules:

    MST PCI module loaded

    MST PCI configuration module loaded
```

Start the mst service or load the mst modules if necessary.

Example:

- ```
  user@H100-01:~$ sudo mst start

  Starting MST (Mellanox Software Tools) driver set

  Loading MST PCI module - Success

  [warn] mst_pciconf is already loaded, skipping

  Create devices

  Unloading MST PCI module (unused) - Success

  user@A100-01:~/scripts$ sudo mst status

  MST modules:

  ------------

      MST PCI module is not loaded

      MST PCI configuration module loaded
  ```

The example shows "MST PCI module is **not loaded**". To load it, use the command **modprobe mst_pci**.

- ```
  user@A100-01:/dev/mst$ sudo modprobe mst_pci

  user@A100-01:/dev/mst$ sudo mst status

  MST modules:

  ------------

      MST PCI module loaded

      MST PCI configuration module loaded
  ```

2) Identify the interface that you want to convert,

This **sudo mst status -v** command will provide a list of Mellanox devices (ConnectX-6 and ConnectX-7 NICs) detected on the system, along with their type, Mellanox device name, PCI addresses, RDMA interface name, NET interface name, and NUMA ID, as shown in the example below:

- ```
  user@A100-01:/dev/mst$ sudo mst status -v


  MST modules:

  ------------

      MST PCI module loaded

      MST PCI configuration module loaded


  PCI devices:

  ------------


  DEVICE_TYPE         MST                           PCI       RDMA      NET                 NUMA

  ConnectX7(rev:0)    /dev/mst/mt4129_pciconf7.1    cb:00.1   mlx5_13   net-eth13           1

  ConnectX7(rev:0)    /dev/mst/mt4129_pciconf7      cb:00.0   mlx5_12   net-gpu6_eth        1

  ConnectX7(rev:0)    /dev/mst/mt4129_pciconf6.1    c8:00.1   mlx5_11   net-enp200s0f1np1   1

  ConnectX7(rev:0)    /dev/mst/mt4129_pciconf6      c8:00.0   mlx5_10   net-gpu7_eth        1

  ConnectX7(rev:0)    /dev/mst/mt4129_pciconf5.1    8e:00.1   mlx5_19   net-eth19           1

  ConnectX7(rev:0)    /dev/mst/mt4129_pciconf5      8e:00.0   mlx5_18   net-gpu5_eth        1

  ConnectX7(rev:0)    /dev/mst/mt4129_pciconf4.1    8b:00.1   mlx5_17   net-enp139s0f1np1   1

  ConnectX7(rev:0)    /dev/mst/mt4129_pciconf4      8b:00.0   mlx5_1    net-gpu4_eth        1
  ```
  ```

```
ConnectX7(rev:0)    /dev/mst/mt4129_pciconf3.1  52:00.1  mlx5_3   net-enp82s0f1np1   0

ConnectX7(rev:0)    /dev/mst/mt4129_pciconf3    52:00.0  mlx5_2   net-gpu3_eth       0

ConnectX7(rev:0)    /dev/mst/mt4129_pciconf2.1  51:00.1  mlx5_1   net-enp81s0f1np1   0

ConnectX7(rev:0)    /dev/mst/mt4129_pciconf2    51:00.0  mlx5_0   net-gpu2_eth       0

ConnectX7(rev:0)    /dev/mst/mt4129_pciconf1.1  11:00.1  mlx5_9   net-enp17s0f1np1   0

ConnectX7(rev:0)    /dev/mst/mt4129_pciconf1    11:00.0  mlx5_8   net-gpu1_eth       0

ConnectX7(rev:0)    /dev/mst/mt4129_pciconf0.1  0e:00.1  mlx5_7   net-enp14s0f1np1   0

ConnectX7(rev:0)    /dev/mst/mt4129_pciconf0    0e:00.0  mlx5_6   net-gpu0_eth       0

ConnectX6DX(rev:0)  /dev/mst/mt4125_pciconf0.1  2c:00.1  mlx5_5   net-enp44s0f1np1   0

ConnectX6DX(rev:0)  /dev/mst/mt4125_pciconf0    2c:00.0  mlx5_4   net-mgmt_eth       0

ConnectX6(rev:0)    /dev/mst/mt4123_pciconf0.1  a9:00.1  mlx5_15  net-eth15          1

ConnectX6(rev:0)    /dev/mst/mt4123_pciconf0    a9:00.0  mlx5_14  net-weka_eth       1



Cable devices:

---------------

mt4129_pciconf7_cable_0

mt4129_pciconf6_cable_0

mt4129_pciconf5_cable_0

mt4129_pciconf4_cable_0

mt4129_pciconf3_cable_0

mt4129_pciconf2_cable_0

mt4129_pciconf1_cable_0
```

```
mt4129_pciconf0_cable_0

mt4125_pciconf0_cable_0

mt4123_pciconf0_cable_0
```

For the first interface in the list, you can identify the following:

- Type = **ConnectX7(rev:0)**
- Mellanox device name = **mt4129_pciconf7 (/dev/mst/mt4129_pciconf7)**
- PCI addresses = **cb:00.0**
- RDMA interface name = **mlx5_12**
- NET interface name = **net-gpu6_eth**
- NUMA = **1**

Notice that for some of the interfaces the name follows the standard Linux interface naming scheme (e.g. net-enp14s0f1np1), while others do not (e.g. net-gpu0_eth). The interface names that do not follow the standard are user defined names for easy identification purposes. That means the default name was changed in the /etc/netplan/. We will show an example of how to do this later in this section.

3) Identify what mode a given interface is running using

mlxconfig -d <device> query

Example:

- ```
  user@A100-01:~/scripts$ sudo mlxconfig -d /dev/mst/mt4129_pciconf7query | grep LINK_TYPE

          LINK_TYPE_P1     IB(1)

          LINK_TYPE_P2     IB(1)  <= indicates link is operating in Infiniband mode
  ```

Notice that you need to use the Mellanox device name, including the path (/dev/mst/mt4129_pciconf7).

Also, LINK_TYPE_P1 and LINK_TYPE_P2 refer to the two physical ports in a dual-port Mellanox adapter.

4) If an interface is operating in Infiniband mode, you can change the mode for ethernet mode using

mlxconfig -d <device> set [LINK_TYPE_P1=<link_type>] [LINK_TYPE_P2=<link_type>]

Example
  ```

- ```
  user@A100-01:~/scripts$ sudo mlxconfig -d /dev/mst/mt4129_pciconf7 set LINK_TYPE_P1=2
  LINK_TYPE_P2=2

  Device #1:

  ----------

  Device type:        ConnectX7

  Name:               MCX755106AS-HEA_Ax

  Description:        NVIDIA ConnectX-7 HHHL Adapter Card; 200GbE (default mode) / NDR200 IB;
  Dual-port QSFP112; PCIe 5.0 x16 with x16 PCIe extension option; Crypto Disabled; Secure Boot
  Enabled

  Device:             /dev/mst/mt4129_pciconf7

  Configurations:                                         Next Boot       New

          LINK_TYPE_P1                                ETH(2)              ETH(2)

          LINK_TYPE_P2                                ETH(2)              ETH(2)

   Apply new Configuration? (y/n) [n] : y

  Applying... Done!

  -I- Please reboot machine to load new configurations.

  user@A100-01:~/scripts$ sudo mlxconfig -d /dev/mst/mt4129_pciconf7query | grep LINK_TYPE

          LINK_TYPE_P1     ETH(2)

          LINK_TYPE_P2     ETH(2)   <= indicates link is operating in Ethernet mode
  ```

Again, notice that you need to use the Mellanox device name, including the path (/dev/mst/mt4129_pciconf7).

Changes via mlxconfig require the box to be power cycled.

To check the status of the interface you can use the mlxlink:
  ```

```
user@A100-01:/dev/mst$ sudo mlxlink -d /dev/mst/mt4129_pciconf4

Operational Info

----------------

State                              : Active

Physical state                     : LinkUp

Speed                              : 200G

Width                              : 4x

FEC                                : Standard_RS-FEC - (544,514)

Loopback Mode                      : No Loopback

Auto Negotiation                   : ON

Supported Info

--------------

Enabled Link Speed (Ext.)          : 0x00003ff2
(200G_2X,200G_4X,100G_1X,100G_2X,100G_4X,50G_1X,50G_2X,40G,25G,10G,1G)

Supported Cable Speed (Ext.)       : 0x000017f2
(200G_4X,100G_2X,100G_4X,50G_1X,50G_2X,40G,25G,10G,1G)

Troubleshooting Info

--------------------

Status Opcode                      : 0

Group Opcode                       : N/A

Recommendation                     : No issue was observed

Tool Information
```

```
----------------

Firmware Version                   : 28.39.2048

amBER Version                      : 2.22

MFT Version                        : mft 4.26.0-93
```

For more details you can refer to:

HowTo Find Mellanox Adapter Type and Firmware/Driver version (Linux) (nvidia.com)

Firmware Support and Downloads - Identifying Adapter Cards (nvidia.com)

Identifying NICs and GPUs Mappings and Assigning the Appropriate Interface Name

NICs can be used by any GPU at any time; it is not hard coded that a given GPU can only communicate with the outside world using a specific NIC card. However, there are preferred communication paths between GPUs and NICs, which in some cases could be seen as a 1:1 correspondence between them. This will be shown in the steps below.

NCCL (NVIDIA Collective Communications Library) will choose the path that has the best connection from a given GPU to one of the NICs.

To identify the paths selected by NCCL and what the best path between a GPU and a NIC is, follow these steps:

Use the nvidia-smi topo -m command, which displays topological information about the system, to identify the connection type between GPUs and NICs:

EXAMPLES:

- *DGX H100:*

Figure 48. Nvidia H100 System Management Interface (SMI) system topology information

```
jvd@H100-01:~$ nvidia-smi topo -m
```

ID	GPU0	GPU1	GPU2	GPU3	GPU4	GPU5	GPU6	GPU7	NIC0	NIC1	NIC2	NIC3	NIC4	NIC5	NIC6	NIC	NIC8	NIC9	NIC10	NIC11	CPU Affinity	NUMA Affinity	GPU NUMA
GPU0	X	NV18	NV18	NV18	NV18	NV18	NV18	NV18	PXB	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	0-55,112-167	0	N/A
GPU1	NV18	X	NV18	NV18	NV18	NV18	NV18	NV18	SYS	SYS	SYS	PXB	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	0-55,112-167	0	N/A
GPU2	NV18	NV18	X	NV18	NV18	NV18	NV18	NV18	SYS	SYS	SYS	SYS	PXB	SYS	SYS	SYS	SYS	SYS	SYS	SYS	0-55,112-167	0	N/A
GPU3	NV18	NV18	NV18	X	NV18	NV18	NV18	NV18	SYS	SYS	SYS	SYS	SYS	PXB	SYS	SYS	SYS	SYS	SYS	SYS	0-55,112-167	0	N/A
GPU4	NV18	NV18	NV18	NV18	X	NV18	NV18	NV18	SYS	SYS	SYS	SYS	SYS	SYS	PXB	SYS	SYS	SYS	SYS	SYS	56-111,168-223	1	N/A
GPU5	NV18	NV18	NV18	NV18	NV18	X	NV18	NV18	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	PXB	SYS	SYS	56-111,168-223	1	N/A
GPU6	NV18	NV18	NV18	NV18	NV18	NV18	X	NV18	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	PXB	SYS	56-111,168-223	1	N/A
GPU7	NV18	NV18	NV18	NV18	NV18	NV18	NV18	X	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	PXB	56-111,168-223	1	N/A
NIC0	PXB	SYS	SYS	SYS	SYS	SYS	SYS	SYS	X	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS			
NIC1	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	X	PIX	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS			
NIC2	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	PIX	X	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS			
NIC3	SYS	PXB	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	X	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS			
NIC4	SYS	SYS	PXB	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	X	SYS	SYS	SYS	SYS	SYS	SYS	SYS			
NIC5	SYS	SYS	SYS	PXB	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	X	SYS	SYS	SYS	SYS	SYS	SYS			
NIC6	SYS	SYS	SYS	SYS	PXB	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	X	SYS	SYS	SYS	SYS	SYS			
NIC7	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	X	PIX	SYS	SYS	SYS			
NIC8	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	PIX	X	SYS	SYS	SYS			
NIC9	SYS	SYS	SYS	SYS	SYS	PXB	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	X	SYS	SYS			
NIC10	SYS	SYS	SYS	SYS	SYS	SYS	PXB	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	X	SYS			
NIC11	SYS	SYS	SYS	SYS	SYS	SYS	SYS	PXB	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	X			

```
Legend:

  X    = Self
  SYS  = Connection traversing PCIe as well as the SMP interconnect between NUMA nodes (e.g., QPI/UPI)
  NODE = Connection traversing PCIe as well as the interconnect between PCIe Host Bridges within a NUMA node
  PHB  = Connection traversing PCIe as well as a PCIe Host Bridge (typically the CPU)
  PXB  = Connection traversing multiple PCIe bridges (without traversing the PCIe Host Bridge)
  PIX  = Connection traversing at most a single PCIe bridge
  NV#  = Connection traversing a bonded set of # NVLinks

NIC Legend:

  NIC0: mlx5_0
  NIC1: mlx5_1
  NIC2: mlx5_2
  NIC3: mlx5_3
  NIC4: mlx5_4
  NIC5: mlx5_5
  NIC6: mlx5_6
  NIC7: mlx5_7
  NIC8: mlx5_8
  NIC9: mlx5_9
  NIC10: mlx5_10
  NIC11: mlx5_11
```

System Management Interface SMI | NVIDIA Developer

Based on our research:

Table 21: Performance per connection type

Connection Type	Description	Performance
PIX	PCIe on the same switch	Good
PXB	PCIe through multiple switches, but not host bridge	Good
PHB	PCIe switch and across a host bridge on the same NUMA - uses CPU	OK
NODE	PCIe switch and across multiple host bridge on the same NUMA	Bad
SYS	PCIe switch and across QPI/UPI bus between NUMA nodes - uses CPU	Very Bad

NV#	NVLink	Very Good

- *HGX A100:*

Figure 49.Nvidia A100 System Management Interface (SMI) system topology information

```
user@A100-01:~/scripts$ nvidia-smi topo -m
```

	GPU0	GPU1	GPU2	GPU3	GPU4	GPU5	GPU6	GPU7	NIC0	NIC1	NIC2	NIC3	NIC4	NIC5	NIC6	NIC7	NIC8	NIC9
GPU0	X	NV12	NV12	NV12	NV12	NV12	NV12	NV12	NODE	NODE	NODE	NODE	NODE	NODE	PXB	PXB	PXB	PXB
GPU1	NV12	X	NV12	NV12	NV12	NV12	NV12	NV12	NODE	NODE	NODE	NODE	NODE	NODE	PXB	PXB	PXB	PXB
GPU2	NV12	NV12	X	NV12	NV12	NV12	NV12	NV12	PXB	PXB	PXB	PXB	NODE	NODE	NODE	NODE	NODE	NODE
GPU3	NV12	NV12	NV12	X	NV12	NV12	NV12	NV12	PXB	PXB	PXB	PXB	NODE	NODE	NODE	NODE	NODE	NODE
GPU4	NV12	NV12	NV12	NV12	X	NV12	NV12	NV12	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS
GPU5	NV12	NV12	NV12	NV12	NV12	X	NV12	NV12	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS
GPU6	NV12	NV12	NV12	NV12	NV12	NV12	X	NV12	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS
GPU7	NV12	NV12	NV12	NV12	NV12	NV12	NV12	X	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS
NIC0	NODE	NODE	PXB	PXB	SYS	SYS	SYS	SYS	X	PIX	PIX	PIX	NODE	NODE	NODE	NODE	NODE	NODE
NIC1	NODE	NODE	PXB	PXB	SYS	SYS	SYS	SYS	PIX	X	PIX	PIX	NODE	NODE	NODE	NODE	NODE	NODE
NIC2	NODE	NODE	PXB	PXB	SYS	SYS	SYS	SYS	PIX	PIX	X	PIX	NODE	NODE	NODE	NODE	NODE	NODE
NIC3	NODE	NODE	PXB	PXB	SYS	SYS	SYS	SYS	PIX	PIX	PIX	X	NODE	NODE	NODE	NODE	NODE	NODE
NIC4	NODE	NODE	NODE	NODE	SYS	SYS	SYS	SYS	NODE	NODE	NODE	NODE	X	PIX	NODE	NODE	NODE	NODE
NIC5	NODE	NODE	NODE	NODE	SYS	SYS	SYS	SYS	NODE	NODE	NODE	NODE	PIX	X	NODE	NODE	NODE	NODE
NIC6	PXB	PXB	NODE	NODE	SYS	SYS	SYS	SYS	NODE	NODE	NODE	NODE	NODE	NODE	X	PIX	PXB	PXB
NIC7	PXB	PXB	NODE	NODE	SYS	SYS	SYS	SYS	NODE	NODE	NODE	NODE	NODE	NODE	PIX	X	PXB	PXB
NIC8	PXB	PXB	NODE	NODE	SYS	SYS	SYS	SYS	NODE	NODE	NODE	NODE	NODE	NODE	PXB	PXB	X	PIX
NIC9	PXB	PXB	NODE	NODE	SYS	SYS	SYS	SYS	NODE	NODE	NODE	NODE	NODE	NODE	PXB	PXB	PIX	X
NIC10	SYS	SYS	SYS	SYS	NODE	NODE	PXB	PXB	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS
NIC11	SYS	SYS	SYS	SYS	NODE	NODE	PXB	PXB	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS
NIC12	SYS	SYS	SYS	SYS	NODE	NODE	PXB	PXB	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS
NIC13	SYS	SYS	SYS	SYS	NODE	NODE	PXB	PXB	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS
NIC14	SYS	SYS	SYS	SYS	NODE	NODE	NODE	NODE	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS
NIC15	SYS	SYS	SYS	SYS	NODE	NODE	NODE	NODE	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS
NIC16	SYS	SYS	SYS	SYS	PXB	PXB	NODE	NODE	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS
NIC17	SYS	SYS	SYS	SYS	PXB	PXB	NODE	NODE	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS
NIC18	SYS	SYS	SYS	SYS	PXB	PXB	NODE	NODE	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS
NIC19	SYS	SYS	SYS	SYS	PXB	PXB	NODE	NODE	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS

	NIC10	NIC11	NIC12	NIC13	NIC14	NIC15	NIC16	NIC17	NIC18	NIC19	CPU Affinity	NUMA Affinity	GPU NUMA ID
GPU0	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	0-63,128-191	0	N/A
GPU1	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	0-63,128-191	0	N/A
GPU2	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	0-63,128-191	0	N/A
GPU3	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	0-63,128-191	0	N/A
GPU4	NODE	NODE	NODE	NODE	NODE	NODE	PXB	PXB	PXB	PXB	64-127,192-255	1	N/A
GPU5	NODE	NODE	NODE	NODE	NODE	NODE	PXB	PXB	PXB	PXB	64-127,192-255	1	N/A
GPU6	PXB	PXB	PXB	PXB	NODE	NODE	NODE	NODE	NODE	NODE	64-127,192-255	1	N/A
GPU7	PXB	PXB	PXB	PXB	NODE	NODE	NODE	NODE	NODE	NODE	64-127,192-255	1	N/A
NIC0	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS			
NIC1	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS			
NIC2	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS			
NIC3	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS			
NIC4	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS			
NIC5	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS			
NIC6	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS			
NIC7	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS			
NIC8	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS			
NIC9	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS	SYS			
NIC10	X	PIX	PXB	PXB	NODE	NODE	NODE	NODE	NODE	NODE			
NIC11	PIX	X	PXB	PXB	NODE	NODE	NODE	NODE	NODE	NODE			
NIC12	PXB	PXB	X	PIX	NODE	NODE	NODE	NODE	NODE	NODE			
NIC13	PXB	PXB	PIX	X	NODE	NODE	NODE	NODE	NODE	NODE			
NIC14	NODE	NODE	NODE	NODE	X	PIX	NODE	NODE	NODE	NODE			
NIC15	NODE	NODE	NODE	NODE	PIX	X	NODE	NODE	NODE	NODE			
NIC16	NODE	NODE	NODE	NODE	NODE	NODE	X	PIX	PXB	PXB			
NIC17	NODE	NODE	NODE	NODE	NODE	NODE	PIX	X	PXB	PXB			
NIC18	NODE	NODE	NODE	NODE	NODE	NODE	PXB	PXB	X	PIX			
NIC19	NODE	NODE	NODE	NODE	NODE	NODE	PXB	PXB	PIX	X			

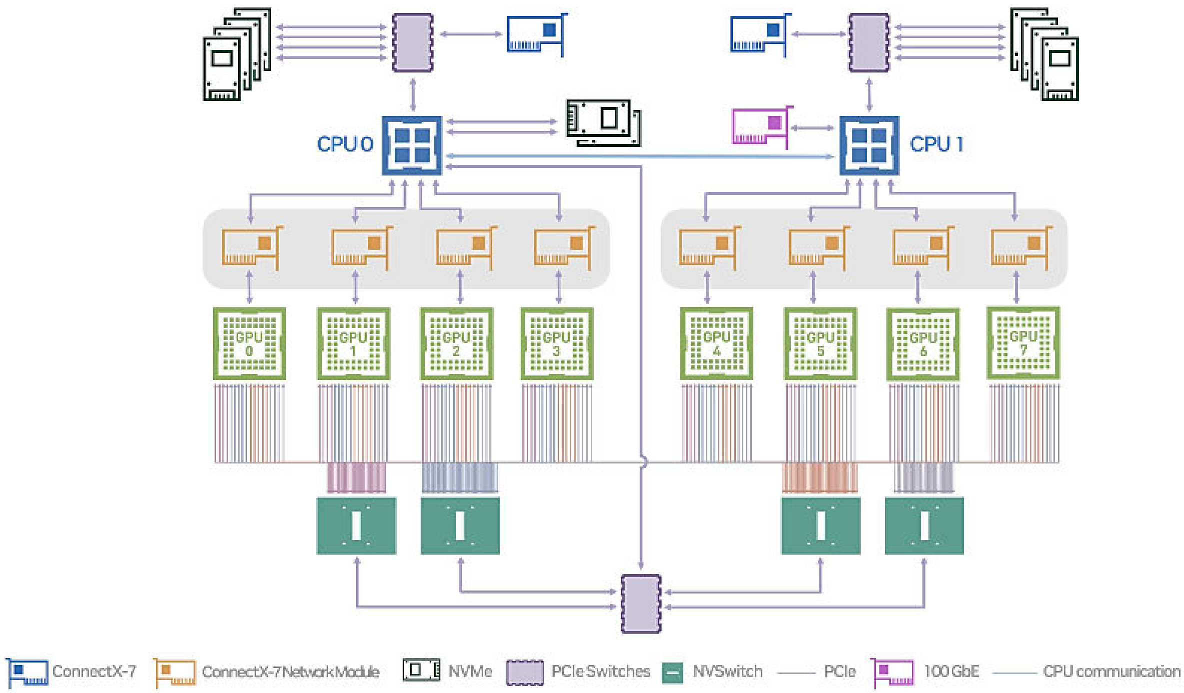
Identify PBX Connections

If you focus on the highlighted sections of the nvidia-smi output, you can see that for each GPU there is one or more NIC connection(s) of type **PXB**. This is the preferred “direct” path from each GPU to a given NIC. That means, when the GPU needs to communicate to a remote device, it will use one of these specific NICs, as the first option.

- *DGX H100:*

Figure 50. Nvidia H100 System Management Interface (SMI) system topology PBX connections

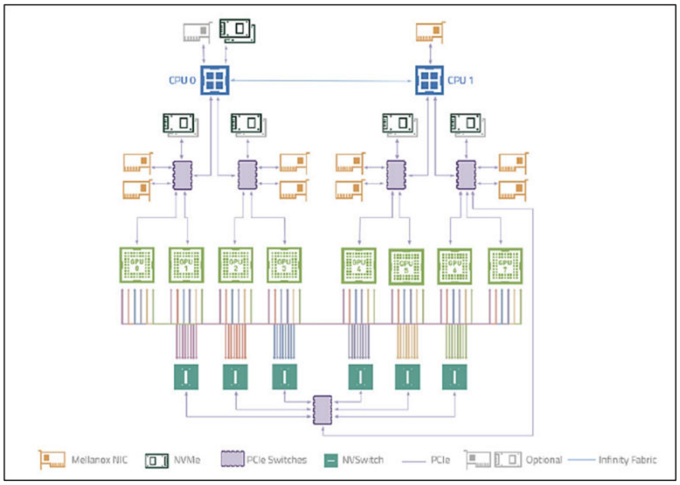
	GPU0	GPU1	GPU2	GPU3	GPU4	GPU5	GPU6	GPU7
NIC0	PXB	SYS	SYS	SYS	SYS	SYS	SYS	SYS
NIC3	SYS	PXB	SYS	SYS	SYS	SYS	SYS	SYS
NIC4	SYS	SYS	PXB	SYS	SYS	SYS	SYS	SYS
NIC5	SYS	SYS	SYS	PXB	SYS	SYS	SYS	SYS
NIC6	SYS	SYS	SYS	SYS	PXB	SYS	SYS	SYS
NIC9	SYS	SYS	SYS	SYS	SYS	PXB	SYS	SYS
NIC10	SYS	SYS	SYS	SYS	SYS	SYS	PXB	SYS
NIC11	SYS	SYS	SYS	SYS	SYS	SYS	SYS	PXB



- *HGX A100:*

Figure 51. Nvidia A100 System Management Interface (SMI) system topology PBX connections

	GPU0	GPU1	GPU2	GPU3	GPU4	GPU5	GPU6	GPU7
NIC0	NODE	NODE	PXB	PXB	SYS	SYS	SYS	SYS
NIC1	NODE	NODE	PXB	PXB	SYS	SYS	SYS	SYS
NIC2	NODE	NODE	PXB	PXB	SYS	SYS	SYS	SYS
NIC3	NODE	NODE	PXB	PXB	SYS	SYS	SYS	SYS
NIC6	PXB	PXB	NODE	NODE	SYS	SYS	SYS	SYS
NIC7	PXB	PXB	NODE	NODE	SYS	SYS	SYS	SYS
NIC8	PXB	PXB	NODE	NODE	SYS	SYS	SYS	SYS
NIC9	PXB	PXB	NODE	NODE	SYS	SYS	SYS	SYS
NIC10	SYS	SYS	SYS	SYS	NODE	NODE	PXB	PXB
NIC11	SYS	SYS	SYS	SYS	NODE	NODE	PXB	PXB
NIC12	SYS	SYS	SYS	SYS	NODE	NODE	PXB	PXB
NIC13	SYS	SYS	SYS	SYS	NODE	NODE	PXB	PXB
NIC16	SYS	SYS	SYS	SYS	PXB	PXB	NODE	NODE
NIC17	SYS	SYS	SYS	SYS	PXB	PXB	NODE	NODE
NIC18	SYS	SYS	SYS	SYS	PXB	PXB	NODE	NODE
NIC19	SYS	SYS	SYS	SYS	PXB	PXB	NODE	NODE



These paths are fixed.

You can also find these mappings in Nvidia's A100 or H100 user guides.

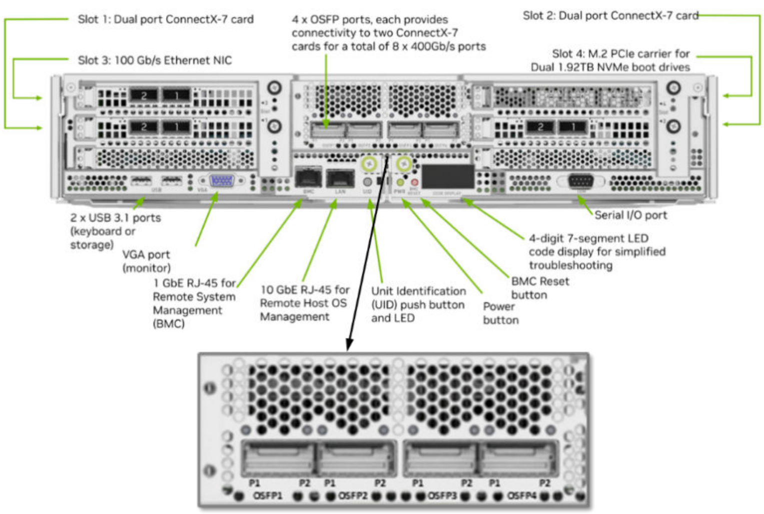
For example, on an DGX H100/H200 System the port mappings according to the NVIDIA's DGX H100/H200 System User Guide table 5 and table 6 is as follows:

Table 22: GPU to NIC Mappings

Port	ConnectX	GPU	Default	RDMA	NIC
OSFP4P2	CX1	0	ibp24s0	mlx5_0	NIC0
OSFP3P2	CX3	1	ibp64s0	mlx5_3	NIC3
OSFP3P1	CX2	2	ibp79s0	mlx5_4	NIC4
OSFP4P1	CX0	3	ibp94s0	mlx5_5	NIC5
OSFP1P2	CX1	4	ibp154s0	mlx5_6	NIC6
OSFP2P2	CX3	5	ibp192s0	mlx5_9	NIC9
OSFP2P1	CX2	6	ibp206s0	mlx5_10	NIC10
OSFP1P1	CX0	7	ibp220s0	mlx5_11	NIC11

Table 23: GPU to NIC Connections

NIC	GPU0	GPU1	GPU2	GPU3	GPU4	GPU5	GPU6	GPU7
NIC0	*PXB*	SYS	SYS	SYS	SYS	SYS	SYS	SYS
NIC3	SYS	*PXB*	SYS	SYS	SYS	SYS	SYS	SYS
NIC4	SYS	SYS	*PXB*	SYS	SYS	SYS	SYS	SYS
NIC5	SYS	SYS	SYS	*PXB*	SYS	SYS	SYS	SYS
NIC6	SYS	SYS	SYS	SYS	*PXB*	SYS	SYS	SYS
NIC9	SYS	SYS	SYS	SYS	SYS	*PXB*	SYS	SYS
NIC10	SYS	SYS	SYS	SYS	SYS	SYS	*PXB*	SYS
NIC11	SYS	SYS	SYS	SYS	SYS	SYS	SYS	*PXB*



		Port Designation		
Port	PCI Bus	Default	Optional	RDMA
OSFP1P1	dc:00.0	ibp220s0	enp220s0np0	mlx5_11
OSFP1P2	9a:00.0	ibp154s0	enp154s0np0	mlx5_6
OSFP2P1	ce:00.0	ibp206s0	enp206s0np0	mlx5_10
OSFP2P2	c0:00.0	ibp192s0	enp192s0np0	mlx5_9
OSFP3P1	4f:00.0	ibp79s0	enp79s0np0	mlx5_4
OSFP3P2	40:00.0	ibp64s0	enp64s0np0	mlx5_3
OSFP4P1	5e:00.0	ibp94s0	enp94s0np0	mlx5_5
OSFP4P2	18:00.0	ibp24s0	enp24s0np0	mlx5_0
Slot1 P1	aa:00.0	ibp170s0f0	enp170s0f0np0	mlx5_7
Slot1 P2	aa:00.1	enp170s0f1np1	ibp170s0f1np1	mlx5_8
Slot2 P1	29:00.0	ibp41s0f0	enp41s0f0np0	mlx5_1
Slot2 P2	29:00.1	enp41s0f1np1	ibp41s0f1np1	mlx5_2
Slot3 P1	82:00.0	ens6f0	N/A	irdma0
Slot3 P2	82:00.1	ens6f1	N/A	irdma1
On-board	0b:00.0	eno3	N/A	

Port	ConnectX Device	Network Module/CPU	GPU	Default	RDMA
OSFP1P1	CX0	1	7	ibp220s0	mlx5_11
OSFP1P2	CX1	1	4	ibp154s0	mlx5_6
OSFP2P1	CX2	1	6	ibp206s0	mlx5_10
OSFP2P2	CX3	1	5	ibp192s0	mlx5_9
OSFP3P1	CX2	0	2	ibp79s0	mlx5_4
OSFP3P2	CX3	0	1	ibp64s0	mlx5_3
OSFP4P1	CX0	0	3	ibp94s0	mlx5_5
OSFP4P2	CX1	0	0	ibp24s0	mlx5_0

For more information and for the mappings on the A100 systems check:

Introduction to the NVIDIA HGX A100 System — NVIDIA HGX A100 User Guide 1 documentation

Introduction to NVIDIA DGX H100/H200 Systems — NVIDIA DGX H100/H200 User Guide 1 documentation

Changing NIC attributes

The following sections describe how to change NIC attributes.

How to Change a NIC's Interface Name, and Assign IP Addresses and Routes

NIC attributes such as the IP address or the interface name can be made by editing and reapplying the netplan.

The network configuration is described in the file: /etc/netplan/01-netcfg.yaml as shown in the example table below. Any attribute changes involve editing this file and reapplying the network plan as will be shown in the examples later in this section.

Table 24: Nvidia HGX A100 interface configuration example:

netcfg.yaml output
jvd@A100-01:/etc/netplan$ more 01-netcfg.yaml

# This is the network config written by 'subiquity'	**gpu0_eth:**	**gpu4_eth:**
network:	match:	match:
version: 2	macaddress: 94:6d:ae:54:72:22	macaddress: 94:6d:ae:5b:28:70
ethernets:	dhcp4: false	dhcp4: false
mgmt_eth:	mtu: 9000	mtu: 9000
match:	addresses:	addresses:
macaddress: 7c:c2:55:42:b2:28	- 10.200.0.8/24	- 10.200.4.8/24
dhcp4: false	routes:	routes:
addresses:	- to: 10.200.0.0/16	- to: 10.200.0.0/16
- 10.10.1.0/31	via: 10.200.0.254	via: 10.200.4.254
nameservers:	from: 10.200.0.8	from: 10.200.4.8
addresses:	**set-name: gpu0_eth**	**set-name: gpu4_eth**
- 8.8.8.8	**gpu1_eth:**	**gpu5_eth:**
routes:	match:	match:
- to: default	macaddress: 94:6d:ae:5b:01:d0	macaddress: 94:6d:ae:5b:27:f0
via: 10.10.1.1	dhcp4: false	dhcp4: false
set-name: mgmt_eth	mtu: 9000	mtu: 9000
weka_eth:	addresses:	addresses:
match:	- 10.200.1.8/24	- 10.200.5.8/24
macaddress: b8:3f:d2:8b:68:e0	routes:	routes:

dhcp4: false	- to: 10.200.0.0/16	- to: 10.200.0.0/16
mtu: 9000	via: 10.200.1.254	via: 10.200.5.254
addresses:	from: 10.200.1.8	from: 10.200.5.8
- 10.100.1.0/31	set-name: gpu1_eth	set-name: gpu5_eth
routes:	gpu2_eth:	gpu6_eth:
- to: 10.100.0.0/22	match:	match:
via: 10.100.1.1	macaddress: 94:6d:ae:5b:28:60	macaddress: 94:6d:ae:54:78:e2
set-name: weka_eth	dhcp4: false	dhcp4: false
	mtu: 9000	mtu: 9000
	addresses:	addresses:
	- 10.200.2.8/24	- 10.200.6.8/24
	routes:	routes:
	- to: 10.200.0.0/16	- to: 10.200.0.0/16
	via: 10.200.2.254	via: 10.200.6.254
	from: 10.200.2.8	from: 10.200.6.8
	set-name: gpu2_eth	set-name: gpu6_eth
	gpu3_eth:	gpu7_eth:
	match:	match:
	macaddress: 94:6d:ae:5b:01:e0	macaddress: 94:6d:ae:54:72:12
	dhcp4: false	dhcp4: false

	mtu: 9000	mtu: 9000
	addresses:	addresses:
	- 10.200.3.8/24	- 10.200.7.8/24
	routes:	routes:
	- to: 10.200.0.0/16	- to: 10.200.0.0/16
	via: 10.200.3.254	via: 10.200.7.254
	from: 10.200.3.8	from: 10.200.7.8
	set-name: gpu3_eth	set-name: gpu7_eth

To Map an Interface Name to a Specific NIC (Physical Interface)

Map the interface name to the MAC of the physical interface in the configuration file:

Figure 53. Nvidia A100 physical interface identification example

```
user@A100-01:/etc/netplan$ ifconfig | grep enp
enp203s0f1np1: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500

user@A100-01:/etc/netplan$ ifconfig enp203s0f1np1
enp203s0f1np1: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
        ether 94:6d:ae:54:78:e3 txqueuelen 1000  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

enp203s0f1np1   <= default logical interface name with MAC =94:6d:ae:54:78:e3,
```

where:

en = ethernet network interface.

p203s0 = physical location of the network interface.

203 bus number.

s0 = slot number 0 on the bus.

f1 = function number 1 for the network interface.

np1 = Network Port 1

```
enp4s10f1                        pci 0000:04:0a.1
| | |  |                                |  |  | |
| | |  |                   domain <- 0000  |  | |
| | |  |                                   |  | |
en| |  |  --> ethernet                     |  | |
  | |  |                                   |  | |
  p4|  |  --> prefix/bus number (4)   <-- 04  | |
    |  |                                      | |
    s10|  --> slot/device number (10) <--    10 |
       |                                        |
       f1 --> function number (1)     <--       1
```

Function 0: Might be the primary Ethernet interface.

Function 1: Might be a second Ethernet interface.

Function 2: Might be a management or diagnostics interface.

Figure 54. Nvidia A100 netplan file modification example

```
user@A100-01:/etc/netplan$ vi 01-netcfg.yaml
---more---
new_interface:
      match:
        macaddress: 94:6d:ae:54:78:e3
      dhcp4: false
      mtu: 9000
      addresses:
        - 10.200.16.1/24
      routes:
        - to: 10.200.0.0/16
          via: 10.200.16.254
          from: 10.200.16.1
      set-name: new_iface_name     <= new logical interface name with MAC =94:6d:ae:54:78:e3
-- INSERT -
```

You can find the names of all the logical interfaces on the devnames file:

- ```
  user@A100-01:/etc/network$ more devnames

  enp139s0f0np0:Mellanox Technologies MT2910 Family [ConnectX-7]

  enp139s0f1np1:Mellanox Technologies MT2910 Family [ConnectX-7]

  enp142s0f0np0:Mellanox Technologies MT2910 Family [ConnectX-7]

  enp142s0f1np1:Mellanox Technologies MT2910 Family [ConnectX-7]
  ```
  ```

```
enp14s0f0np0:Mellanox Technologies MT2910 Family [ConnectX-7]

enp14s0f1np1:Mellanox Technologies MT2910 Family [ConnectX-7]

enp17s0f0np0:Mellanox Technologies MT2910 Family [ConnectX-7]

enp17s0f1np1:Mellanox Technologies MT2910 Family [ConnectX-7]

enp200s0f0np0:Mellanox Technologies MT2910 Family [ConnectX-7]

enp200s0f1np1:Mellanox Technologies MT2910 Family [ConnectX-7]

enp203s0f0np0:Mellanox Technologies MT2910 Family [ConnectX-7]

enp203s0f1np1:Mellanox Technologies MT2910 Family [ConnectX-7]

enp44s0f0:Intel Corporation Ethernet Controller X710 for 10GBASE-T

enp44s0f1:Intel Corporation Ethernet Controller X710 for 10GBASE-T

enp44s0f2:Intel Corporation Ethernet Controller X710 for 10 Gigabit SFP+

enp44s0f3:Intel Corporation Ethernet Controller X710 for 10 Gigabit SFP+

enp81s0f0np0:Mellanox Technologies MT2910 Family [ConnectX-7]

enp81s0f1np1:Mellanox Technologies MT2910 Family [ConnectX-7]

enp82s0f0np0:Mellanox Technologies MT2910 Family [ConnectX-7]

enp82s0f1np1:Mellanox Technologies MT2910 Family [ConnectX-7]

ibp169s0f0:Mellanox Technologies MT28908 Family [ConnectX-6]

ibp169s0f1:Mellanox Technologies MT28908 Family [ConnectX-6]
```

Apply the changes using the netplan apply command

Figure 55. Nvidia A100 netplan application example

```
user@A100-01:/etc/netplan$ sudo ip link set dev enp203s0f1np1 down

user@A100-01:/etc/netplan$ ifconfig enp203s0f1np1
enp203s0f1np1: error fetching interface information: Device not found

user@A100-01:/etc/netplan$ sudo netplan apply

user@A100-01:/etc/netplan$ ifconfig new_iface_name
new_iface_name: flags=4099<UP,BROADCAST,MULTICAST>  mtu 9000
        ether 94:6d:ae:54:78:e3  txqueuelen 1000  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

To Change the NIC Name

Change the value of set-name in the configuration file and save the changes:

Figure 56. Nvidia A100 netplan interface name change example

```
jvd@A100-01:/etc/netplan$ ifconfig gpu0_eth  <= current name
gpu0_eth: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 9000
        inet 10.200.0.8  netmask 255.255.255.0  broadcast 10.200.0.255
        inet6 fe80::966d:aeff:fe54:7222  prefixlen 64  scopeid 0x20<link>
        ether 94:6d:ae:54:72:22  txqueuelen 1000  (Ethernet)
        RX packets 2079477652  bytes 17618315023885 (17.6 TB)
        RX errors 0  dropped 8  overruns 0  frame 0
        TX packets 2082335255  bytes 17741532549214 (17.7 TB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

jvd@A100-01:/etc/netplan$ vi 01-netcfg.yaml
 ---more---
    gpu0_eth:
      match:
        macaddress: 94:6d:ae:54:72:22
      dhcp4: false
      mtu: 9000
      addresses:
        - 10.200.0.8/24
      routes:
        - to: 10.200.0.0/16
          via: 10.200.0.254
          from: 10.200.0.8
            set-name: gpu0_eth   <= current name

jvd@A100-01:/etc/netplan$ cat 01-netcfg.yaml
 ---more---
    gpu0_eth:
      match:
        macaddress: 94:6d:ae:54:72:22
      dhcp4: false
      mtu: 9000
      addresses:
        - 10.200.0.8/24
      routes:
        - to: 10.200.0.0/16
          via: 10.200.0.254
          from: 10.200.0.8
            set-name: gpu0_eth0   <= new name
:wq
```

Apply the Changes Using the netplan apply command

Figure 57. Nvidia A100 netplan interface name change application and verification example

Figure 45. Nvidia A100 netplan interface name change application and verification example

```
user@A100-01:/etc/netplan$ sudo netplan apply

user@A100-01:/etc/netplan$ ifconfig gpu0_eth0    <= new name
gpu0_eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 9000
        inet 10.200.0.8  netmask 255.255.255.0  broadcast 10.200.0.255
        inet6 fe80::966d:aeff:fe54:7222  prefixlen 64  scopeid 0x20<link>
        ether 94:6d:ae:54:72:22  txqueuelen 1000  (Ethernet)
        RX packets 2079477704  bytes 17618315028610 (17.6 TB)
        RX errors 0  dropped 8  overruns 0  frame 0
        TX packets 2082335268  bytes 17741532551122 (17.7 TB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

To Change the Current IP Address or Assign an IP Address to the NIC

Change or add the address under the proper interface in the configuration file, and save the changes:

Figure 58. Nvidia A100 netplan interface IP address change example

```
user@A100-01:/etc/netplan$ ifconfig gpu0_eth
gpu0_eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 9000
    inet 10.200.0.8  netmask 255.255.255.0  broadcast 10.200.0.255 <= current IP address
    inet6 fe80::966d:aeff:fe54:7222  prefixlen 64  scopeid 0x20<link>
    ether 94:6d:ae:54:72:22  txqueuelen 1000  (Ethernet)
    RX packets 2079477704  bytes 17618315028610 (17.6 TB)
    RX errors 0  dropped 8  overruns 0  frame 0
    TX packets 2082335268  bytes 17741532551122 (17.7 TB)
    TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

user@A100-01:/etc/netplan$ vi 01-netcfg.yaml
---more---
   gpu0_eth:
      match:
        macaddress: 94:6d:ae:54:72:22
      dhcp4: false
      mtu: 9000
      addresses:
        - 10.200.0.8/24 <= current IP address

user@A100-01:/etc/netplan$ vi 01-netcfg.yaml
---more---
   gpu0_eth:
      match:
        macaddress: 94:6d:ae:54:72:22
      dhcp4: false
      mtu: 9000
      addresses:
        - 10.200.0.18/24        <= new IP address
:wq
```

Enter the IP addresses preceded with a hyphen and indented; make sure to add the subnet mask.

Apply the Changes Using the netplan apply Command

Figure 59. Nvidia A100 netplan interface new IP address application and verification example

```
user@A100-01:/etc/netplan$ sudo netplan apply

user@A100-01:/etc/netplan$ ifconfig gpu0_eth
gpu0_eth: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 9000
        inet 10.200.0.18  netmask 255.255.255.0  broadcast 10.200.0.255  <= new IP address
        inet6 fe80::966d:aeff:fe54:7222  prefixlen 64  scopeid 0x20<link>
        ether 94:6d:ae:54:72:22  txqueuelen 1000  (Ethernet)
        RX packets 2079478284  bytes 17618315075628 (17.6 TB)
        RX errors 0  dropped 8  overruns 0  frame 0
        TX packets 2082335328  bytes 17741532561365 (17.7 TB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

To Change or Add Routes to the NIC

Change or add the routes under the proper interface in the configuration file, and save the changes.

Figure 60. Nvidia A100 netplan additional routes example

```
jvd@A100-02:~$ route | grep gpu0
10.200.0.0  0.0.0.0       255.255.255.0   U   0  0   0 gpu0_eth
10.200.0.0  10.200.0.254  255.255.0.0     UG  0  0   0 gpu0_eth  <= current routes

jvd@A100-01:/etc/netplan$ vi 01-netcfg.yaml
---more---
    gpu0_eth:
      match:
        macaddress: 94:6d:ae:54:72:22
      dhcp4: false
      mtu: 9000
      addresses:
        - 10.200.0.8/24
      routes:
        - to: 10.200.0.0/16
          via: 10.200.0.254
          from: 10.200.0.8  <= current routes
      set-name: gpu0_eth

jvd@A100-01:/etc/netplan$ vi 01-netcfg.yaml
---more---
    gpu0_eth:
      match:
        macaddress: 94:6d:ae:54:72:22
      dhcp4: false
      mtu: 9000
      addresses:
        - 10.200.0.18/24
      routes:
        - to: 10.200.0.0/16
          via: 10.200.0.254
          from: 10.200.0.8
        - to: 10.100.0.0/16
          via: 10.200.0.254     <= new route
      set-name: gpu0_eth
```

Apply the changes using the netplan apply command

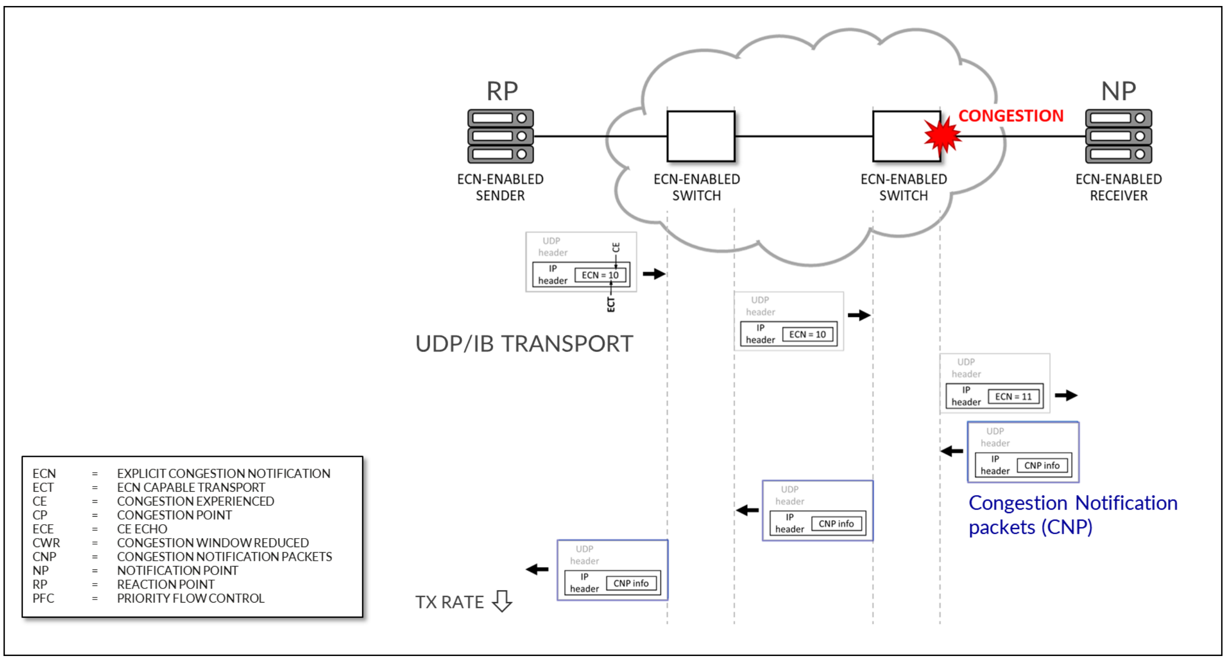
Figure 61. Nvidia A100 netplan additional routes application and verification example:

```
user@A100-01:/etc/netplan$ sudo netplan apply

user@A100-01:/etc/netplan$ route| grep gpu0
10.100.0.0      10.200.0.254    255.255.0.0     UG    0      0        0 gpu0_eth   <= new route
10.200.0.0      0.0.0.0         255.255.255.0   U     0      0        0 gpu0_eth
10.200.0.0      10.200.0.254    255.255.0.0     UG    0      0        0 gpu0_eth
```

Configuring NVIDIA DCQCN – ECN

Figure 62: NVIDIA DCQCN – ECN



Starting from MLNX_OFED 4.1 ECN is enabled by default (in the firmware).

To confirm that ECN is enabled, use the following command: **mlxconfig -d <device> q | grep ROCE_CC**

Example:

```
root@A100-01:/home/ylara# mlxconfig -d mlx5_0 q | grep ROCE_CC

        ROCE_CC_PRIO_MASK_P1        255

        ROCE_CC_PRIO_MASK_P2        255
```

A mask of 255 means DCQCN (ECN) is enabled for all TC (traffic classes) configured on the NIC.

To disable ECN you can change the mask using the following command: **mlxconfig -d <device> s** ROCE_CC_PRIO_MASK_P1=<mask>

Example:

- ```
  root@A100-01:/home/ylara# sudo mlxconfig -d mlx5_0 s ROCE_CC_PRIO_MASK_P1=0

  Device #1:

  ----------

  Device type:    ConnectX7

  Name:           MCX755106AS-HEA_Ax

  Description:    NVIDIA ConnectX-7 HHHL Adapter Card; 200GbE (default mode) / NDR200 IB; Dual-
  port QSFP112; PCIe 5.0 x16 with x16 PCIe extension option; Crypto Disabled; Secure Boot
  Enabled

  Device:         mlx5_0

  Configurations:                                    Next Boot       New

           ROCE_CC_PRIO_MASK_P1                      0               0

   Apply new Configuration? (y/n) [n] :
  ```

If you want to avoid being asked whether you want to apply the new configuration you an include the -y option as shown in the following example:

- ```
  root@A100-01:/home/ylara# sudo mlxconfig -d mlx5_0 -y s ROCE_CC_PRIO_MASK_P1=0

  Device #1:

  ----------

  Device type:    ConnectX7

  Name:           MCX755106AS-HEA_Ax

  Description:    NVIDIA ConnectX-7 HHHL Adapter Card; 200GbE (default mode) / NDR200 IB; Dual-
  ```

```
port QSFP112; PCIe 5.0 x16 with x16 PCIe extension option; Crypto Disabled; Secure Boot
Enabled

Device:         mlx5_0

Configurations:                                     Next Boot       New

         ROCE_CC_PRIO_MASK_P1                       0               0

 Apply new Configuration? (y/n) [n] : y

Applying... Done!

-I- Please reboot machine to load new configurations.
```

The output states that a server reboot is required. As an alternative, you can reset the interface using the command: **mlxfwreset -d <device> -l 3 -y r.**

The device can be entered as /dev/mst/mt4129_pciconf2 or **mlx5_0** (gpu0_eth is not a valid format for this command)

Example:

- ```
  root@A100-01:/home/ylara# mlxfwreset -d mlx5_0 -l 3 -y r

  Requested reset level for device, /dev/mst/mt4129_pciconf2:

  3: Driver restart and PCI reset

  Continue with reset?[y/N] y

  -I- Sending Reset Command To Fw             -Done

  -I- Stopping Driver                         -Done

  -I- Resetting PCI                           -Done

  -I- Starting Driver                         -Done

  -I- Restarting MST                          -Done

  -I- FW was loaded successfully.
  ```
  ```

ECN operations parameters are located on the following path /sys/class/net/<interface>/ecn

Use the following command to find the interface:

- ```
  jvd@A100-01:~/$ ls /sys/class/net/

  docker0   enp14s0f1np1 enp17s0f1np1 enp44s0f1np1  gpu0_eth  gpu3_eth  gpu6_eth mgmt_eth

  enp139s0f1np1 enp169s0f0np0 enp200s0f1np1 enp81s0f1np1 gpu1_eth gpu4_eth gpu7_eth usb0

  enp142s0f1np1 enp169s0f1np1 enp203s0f1np1 enp82s0f1np1 gpu2_eth gpu5_eth lo

  jvd@A100-01:/sys/class/net/gpu0_eth/ecn$ ls

  roce_np  roce_rp
  ```

ECN bits on the IP header are always marked with 10 for RoCE traffic.

Notification Point (NP) Parameters

When the ECN-enabled receiver receives ECN-marked RoCE packets, it responds by sending CNP (Congestion Notification Packets).

The following commands describe the notification parameters:

- ```
  jvd@A100-01:/sys/class/net/gpu0_eth/ecn$ ls /roce_np/

  cnp_802p_prio  cnp_dscp  enable  min_time_between_cnps
  ```

Examples:

- ```
  jvd@A100-01:/sys/class/net/gpu0_eth/ecn$ cat roce_np/cnp_802p_prio

  6
  ```

cnp_802p_prio = the value of the PCP (Priority Code Point) field of the CNP packets.

PCP is a 3-bit field within an Ethernet frame header when using VLAN tagged frames as defined by IEEE 802.1Q.
  ```

- ```
  jvd@A100-01:/sys/class/net/gpu0_eth/ecn$ cat roce_np/cnp_dscp

  48
  ```

cnp_dscp = the value of the DSCP (Differentiated Services Code Point) field of the CNP packets.

- ```
  jvd@A100-01:/sys/class/net/gpu0_eth/ecn$ cat roce_np/min_time_between_cnps

  4
  ```

min_time_between_cnps = minimal time between two consecutive CNPs sent. if ECN-marked RoCE packet arrives in a period smaller than min_time_between_cnps since previous sent CNP, no CNP will be sent as a response. This value is in microseconds. Default = 0

- ```
  jvd@A100-01:/sys/class/net/gpu0_eth/ecn$ cat roce_np/enable/*

  1

  1

  1

  1

  1

  1

  1

  1
  ```

The output shows that roce_np is enabled for all priority values.

NOTE: Sending CNP packets is handled globally per port, any priority enabled here will set sending CNP packets to on (1).

To change the attributes described above, use the **mlxconfig** utility:

- ```
  mlxconfig -d /dev/mst/<mst_module> -y s CNP_DSCP_P1=<value> CNP_802P_PRIO_P1=<value>
  ```

Example:

- ```
  jvd@A100-01:/dev/mst$ sudo mst start

  Starting MST (Mellanox Software Tools) driver set

  Loading MST PCI module - Success

  [warn] mst_pciconf is already loaded, skipping

  Create devices

  Unloading MST PCI module (unused) - Success

  jvd@A100-01:~/scripts$ ./map_full_mellanox.sh



  Mellanox Device to mlx and Network Interface Mapping:

  /dev/mst/mt4123_pciconf0 => mlx5_14 => enp169s0f0np0 (0000:a9:00.0)

  /dev/mst/mt4125_pciconf0 => mlx5_4 => mgmt_eth (0000:2c:00.0)

  /dev/mst/
                  mt4129_pciconf0
                 => mlx5_6 => gpu0_eth (0000:0e:00.0)

  /dev/mst/mt4129_pciconf1 => mlx5_8 => gpu1_eth (0000:11:00.0)

  /dev/mst/mt4129_pciconf2 => mlx5_0 => gpu2_eth (0000:51:00.0)

  /dev/mst/mt4129_pciconf3 => mlx5_2 => gpu3_eth (0000:52:00.0)

  /dev/mst/mt4129_pciconf4 => mlx5_16 => gpu4_eth (0000:8b:00.0)

  /dev/mst/mt4129_pciconf5 => mlx5_18 => gpu5_eth (0000:8e:00.0)

  /dev/mst/mt4129_pciconf6 => mlx5_10 => gpu7_eth (0000:c8:00.0)

  /dev/mst/mt4129_pciconf7 => mlx5_12 => gpu6_eth (0000:cb:00.0)
  ```
  ```

```
jvd@A100-01:/sys/class/net/gpu0_eth/ecn$ sudo mlxconfig -d /dev/mst/
                mt4129_pciconf0
               -y set CNP_DSCP_P1=40 CNP_802P_PRIO_P1=7

Device #1:

----------

Device type:        ConnectX7

Name:               MCX755106AS-HEA_Ax

Description:        NVIDIA ConnectX-7 HHHL Adapter Card; 200GbE (default mode) / NDR200 IB;
Dual-port QSFP112; PCIe 5.0 x16 with x16 PCIe extension option; Crypto Disabled; Secure Boot
Enabled

Device:             /dev/mst/mt4129_pciconf0

Configurations:                                         Next Boot       New

        CNP_DSCP_P1                                 48                  40

        CNP_802P_PRIO_P1                            6                   7

Apply new Configuration? (y/n) [n] : y

Applying... Done!

-I- Please reboot machine to load new configurations.
```

Reaction Point (RP) Parameters

When the ECN-enabled sender receives CNP packets, it responds by slowing down transmission for the specified flows (priority).

The following parameters define how traffic flows will be rate limited, after CNP packets arrival:

- ```
  jvd@A100-01:/sys/class/net$ ls gpu0_eth/ecn/roce_rp/

  clamp_tgt_rate  enable  rpg_ai_rate     rpg_max_rate    rpg_time_reset
  ```
  ```

```
clamp_tgt_rate_after_time_inc   initial_alpha_value     rpg_byte_reset  rpg_min_dec_fac

dce_tcp_g               rate_reduce_monitor_period              rpg_gd

rpg_min_rate            dce_tcp_rtt     rate_to_set_on_first_cnp

rpg_hai_rate                    rpg_threshold
```

Examples:

- ```
  jvd@A100-01:/sys/class/net/gpu0_eth/ecn$ cat roce_rp/enable/*

  1

  1

  1

  1

  1

  1

  1

  1

  jvd@A100-01:/sys/class/net/gpu0_eth/ecn$ cat roce_rp/rpg_max_rate

  0
  ```

rpg_max_rate = Maximum rate at which reaction point node can transmit. Once this limit is reached, RP is no longer rate limited.

This value is configured in Mbits/sec. Default = 0 (full speed – no max)

The output shows that roce_rp is enabled for all priority values.

NOTE: Handling CNP is configured per priority.

To check the ECN statistics use: ethtool -S <interface> | grep ecn
  ```

Example:

- ```
  jvd@A100-01:~/scripts$ ethtool -S gpu0_eth | grep ecn

       rx_ecn_mark: 0

       rx_xsk_ecn_mark: 0

       rx0_ecn_mark: 0

       rx1_ecn_mark: 0

       rx2_ecn_mark: 0

       rx3_ecn_mark: 0

       rx4_ecn_mark: 0

       rx5_ecn_mark: 0

       rx6_ecn_mark: 0

       rx7_ecn_mark: 0

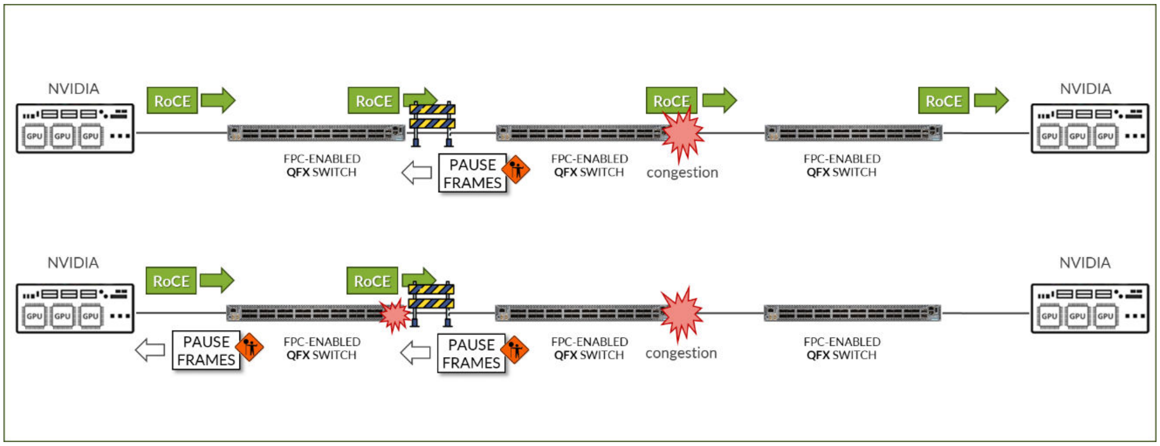
       rx8_ecn_mark: 0

   ---more---
  ```

NVIDIA DCQCN – PFC Configuration

IEEE 802.1Qbb applies pause functionality to specific classes of traffic on the Ethernet link.
  ```

Figure 63: NVIDIA DCQCN – PFC Configuration



To check whether PFC is enabled on an interface use: mlnx_qos -i <interface>

Example:

```
jvd@A100-01:/sys/class/net/gpu0_eth/ecn$ sudo mlnx_qos -i gpu0_eth

DCBX mode: OS controlled

Priority trust state: dscp

dscp2prio mapping:

        prio:0 dscp:07,06,05,04,03,02,01,00,

        prio:1 dscp:15,14,13,12,11,10,09,08,

        prio:2 dscp:23,22,21,20,19,18,17,16,

        prio:3 dscp:31,30,29,28,27,26,25,24,

        prio:4 dscp:39,38,37,36,35,34,33,32,

        prio:5 dscp:47,46,45,44,43,42,41,40,

        prio:6 dscp:55,54,53,52,51,50,49,48,

        prio:7 dscp:63,62,61,60,59,58,57,56,

default priority:
```

```
Receive buffer size (bytes): 19872,243072,0,0,0,0,0,0,max_buffer_size=2069280

Cable len: 7


                PFC configuration
              :

        priority    0   1   2
                3
                 4   5   6   7

        enabled     0   0   0
                1
                 0   0   0   0

        buffer      0   0   0
                1
                 0   0   0   0

tc: 0 ratelimit: unlimited, tsa: vendor

         priority:  1

tc: 1 ratelimit: unlimited, tsa: vendor

         priority:  0

tc: 2 ratelimit: unlimited, tsa: vendor

         priority:  2

tc: 3 ratelimit: unlimited, tsa: vendor

         priority:  3

tc: 4 ratelimit: unlimited, tsa: vendor

         priority:  4

tc: 5 ratelimit: unlimited, tsa: vendor

         priority:  5
```

```
tc: 6 ratelimit: unlimited, tsa: vendor

         priority:  6

tc: 7 ratelimit: unlimited, tsa: vendor

         priority:  7
```

To enable/disable PFC use: mlnx_qos -i <interface> --pfc <0/1>,<0/1>,<0/1>,<0/1>,<0/1>,<0/1>,<0/1>,<0/1>

Example:

- Check the current configuration:

- ```
  jvd@A100-01:/sys/class/net/gpu0_eth/ecn$ sudo mlnx_qos -i gpu0_eth

  DCBX mode: OS controlled

  Priority trust state: dscp

  dscp2prio mapping:

          prio:0 dscp:07,06,05,04,03,02,01,00,

          prio:1 dscp:15,14,13,12,11,10,09,08,

          prio:2 dscp:23,22,21,20,19,18,17,16,

          prio:3 dscp:31,30,29,28,27,26,25,24,

          prio:4 dscp:39,38,37,36,35,34,33,32,

          prio:5 dscp:47,46,45,44,43,42,41,40,

          prio:6 dscp:55,54,53,52,51,50,49,48,

          prio:7 dscp:63,62,61,60,59,58,57,56,

  default priority:

  Receive buffer size (bytes): 19872,243072,0,0,0,0,0,0,max_buffer_size=2069280
  ```
  ```

```
Cable len: 7

                PFC configuration
              :

        priority    0   1   2
                3
                 4   5   6   7

        enabled     0   0   0
                1
                 0   0   0   0

        buffer      0   0   0
                1
                 0   0   0   0

---more---
```

The output in the example, indicates that PFC is enable for Priority 3.

- Enable PFC for priority 2 and disable PFC for priority 3:

This example shows how to change the configuration; make sure it matches the PFC configuration on the leaf nodes (set class-of-service forwarding-classes class NO-LOSS pfc-priority 3).

- ```
  jvd@A100-01:~/scripts$ sudo mlnx_qos -i gpu0_eth --pfc 0,0,
                  1
                ,0,0,0,0,0

  DCBX mode: OS controlled

  Priority trust state: dscp

  dscp2prio mapping:

          prio:0 dscp:07,06,05,04,03,02,01,00,

          prio:1 dscp:15,14,13,12,11,10,09,08,

          prio:2 dscp:23,22,21,20,19,18,17,16,
  ```
  ```

```
        prio:3 dscp:31,30,29,28,27,26,25,24,

        prio:4 dscp:39,38,37,36,35,34,33,32,

        prio:5 dscp:47,46,45,44,43,42,41,40,

        prio:6 dscp:55,54,53,52,51,50,49,48,

        prio:7 dscp:63,62,61,60,59,58,57,56,

default priority:

Receive buffer size (bytes): 19872,243072,0,0,0,0,0,0,max_buffer_size=2069280

Cable len: 7


                PFC configuration:


        priority    0   1
                2


                3
                 4   5   6   7

        enabled     0   0
                1


                0
                 0   0   0   0

        buffer      0   0
                1


                0
                 0   0   0   0
```

```
---more---
```

- Check PFC statistics:

```
jvd@A100-01:~/scripts$ ethtool -S gpu0_eth | grep pause

     rx_pause_ctrl_phy: 8143294

     tx_pause_ctrl_phy: 502


               rx_prio3
              _pause: 8143294


               rx_prio3
              _pause_duration: 10848932


               tx_prio3
              _pause: 502


               tx_prio3
              _pause_duration: 30445


               rx_prio3
              _pause_transition: 4071126

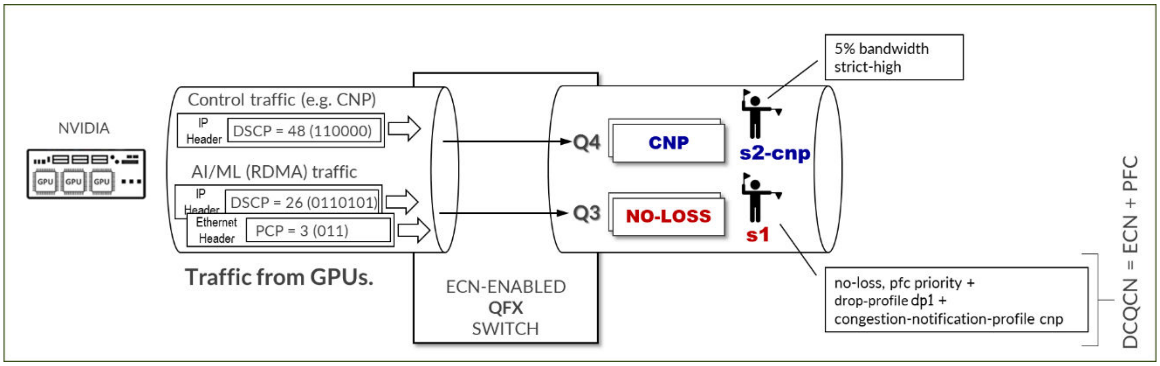
     tx_pause_storm_warning_events: 0

     tx_pause_storm_error_events: 0
```

NOTE: The Pause counters are visible via ethtool only for priorities on which PFC is enabled.

NVIDIA TOS/DSCP Configuration for RDMA-CM QPS (RDMA Traffic)

Figure 64: NVIDIA TOS/DSCP



RDMA traffic must be properly marked to allow the switch to correctly classify it, and to place it in the lossless queue for proper treatment. Marking can be either DSCP within the IP header, or PCP in the ethernet frame vlan-tag field. Whether DSCP or PCP is used depends on whether the interface between the GPU server and the switch is doing vlan tagging (802.1q) or not.

To check the current configuration and to change the values of TOS for the RDMA outbound traffic, use the **cma_roce_tos** script that is part of MLNX_OFED 4.0.

- ```
  jvd@A100-01:/sys/class/net/gpu0_eth/ecn$ sudo cma_roce_tos -h

  Set/Show RoCE default TOS of RDMA_CM applications

  Usage:

          cma_roce_tos OPTIONS

  Options:

          -h              show this help

          -d <dev>        use IB device <dev> (default mlx5_0)

          -p <port>       use port <port> of IB device (default 1)

          -t <TOS>        set TOS of RoCE RDMA_CM applications (0)
  ```

To check the current value of the TOS field enter sudo cma_roce_tos without any options.

Example:
  ```

- ```
  jvd@A100-01:/sys/class/net/gpu0_eth/ecn$ sudo cma_roce_tos

  106
  ```

In the example, the current TOS value = 106, which means a DSCP value = 48 and the ECN bits set to 10.

NOTE: The TOS field is 8 bits, while the DSCP is 6 bits. To set a DSCP value of X, you need to multiply this value by 4 (SHIFT 2). For example, to set DSCP value of 24, (24x4=96). Set the TOS bit to 96. You need to add 2 to include the ECN.

DSCP | ECN (RFC3168): ECT, CE

IP PRECEDENCE

	TYPE OF SERVICE FIELD (TOS) BINARY								TOS		DSCP		IP PRECEDENCE		
									DECIMAL VALUE	HEX VALUE	DECIMAL VALUE	HEX VALUE	DECIMAL VALUE	HEX VALUE	NAME
CNP	1	1	0	0	0	0	1	0	194	0xC2	48	0x30	6	0x6	Internetwork Control
NO-LOSS	0	1	1	0	1	0	1	0	106	0x6A	26	0x1A	3	0x3	Flash

CLASS | DROP PROB

To change the value use: cma_roce_tos –d <ib_device> -t <TOS>

You need to enter the ib_device in this command. The following script automatically does the mapping between the physical interfaces and the ib_device.

- ```
  map_full_mellanox.sh

  #!/bin/bash

  # Script to map Mellanox devices to mlx and network interfaces

  # Get Mellanox device PCI addresses

  mst_status=$(sudo mst status | awk '

  /\/dev\/mst/ {

      dev = $1

  }
  ```

```
/domain:bus:dev.fn/ {
    pci = $1
    printf "%s: %s\n", dev, pci
}
')
# Get network interface PCI addresses
iface_status=$(for iface in $(ls /sys/class/net/); do
    pci_addr=$(ethtool -i $iface 2>/dev/null | grep bus-info | awk '{print $2}')
    if [ ! -z "$pci_addr" ]; then
        echo "$iface: $pci_addr"
    fi
done)
# Get network interface to mlx interface mapping
mlx_iface_status=$(for iface in $(ls /sys/class/net/); do
    if [ -d /sys/class/net/$iface/device/infiniband_verbs ]; then
        mlx_iface=$(cat /sys/class/net/$iface/device/infiniband_verbs/*/ibdev)
        echo "$iface: $mlx_iface"
    fi
done)
# Combine and print the mapping
echo "Mellanox Device to mlx and Network Interface Mapping:"
```

```
echo "$mst_status" | while read -r mst_line; do

    mst_dev=$(echo $mst_line | awk -F ': ' '{print $1}')

    mst_pci=$(echo $mst_line | awk -F '=| ' '{print $3}')

    iface=$(echo "$iface_status" | grep $mst_pci | awk -F ': ' '{print $1}')

    iface_pci=$(echo "$iface_status" | grep $mst_pci | awk -F ': ' '{print $2}')

    mlx_iface=$(echo "$mlx_iface_status" | grep $iface | awk -F ': ' '{print $2}')

    if [ ! -z "$iface" ] && [ ! -z "$mlx_iface" ]; then

        echo "$mst_dev => $mlx_iface => $iface ($iface_pci)"

    fi

done
```

Example:

Figure 65. script results example

```
jvd@A100-01:~/scripts$ ./map_full_mellanox.sh

Mellanox Device to mlx and Network Interface Mapping:
/dev/mst/mt4123_pciconf0 => mlx5_14 => enp169s0f0np0 (0000:a9:00.0)
/dev/mst/mt4125_pciconf0 => mlx5_4 => mgmt_eth (0000:2c:00.0)
/dev/mst/mt4129_pciconf0 => mlx5_6 => GPU0_eth (0000:0e:00.0)
/dev/mst/mt4129_pciconf1 => mlx5_8 => GPU1_eth (0000:11:00.0)
/dev/mst/mt4129_pciconf2 => mlx5_0 => GPU2_eth (0000:51:00.0)
/dev/mst/mt4129_pciconf3 => mlx5_2 => GPU3_eth (0000:52:00.0)
/dev/mst/mt4129_pciconf4 => mlx5_16 => GPU4_eth (0000:8b:00.0)
/dev/mst/mt4129_pciconf5 => mlx5_18 => GPU5_eth (0000:8e:00.0)
/dev/mst/mt4129_pciconf6 => mlx5_10 => GPU7_eth (0000:c8:00.0)
/dev/mst/mt4129_pciconf7 => mlx5_12 => GPU6_eth (0000:cb:00.0)

jvd@A100-01:~/scripts$ cma_roce_tos -d mlx5_6 -t 194
194

jvd@A100-01:~/scripts$ cma_roce_tos -d mlx5_6
194
```

Figure 66. Reference TOS, DSCP Mappings:

	DSCP						ECN					
	IP PRECEDENCE											
	TYPE OF SERVICE FIELD (TOS) BINARY								TOS		DSCP	
									DECIMAL VALUE	HEX VALUE	DECIMAL VALUE	HEX VALUE
	0	0	0	0	0	0	0	0	0	0x0	0	0x0
	0	0	1	0	0	0	0	0	32	0x20	8	0x8
	0	1	0	0	0	0	0	0	64	0x40	16	0x10
	0	1	1	0	0	0	0	0	96	0x60	24	0x18
	1	0	0	0	0	0	0	0	128	0x80	32	0x20
	1	0	1	0	0	0	0	0	160	0xA0	40	0x28
CNP	**1**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**192**	**0xC0**	**48**	**0x30**
	1	1	1	0	0	0	0	0	224	0xE0	56	0x38
	0	0	1	0	1	0	0	0	40	0x28	10	0xA
	0	0	1	1	0	0	0	0	48	0x30	12	0xC
	0	0	1	1	1	0	0	0	56	0x38	14	0xE
	0	1	0	0	1	0	0	0	72	0x48	18	0x12
	0	1	0	1	0	0	0	0	80	0x50	20	0x14
	0	1	0	1	1	0	0	0	88	0x58	22	0x16
NO-LOSS	**0**	**1**	**1**	**0**	**1**	**0**	**0**	**0**	**104**	**0x68**	**26**	**0x1A**
	0	1	1	1	0	0	0	0	112	0x70	28	0x1C
	0	1	1	1	1	0	0	0	120	0x78	30	0x1E
	1	0	0	0	1	0	0	0	136	0x88	34	0x22
	1	0	0	1	0	0	0	0	144	0x90	36	0x24
	1	0	0	1	1	0	0	0	152	0x98	38	0x26
	1	0	1	1	1	0	0	0	184	0xB8	46	0x2E
	CLASS											
				DROP PROB								

Configuring NVIDIA to Use the Management Interface for NCCL Control Traffic

NCCL uses TCP sessions to connect processes together and exchange QP information for RoCE, GIDs (Global IDs), Local and remote buffer addresses, RDMA keys (RKEYs for memory access permissions)

These are separate to the RoCEv2 traffic (port 4791) used for synchronizing model parameters, partial results operations, and so on.

These sessions are created when the job starts and by default use one of the GPU interfaces (same interfaces used for RoCEv2 traffic).

Example:

- ```
  ylara@A100-01:~$ netstat -atn | grep 10.200 | grep "ESTABLISHED"

  tcp        0      0 10.200.4.8:47932        10.200.4.2:43131        ESTABLISHED

  tcp        0      0 10.200.4.8:46699        10.200.4.2:37236        ESTABLISHED
  ```
  ```

```
tcp        0      0 10.200.2.8:60502        10.200.13.2:35547       ESTABLISHED

tcp        0      0 10.200.4.8:37330        10.200.4.2:55355        ESTABLISHED

tcp        0      0 10.200.4.8:56438        10.200.4.2:53947        ESTABLISHED

---more---
```

It is recommended, move to the management interface (connected to the (Frontend Fabric) including the following parameter when starting a job: **export NCCL_SOCKET_IFNAME="mgmt_eth"**

Example:

- ```
  ylara@A100-01:~$ netstat -atn | grep 10.10.1 | grep "ESTABLISHED"

  tcp        0      0 10.10.1.0:44926         10.10.1.2:33149         ESTABLISHED

  tcp        0      0 10.10.1.0:46705         10.10.1.0:40320         ESTABLISHED

  tcp        0      0 10.10.1.0:54661         10.10.1.10:52452        ESTABLISHED

  ---more---
  ```

ECN is enabled by default for these sessions; *net.ipv4.tcp_ecn = 1* , but can be disable with: *sudo sysctl -w net.ipv4.tcp_ecn=0*

Terraform Automation of Apstra for the AI Fabric

IN THIS SECTION
  ```

AI Terraform Configs

Juniper has compiled a set of Terraform configs to help set up data center fabrics for an AI cluster. AI training requires a dedicated GPU Backend fabric, a dedicated Storage Backend fabric, and a Frontend fabric. Here we show such Apstra-managed network fabrics deploying logical devices, racks and templates for DGX (or HGX equivalent) servers based on A100 and H100 GPUs having 200GE and 400GE access connectivity respectively. The logical devices, racks and templates defined here create the NVIDIA Rail-optimized topology.

The github repository for AI designs using Apstra can be found:

https://github.com/Juniper/terraform-apstra-examples/tree/master/ai-cluster-designs/

AI JVD Specific Terraform Configs

Based on the AI cluster designs with rail-optimized GPU fabrics of various sizes, this Terraform config for Apstra will build a set of 3 blueprints for a reference AI cluster's dedicated GPU Backend fabric, a dedicated Storage Backend fabric, and a Frontend fabric.

This example shall serve as a Juniper Validated Design (JVD) set of configurations that can be applied to larger clusters. It has two NVIDIA rail-optimized groups with Juniper QFX5220 leaf switches in one stripe of 8 and QFX5230 leaf switches in another stripe of 8. It has options for both QFX5230 spines or high-radix PTX10008 spines, with examples here for A100s and H100-based servers in uniform racks or as deployed in the "Lab Leaf" rack with mixed server access for half A100 and half H100 connectivity to serve as an example, and because that is what is used in the real lab test environment for this configuration.

The github repository for this specific AI JVD can be found:

https://github.com/Juniper/terraform-apstra-examples/tree/master/ai-cluster-jvd/

Figure 67: Sample GPU Backend Terraform Template

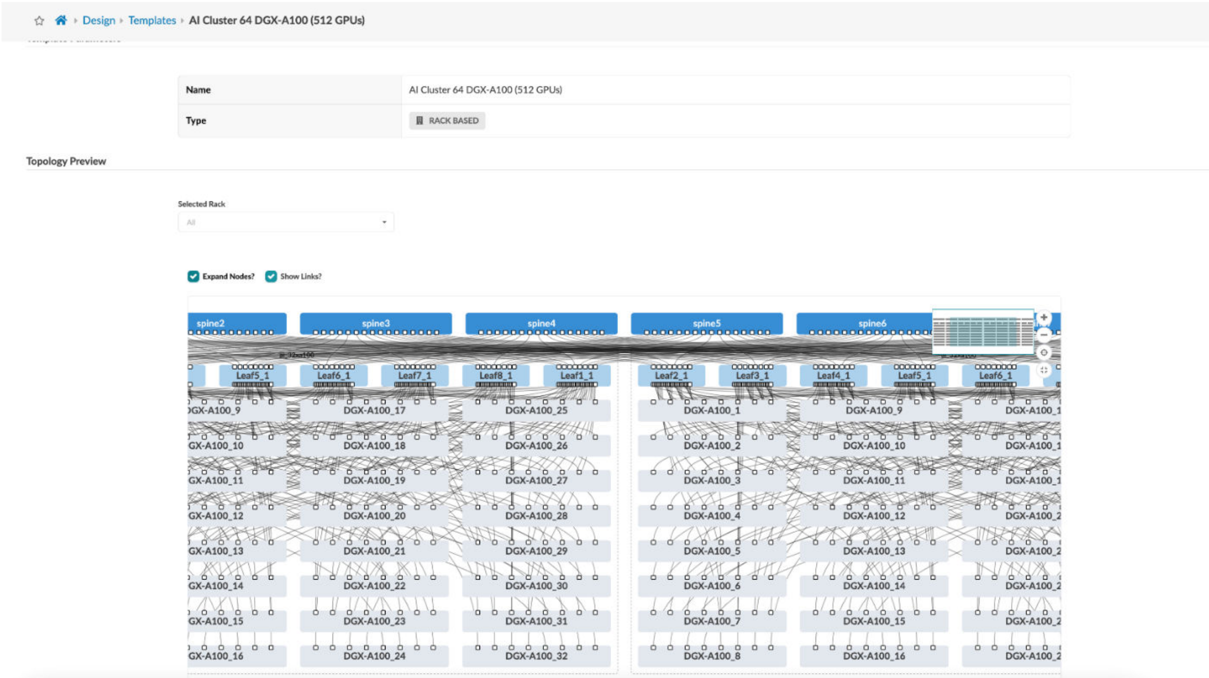


Figure 68: Sample GPU Backend Terraform Template: Rack Type

Figure 69: Sample GPU Backend Terraform Template: Logical Device

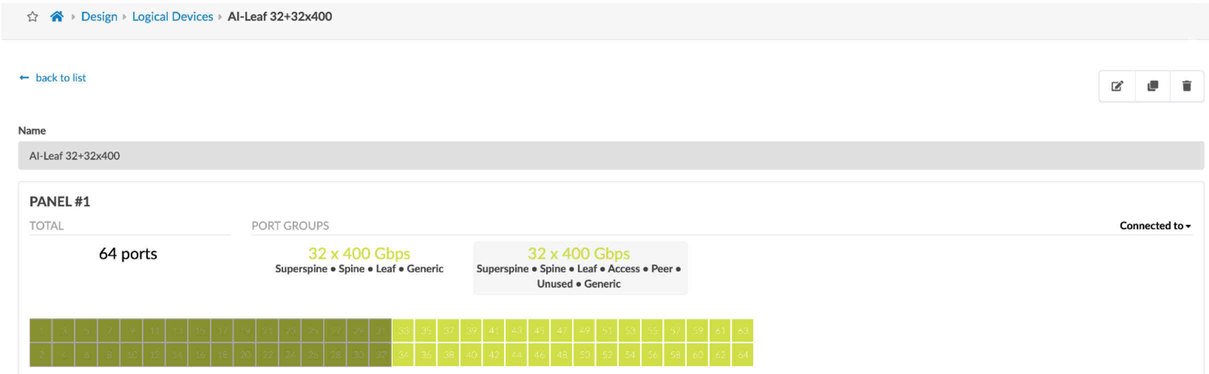
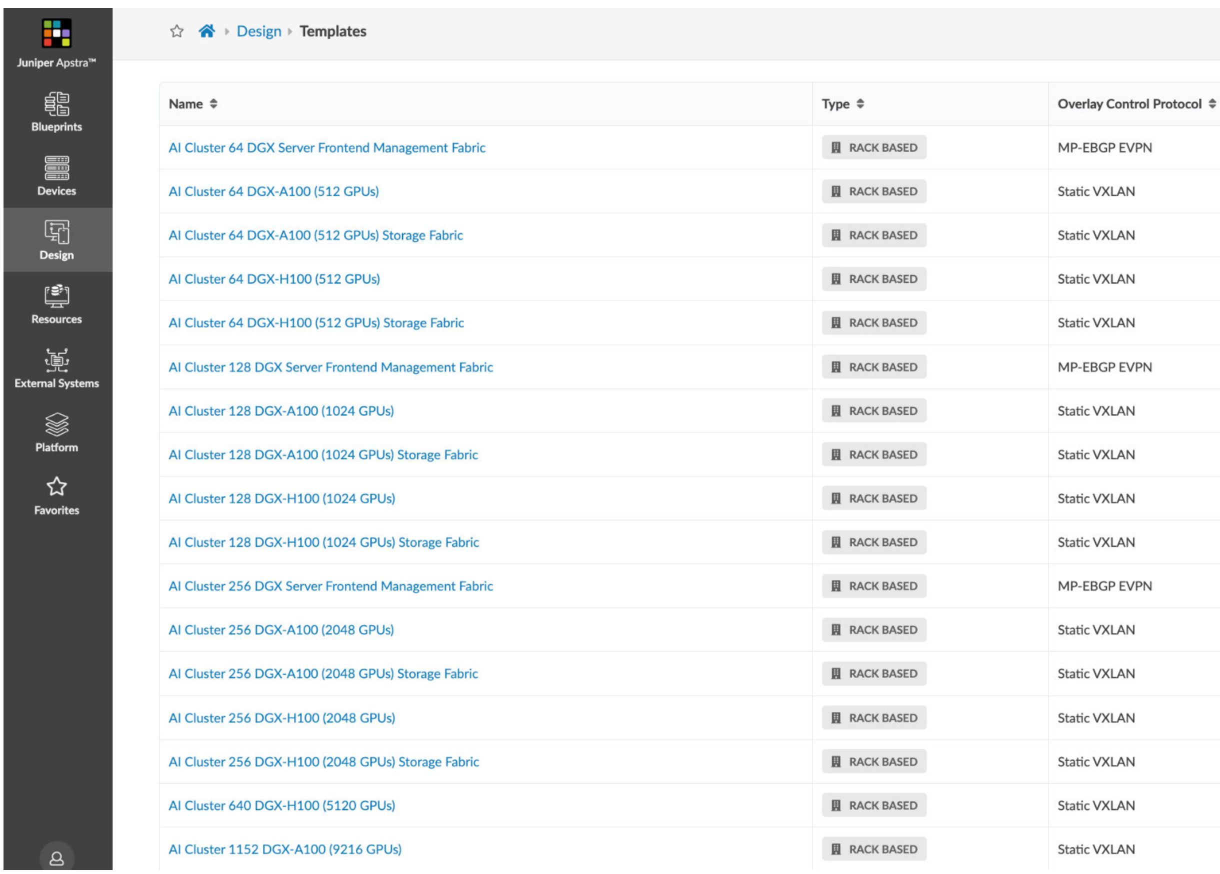


Figure 70: Terraform Template: All Templates Examples



Name	Type	Overlay Control Protocol
AI Cluster 64 DGX Server Frontend Management Fabric	RACK BASED	MP-EBGP EVPN
AI Cluster 64 DGX-A100 (512 GPUs)	RACK BASED	Static VXLAN
AI Cluster 64 DGX-A100 (512 GPUs) Storage Fabric	RACK BASED	Static VXLAN
AI Cluster 64 DGX-H100 (512 GPUs)	RACK BASED	Static VXLAN
AI Cluster 64 DGX-H100 (512 GPUs) Storage Fabric	RACK BASED	Static VXLAN
AI Cluster 128 DGX Server Frontend Management Fabric	RACK BASED	MP-EBGP EVPN
AI Cluster 128 DGX-A100 (1024 GPUs)	RACK BASED	Static VXLAN
AI Cluster 128 DGX-A100 (1024 GPUs) Storage Fabric	RACK BASED	Static VXLAN
AI Cluster 128 DGX-H100 (1024 GPUs)	RACK BASED	Static VXLAN
AI Cluster 128 DGX-H100 (1024 GPUs) Storage Fabric	RACK BASED	Static VXLAN
AI Cluster 256 DGX Server Frontend Management Fabric	RACK BASED	MP-EBGP EVPN
AI Cluster 256 DGX-A100 (2048 GPUs)	RACK BASED	Static VXLAN
AI Cluster 256 DGX-A100 (2048 GPUs) Storage Fabric	RACK BASED	Static VXLAN
AI Cluster 256 DGX-H100 (2048 GPUs)	RACK BASED	Static VXLAN
AI Cluster 256 DGX-H100 (2048 GPUs) Storage Fabric	RACK BASED	Static VXLAN
AI Cluster 640 DGX-H100 (5120 GPUs)	RACK BASED	Static VXLAN
AI Cluster 1152 DGX-A100 (9216 GPUs)	RACK BASED	Static VXLAN

Validation Framework

IN THIS SECTION

Platforms / Devices Under Test (DUT)

Table 25: Platforms / Devices Under Test (DUT)

Component	Frontend	Storage Backend	GPU Backend (Cluster 1 and 2)
Architecture	3-stage clos	3-stage clos	3-stage clos rail optimized
Spine nodes	QFX5130-32CD x 2	QFX5220-32CD x 2	QFX5230-64CD x 2 (cluster 1) PTX-10008 JNP10K-LC1201 (cluster 1) QFX5240-64OD x 2 (cluster 2) QFX5241-64OD x 2 (cluster 2)
Leaf nodes	QFX5130-32CD x 1 *(frontend-gpu-leaf)* QFX5130-32CD x 1 *(frontend-weka-leaf)*	QFX5220-32CD x 2 *(storage-backend-gpu-leaf)* QFX5220-32CD x 2 *(storage-backend-weka-leaf)*	QFX5220-64CD x 8 (cluster 1 – stripe 1) QFX5230-64CD x 8 (cluster 1 – stripe 2) QFX5240-64CD x 8 (cluster 2 – stripes 1-2) QFX5241-64OD x 2 (cluster 2)

(Continued)

Component	Frontend	Storage Backend	GPU Backend (Cluster 1 and 2)
Leaf nodes <=> spine node links	2 x 400GE (per *frontend-leaf* <=> *frontend-spine* link)	2 x 400GE (per *storage-backend-weka-leaf* <=> *storage-backend-spine*) 3 x 400GE (per *storage-backend-gpu-leaf* <=> *storage-backend-spine*)	2 x 400GE (per *gpu-backend-spine* <=> *gpu-backend-leaf* link)
Number of NVIDIA DGX H100 GPU servers	2 (Cluster 2 - stripe 1) 2 (Cluster 2 - stripe 2)		
Number of NVIDIA HGX A100 GPU servers	4 (Cluster 1 - stripe 1) 4 (Cluster 1 - stripe 1)		
NVIDIA DGX H100 GPU servers <=> GPU leaf nodes links	1 x 100GE (per *gpu server* <=> *frontend-gpu-leaf* link)	1 x 200GE (per gpu server <=> *storage-backend-gpu-leaf* link)	1 x 400GE (Cluster 2) (per gpu server <=> *gpu-backend-leaf* link)
NVIDIA HGX A100 GPU servers <=> GPU leaf nodes links	1 x 100GE (per *gpu server* <=> *frontend-gpu-leaf* link)	1 x 100GE (per *gpu server* <=> *storage-backend-gpu-leaf* link)	1 x 200GE (Cluster 1) (per *gpu server* <=> *gpu-backend-leaf* link)

(Continued)

Component	Frontend	Storage Backend	GPU Backend (Cluster 1 and 2)
Total number of GPUs	96: 32 x stripe in cluster 1 16 x stripe in cluster 2		
WEKA storage servers	8		
WEKA storage servers <=> WEKA storage leaf nodes links	1 x 100GE (per weka server <=> *frontend-weka-leaf* link)	1 x 200GE (per weka server <=> *storage-backend-weka-leaf* link)	N/A

Network Connectivity: Reference Examples

IN THIS SECTION

For those who want more details, this section provides insight into the setup of each fabric and the expected values for the reference examples.

The section describes the IP connectivity across the common Frontend, and Storage Backend fabrics, and the GPU Backend fabric in Cluster 1, Stripe 1. The GPU Backend fabrics for cluster 1, stripe 2, and cluster 2 follow the same model.

Regardless of whether you are using Apstra with or without Terraform automation **with** Apstra, the IP addressing Pools, ASN Pools, and interface addresses are largely automatically assigned and configured with little interaction from the administrator unless desired.

Notice that all the addresses shown in this section represent the IP addressing schema used in the Juniper lab to validate the design.

Frontend Network Connectivity

The Frontend fabric is designed as a Layer 3 IP Fabric, where the links between the leaf and spine nodes are configured with /31 IP addresses, as shown in Table 26. The fabric consists of 2 spine nodes and 2 leaf nodes, where 1 leaf node is used to connect to the storage servers (named ***frontend-weka-leaf*** *1)* and 1 is used to connect to the GPU servers (named *frontend-ai-leaf1).* Additionally, the Headend Servers that execute the workload manager (Slurm) for AI Training and Inference models reside in this fabric.

In this example, leaf nodes connecting to the GPU servers in the Frontend fabric are named ***frontend-ai-leaf#*** *instead of* ***frontend-gpu-leaf#*** *but they represent the same role.*

There are two 400GE links between each ***frontend-weka-leaf*** *1* node and the spine nodes and two 400GE links between each *frontend-ai-leaf1* node and the spine nodes as shown in Figure 71.

Figure 71: Frontend Spine to Leaf Nodes Connectivity

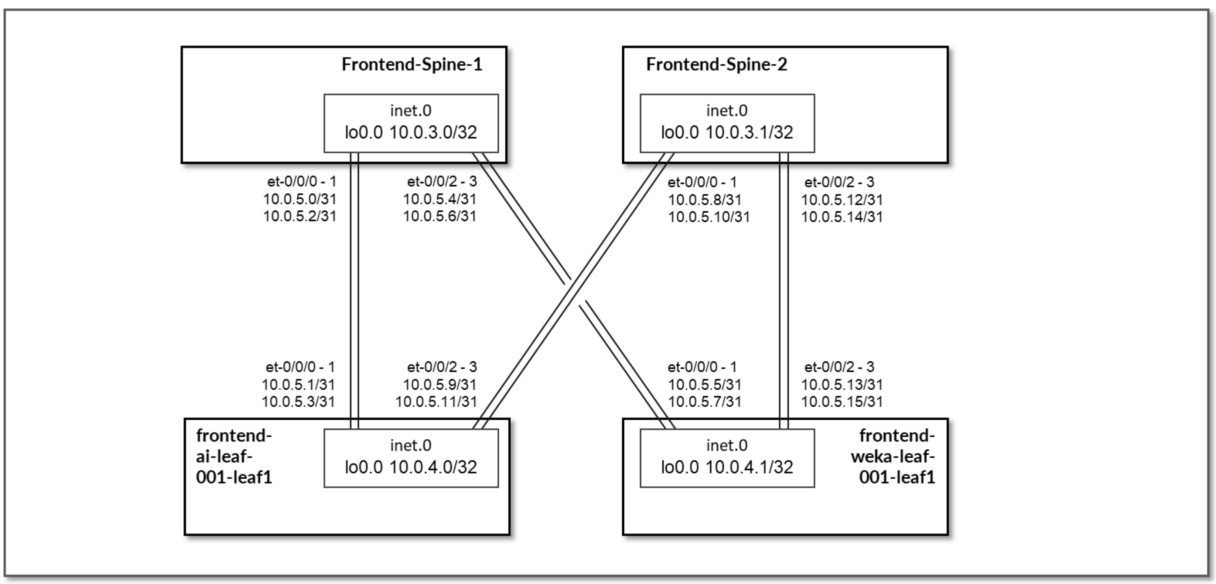


Table 26: Frontend Interface Addresses

Spine node	Leaf node	Spine IP address	Leaf IP address
frontend-spine1	*frontend-ai-leaf1*	10.0.5.0/31 10.0.5.2/31	10.0.5.1/31 10.0.5.3/31
frontend-spine1	*frontend-weka-leaf1*	10.0.5.4/31 10.0.5.6/31	10.0.5.5/31 10.0.5.7/31
frontend-spine2	*frontend-ai-leaf1*	10.0.5.8/31 10.0.5.10/31	10.0.5.9/31 10.0.5.11/31
frontend-spine2	*frontend-weka-leaf1*	10.0.5.12/31 10.0.5.14/31	10.0.5.13/31 10.0.5.15/31

NOTE: all the Autonomous System and IP addresses are assigned by Apstra (from predefined pools of resources) based on the intent.

The loopback interfaces also have addresses automatically assigned by Apstra from a predefined pool.

Table 27: Frontend Loopback Addresses

Device	Loopback interface address
frontend-spine1	10.0.3.0/32
frontend-spine2	10.0.3.1/32
frontend-ai-leaf1	10.0.1.0/32
frontend-weka-leaf1	10.0.1.1/32

The H100 GPU Servers and A100 GPU Servers are all connected to the frontend-ai-leaf1 node.

The links between the GPU servers and the leaf node Leaf 1 are assigned /31 subnets out of 10.0.5.0/24, shown in Figure 72 and Table 28.

Figure 72: Frontend Leaf Nodes to GPU Servers Connectivity

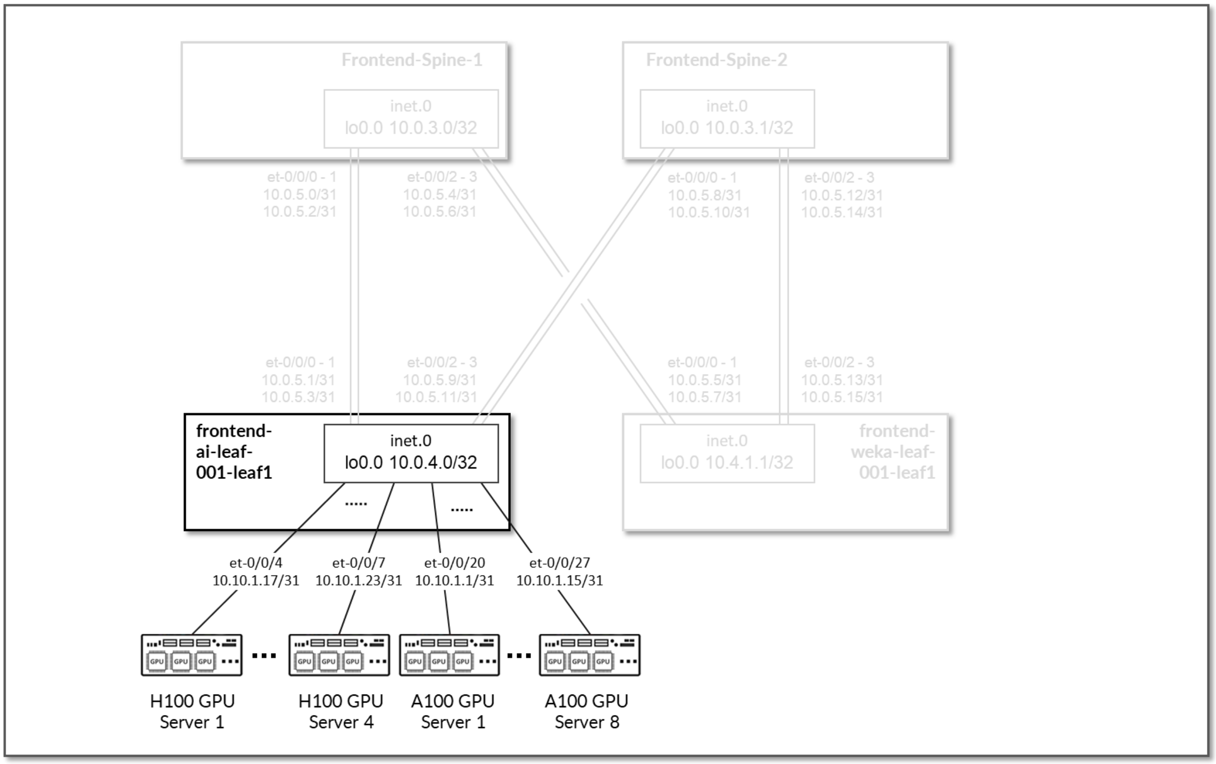


Table 28: Frontend Leaf Nodes to GPU Servers Interfaces Addresses

GPU Server	Leaf node	GPU Server IP address	Leaf IP address
H100 GPU Server 1	*frontend-ai-leaf1*	10.10.1.17/31	10.100.1.9/31
H100 GPU Server 2		10.10.1.19/31	10.100.1.11/31
H100 GPU Server 3		10.10.1.21/31	10.100.1.1/31
H100 GPU Server 4		10.10.1.23/31	10.100.1.3/31
A100 GPU Server 1		10.10.1.1/31	10.100.1.5/31
A100 GPU Server 2		10.10.1.3/31	10.100.1.7/31
A100 GPU Server 3		10.10.1.5/31	10.100.2.9/31
A100 GPU Server 4		10.10.1.7/31	10.100.2.11/31

(Continued)

GPU Server	Leaf node	GPU Server IP address	Leaf IP address
A100 GPU Server 5		10.10.1.9/31	10.100.2.1/31
A100 GPU Server 6		10.10.1.11/31	10.100.2.3/31
A100 GPU Server 7		10.10.1.13/31	10.100.2.5/31
A100 GPU Server 8		10.10.1.15/31	10.100.2.7/31

The WEKA storage servers are all connected to the *frontend-weka-leaf* 1 node.

The links to these servers do not have IP addresses assigned on the leaf node. Layer 3 connectivity is provided via an irb interface with an address out of subnet 10.10.2.1/24. The WEKA servers are assigned addresses out of 10.10.2.0/24, as shown Figure 73 and Table 29.

Figure 73: Frontend Leaf Nodes to WEKA Storage Connectivity

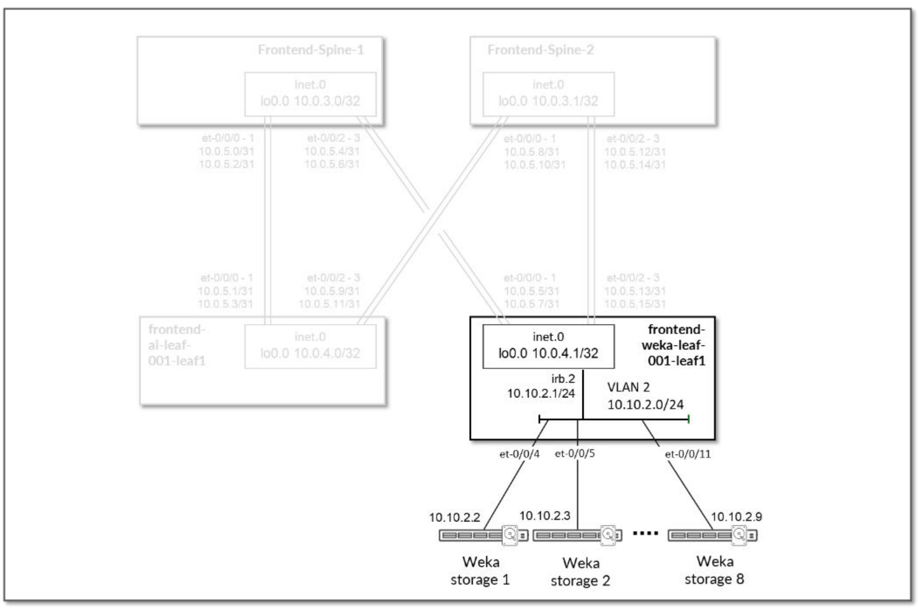


Table 29: Frontend Leaf Nodes to WEKA Storage Interface Addresses

GPU Server	Leaf node	WEKA Server IP Address	Leaf IP Address
WEKA Storage Server 1	*frontend-weka-leaf1*	10.10.2.2/24	10.10.2.1/24 (irb.2)
WEKA Storage Server 2		10.10.2.3/24	

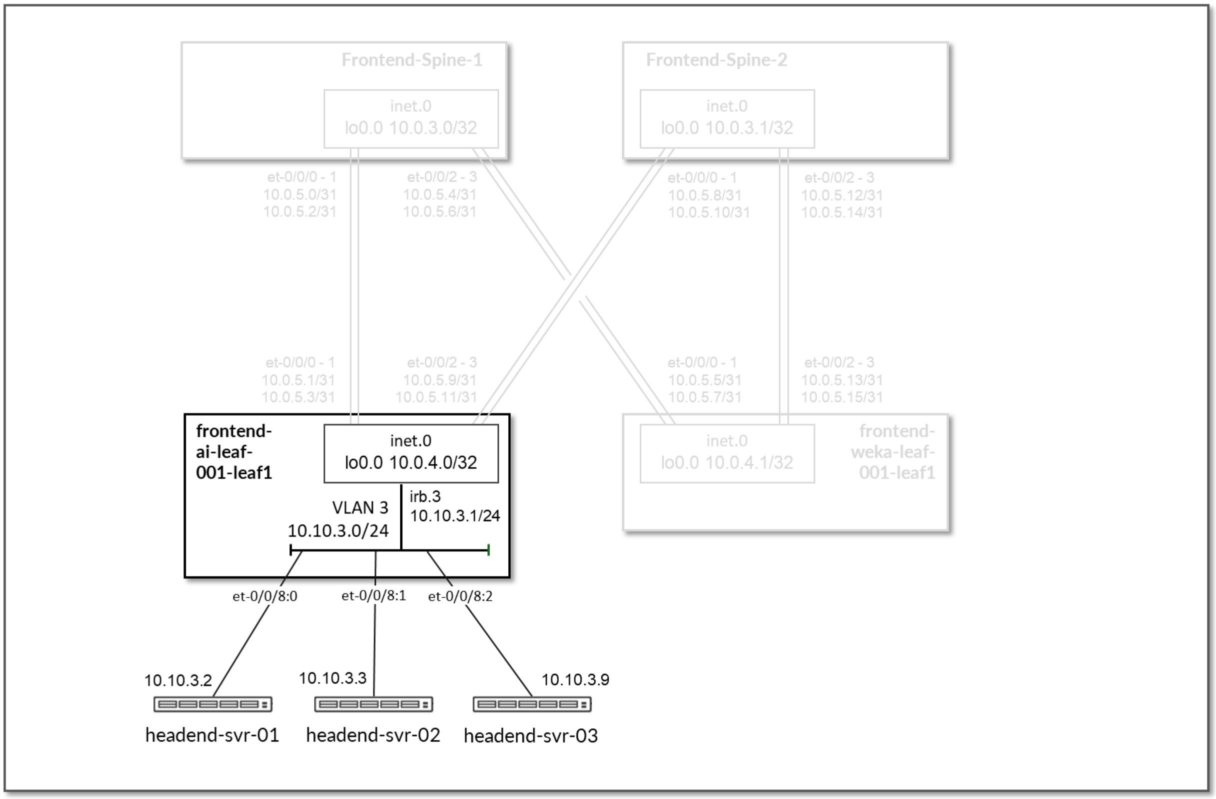
(Continued)

GPU Server	Leaf node	WEKA Server IP Address	Leaf IP Address
WEKA Storage Server 3		10.10.2.4/24	
WEKA Storage Server 4		10.10.2.5/24	
WEKA Storage Server 5		10.10.2.6/24	
WEKA Storage Server 6		10.10.2.7/24	
WEKA Storage Server 7		10.10.2.8/24	
WEKA Storage Server 8		10.10.2.9/24	

The Headend servers executing the workload manager are all connected to the frontend-ai-leaf1 node.

The links to these servers do not have IP addresses assigned on the leaf node. Layer 3 connectivity is provided via an irb interface with the address 10.10.3.1/24. The headend servers assigned addresses out of 10.10.3.0/24, as shown in Figure 74 and table below.

Figure 74: Frontend Leaf Nodes to Headend Servers Connectivity



EBGP is configured between the IP addresses assigned to the spine-leaf nodes links. There will be 2 EBGP sessions between the *frontend-ai-leaf#* node and each spine node, and 2 EBGP sessions between each ***frontend-weka-leaf #*** node and each of the spine nodes, as shown in Figure 75.

Figure 75: Frontend EBGP

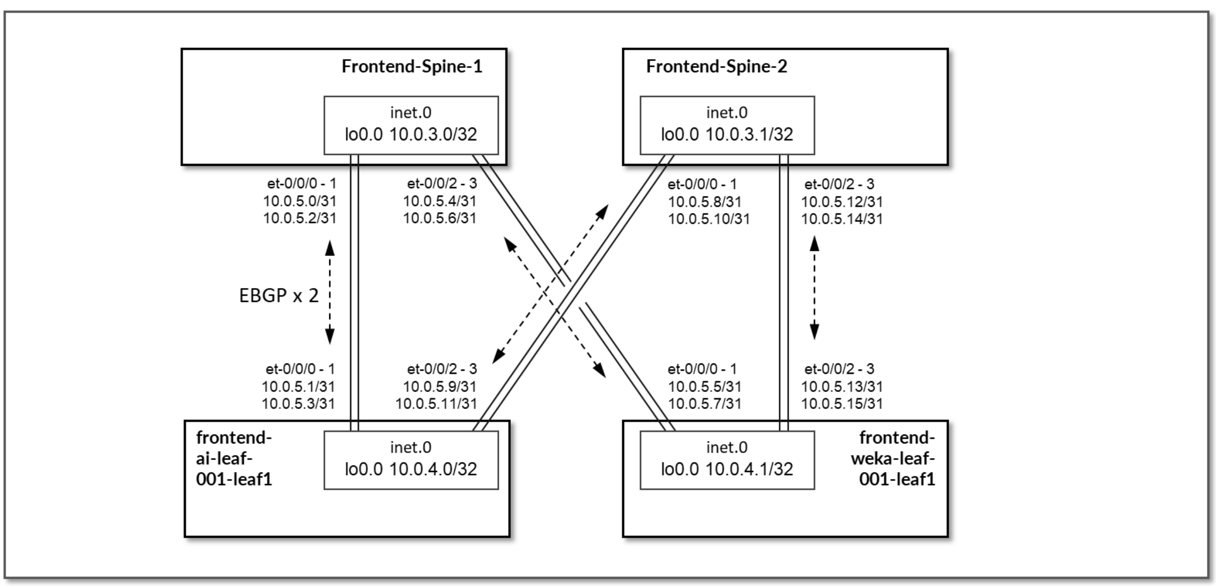


Table 30: Frontend Sessions

Spine node	Leaf node	Spine	Leaf ASN	Spine IP address	Leaf IP address
frontend-spine1	*frontend-ai-leaf1*	4201032300	4201032400	10.0.5.0/31 10.0.5.2/31	10.0.5.1/31 10.0.5.3/31
frontend-spine1	*frontend-weka-leaf* 1		4201032401	10.0.5.4/31 10.0.5.6/31	10.0.5.4/31 10.0.5.7/31
frontend-spine2	*frontend-ai-leaf1*	4201032301	4201032400	10.0.5.8/31 10.0.5.10/31	10.0.5.9/31 10.0.5.11/31
frontend-spine2	*frontend-weka-leaf* 1		4201032401	10.0.5.12/31 10.0.5.14/31	10.0.5.13/31 10.0.5.15/31

On the frontend-ai-leaf1 nodes BGP policies are configured by Apstra to advertise the following routes to the spine nodes:

NOTE: all the Autonomous System and community values are assigned by Apstra (from predefined pools of resources) based on the intent.

- frontend-ai-leaf1 node own loopback interface address,
- frontend-ai-leaf1 node to spines interfaces subnets and
- GPU servers to frontend-ai-leaf1 node link subnets.
- WEKA server's management subnet

Figure 76: Frontend Leaf to GPU Servers BGP

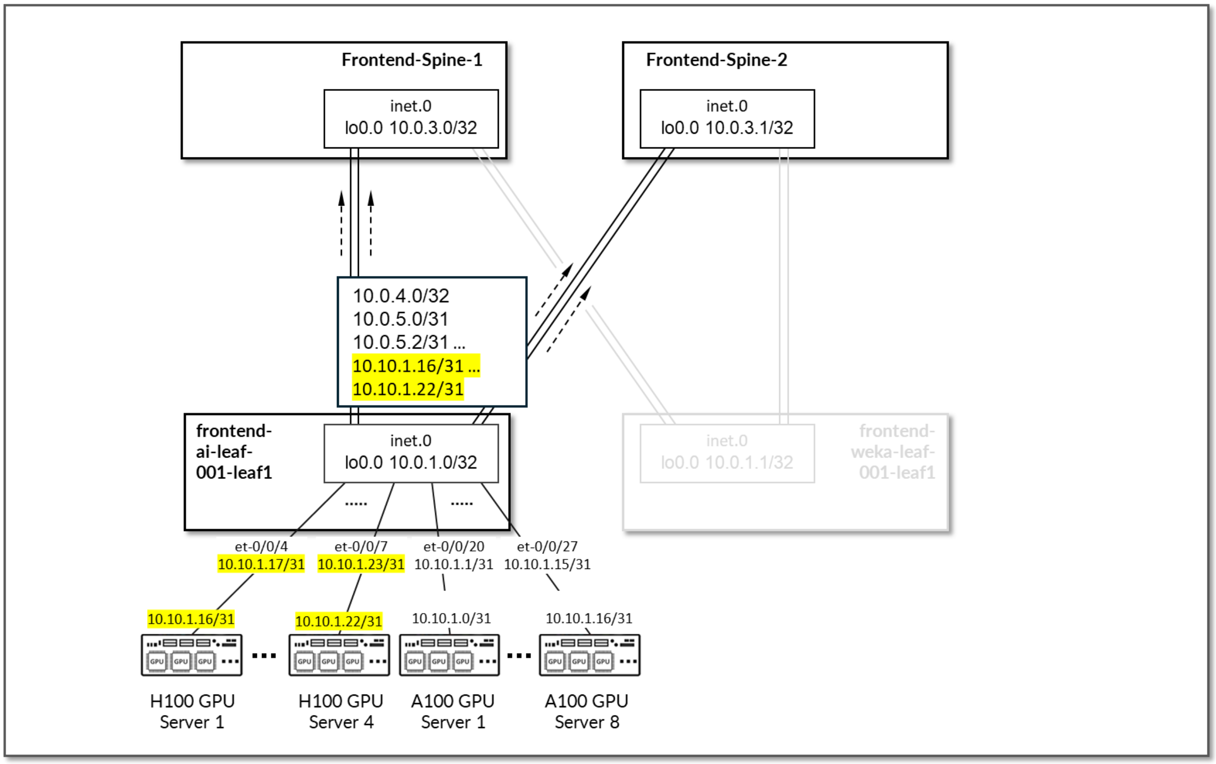


Figure 77: Frontend Leaf to Headend Server BGP

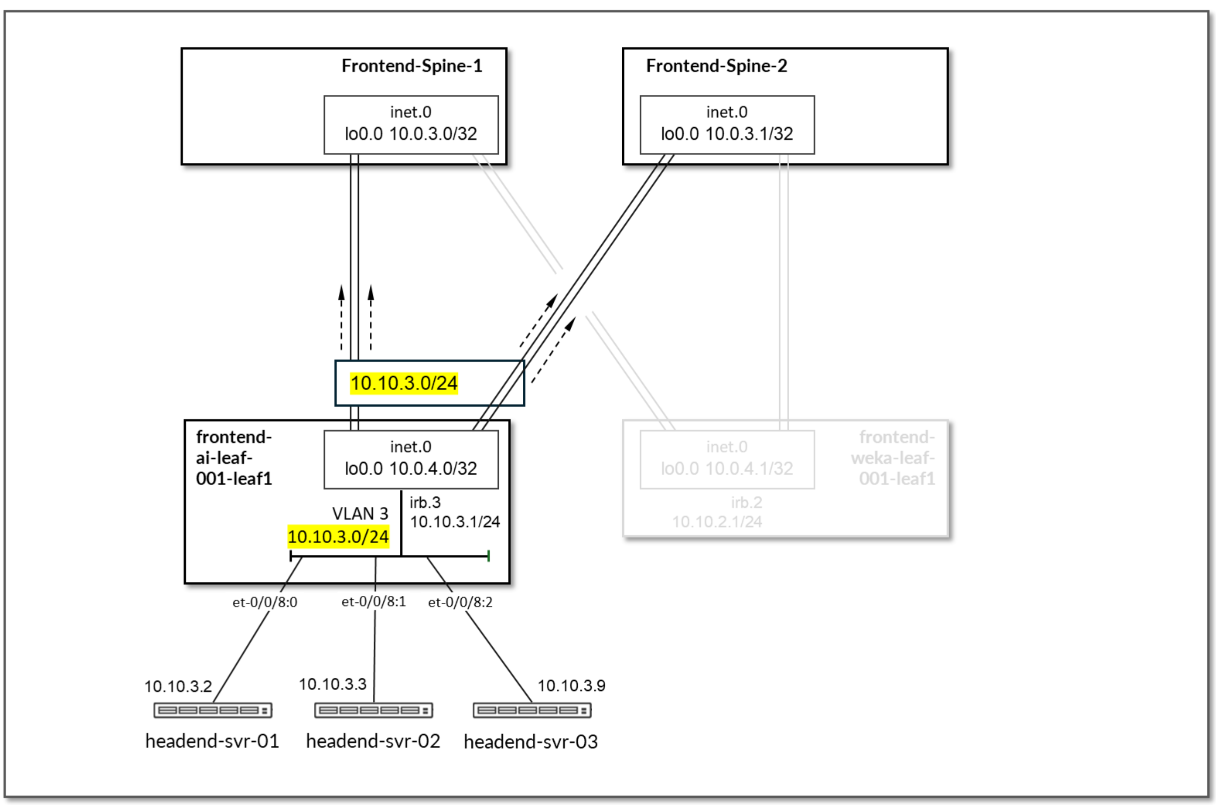


Table 31: Frontend Leaf to GPU/Headend Servers Advertised Routes

Leaf Node	Peer(s)	Advertised Routes		BGP Communities
frontend-ai-leaf1	*frontend-spine1 & frontend-spine2*	Loopback: 10.0.4.0/32 Leaf-spines links: 10.0.5.0/31 10.0.5.2/31 10.0.5.8/31 10.0.5.10/31	GPU servers <=> frontend spine links: 10.10.1.16/31 10.10.1.18/31 10.10.1.20/31 10.10.1.22/31 10.10.1.0/31 10.10.1.2/31 10.10.1.4/31 10.10.1.6/31 10.10.1.8/31 10.10.1.10/31 10.10.1.12/31 10.10.1.14/31 WEKA Management server’s subnet: 10.10.3.0/24	3:20007 21001:26000

On the *frontend-weka-leaf* 1 node BGP policies are configured by Apstra to advertise the following routes to the spine nodes:

- *frontend-weka-leaf* 1 node own loopback interface address,
- *frontend-weka-leaf* 1 node to spines interfaces subnets and
- WEKA storage server’s subnet

Figure 78: Frontend Leaf to WEKA Storage BGP

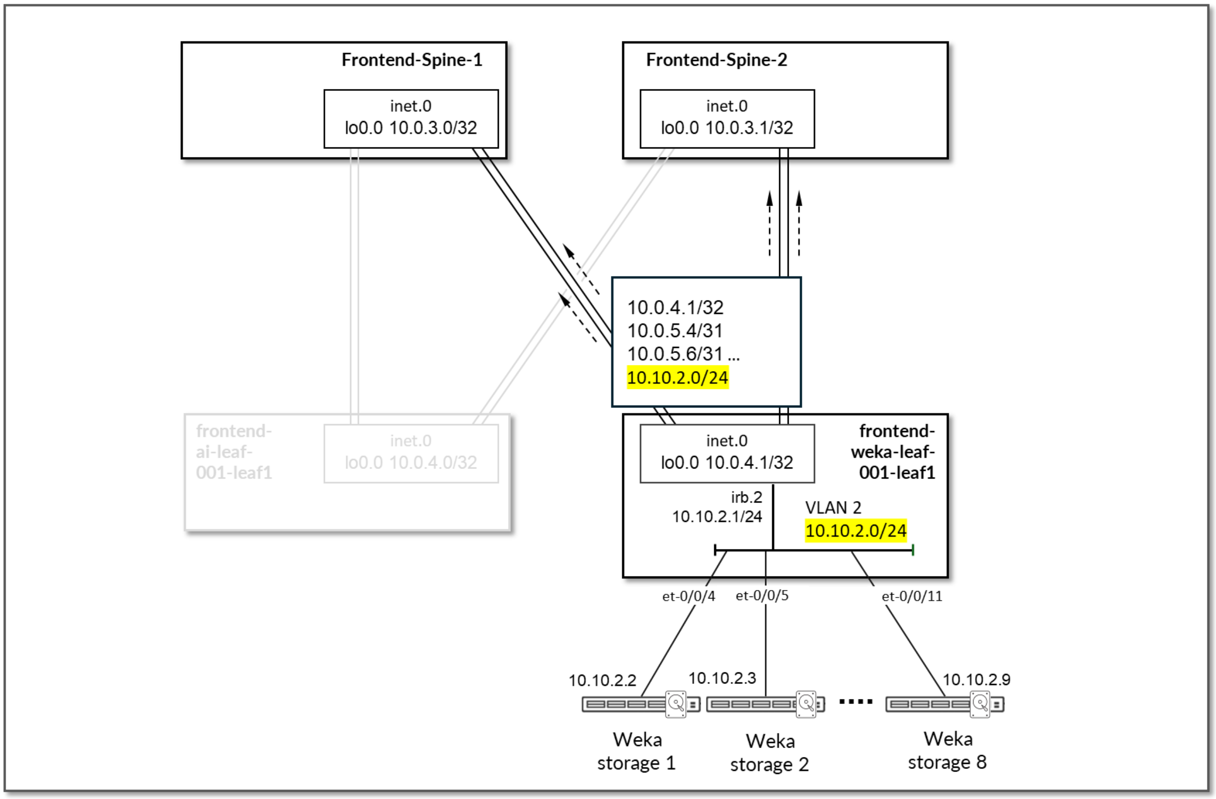


Table 32: Frontend Leaf to Weka Storage Advertised Routes

Leaf Node	Peer(s)	Advertised Routes		BGP Communities
frontend-weka-leaf 1	*frontend-spine1* & *frontend-spine2*	Loopback: 10.0.4.1/32 Leaf-spines links: 10.0.5.4/31 10.0.5.6/31 10.0.5.12/31 10.0.5.14/31	GPU servers <=> frontend spine links: 10.10.2.0/24	4:20007 21001:26000

On the Spine nodes, BGP policies are configured by Apstra to advertise the following routes to the frontend-ai-leaf node:

- *frontend-spine* node own loopback interface address
- *frontend-weka-leaf* 1 loopback interface address

- *frontend-spine* to *frontend-weka-leaf* 1 nodes interfaces subnets
- WEKA storage server's subnet (learned from *frontend-weka-leaf* 1)

Figure 79: Frontend Spine to Frontend Leaf for GPU/Headed Servers BGP

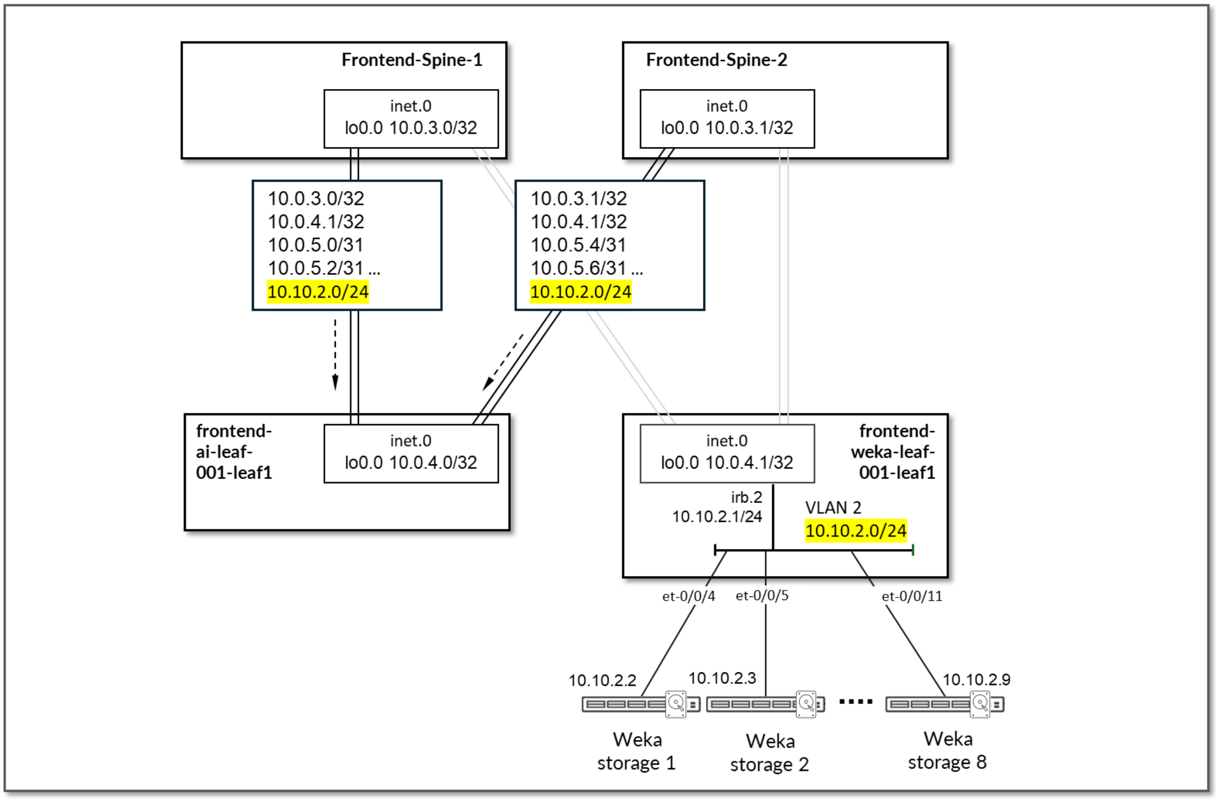


Table 33: Frontend Spine to Frontend Leaf for GPU/Headed Servers Advertised Routes

Leaf Node	Peer(s)	Advertised Routes		BGP Communities
frontend-spine1	*frontend-ai-leaf*	Loopback: 10.0.3.0/32 10.0.4.0/32 Leaf-spines links: 10.0.5.0/31 10.0.5.2/31 10.0.5.4/31 10.0.5.6/31 10.0.5.12/31 10.0.5.14/31	WEKA Servers subnet: 10.10.2.0/24	0:15 1:20007 21001:26000 Except for 10.0.4.0/32 (0:15 3:20007 21001:26000)
frontend-spine2	*frontend-ai-leaf*	Loopbacks: 10.0.3.1/32 10.0.4.0/32 Leaf-spines links: 10.0.5.4/31 10.0.5.6/31 10.0.5.8/31 10.0.5.10/31 10.0.5.12/31 10.0.5.14/31	WEKA Servers subnet: 10.10.2.0/24	0:15 2:20007 21001:26000 Except for 10.0.4.0/32 (0:15 3:20007 21001:26000)

On the Spine nodes, BGP policies are configured by Apstra to advertise the following routes to the *frontend-weka-leaf* 1 leaf node:

- spine node own loopback interface address
- frontend-ai-leaf1 loopback interface address
- spine to frontend-ai-leaf1 nodes interfaces subnets

- GPU servers to frontend-ai-leaf1 node link subnets

Figure 80: Frontend Spine to Frontend Leaf for WEKA Storage Headend Server BGP

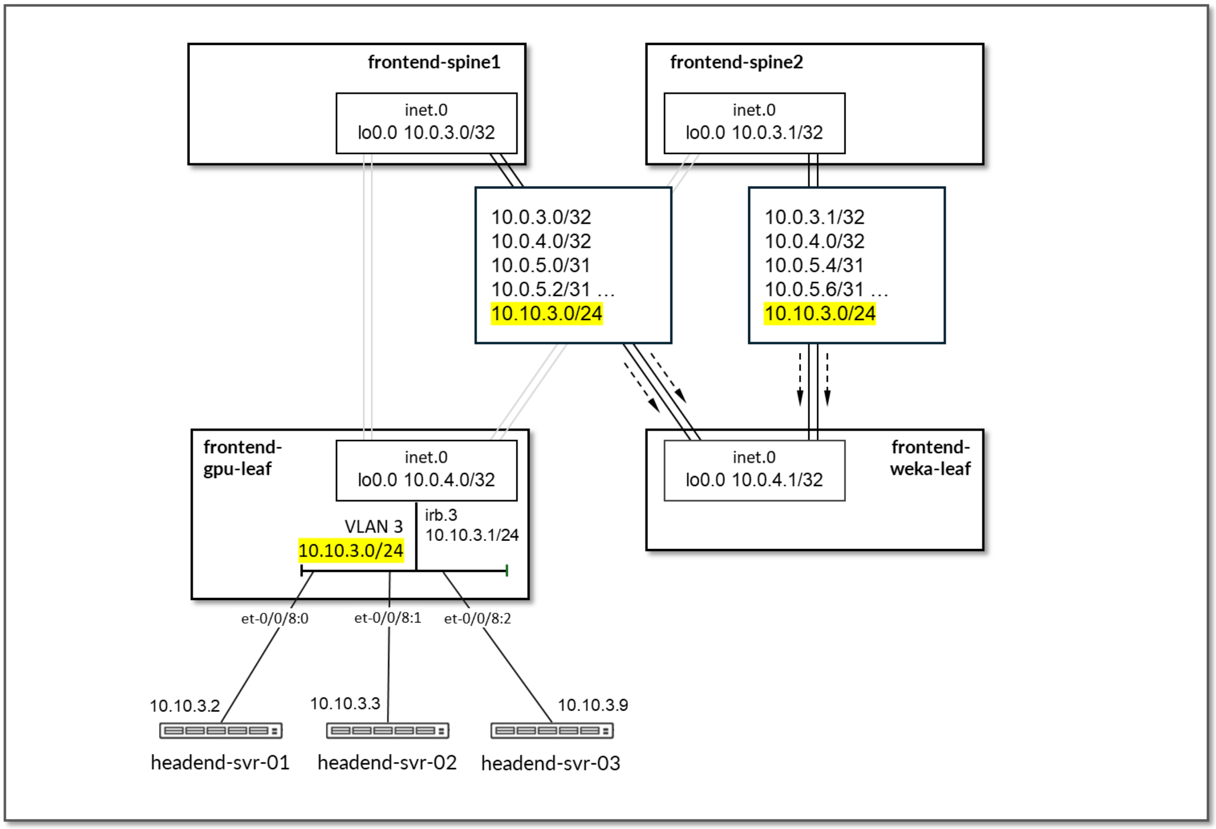


Figure 81: Frontend Spine to Frontend Leaf for WEKA Storage GPU Server BGP

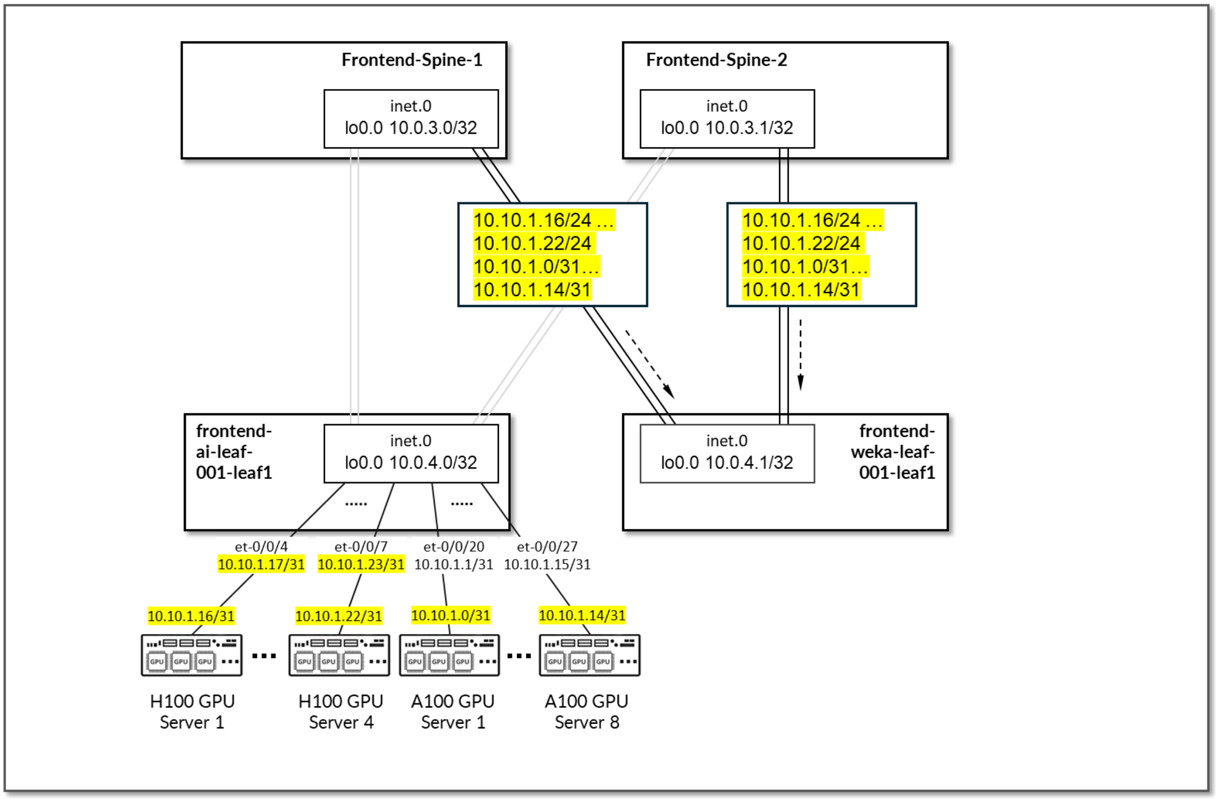


Table 34 Frontend Spine to Frontend Leaf for WEKA Storage Advertised Routes

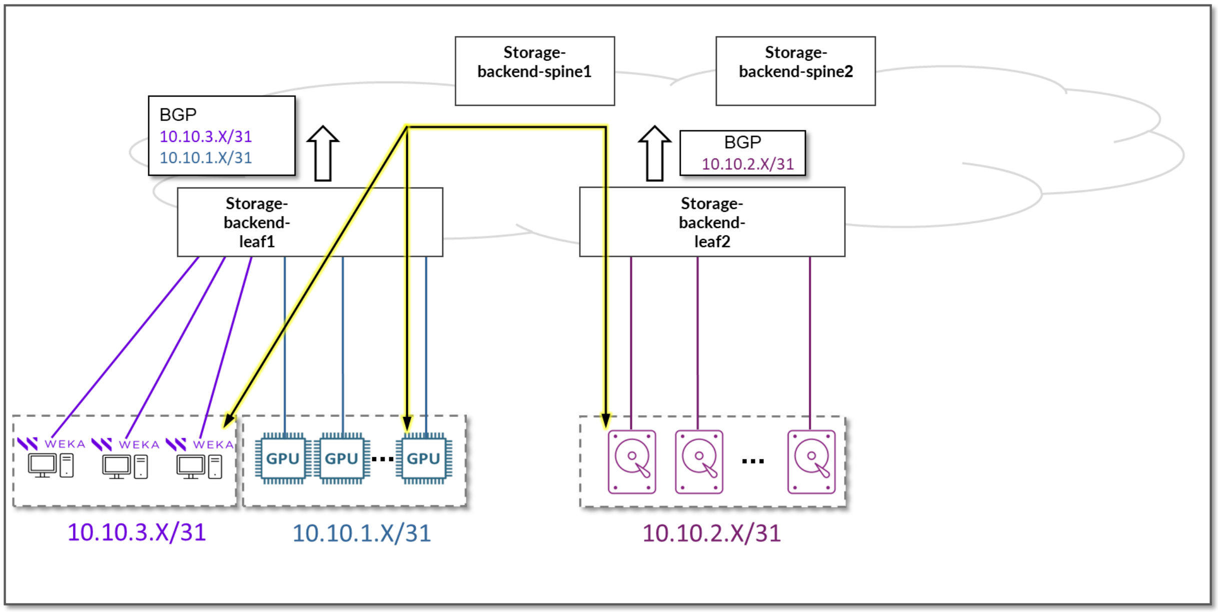
Leaf Node	Peer(s)	Advertised Routes		BGP Communities
frontend-spine1	*frontend-ai-leaf*	Loopback: 10.0.3.0/32 10.0.4.1/32 Leaf-spines links: 10.0.5.0/31 10.0.5.2/31 10.0.5.4/31 10.0.5.6/31 10.0.5.8/31 10.0.5.10/31	GPU server <=> frontend spine links: 10.10.1.16/31 10.10.1.18/31 10.10.1.20/31 10.10.1.22/31 10.10.1.0/31 10.10.1.2/31 10.10.1.4/31 10.10.1.6/31 10.10.1.8/31 10.10.1.10/31 10.10.1.12/31 10.10.1.14/31 WEKA Server's Management subnet: 10.10.3.0/24	0:15 1:20007 21001:26000 Except for 10.0.4.1/32 (0:15 4:20007 21001:26000)

(Continued)

Leaf Node	Peer(s)	Advertised Routes		BGP Communities
frontend-spine2	*frontend-ai-leaf*	Loopbacks: 10.0.3.1/32 10.0.4.1/32 Leaf-spines links: 10.0.5.0/31 10.0.5.2/31 10.0.5.8/31 10.0.5.10/31 10.0.5.12/31 10.0.5.14/31	GPU servers <=> frontend spine links: 10.10.1.16/31 10.10.1.18/31 10.10.1.20/31 10.10.1.22/31 10.10.1.0/31 10.10.1.2/31 10.10.1.4/31 10.10.1.6/31 10.10.1.8/31 10.10.1.10/31 10.10.1.12/31 10.10.1.14/31 WEKA Management server's subnet: 10.10.3.0/24	0:15 2:20007 21001:26000 Except for 10.0.4.1/32 (0:15 4:20007 21001:26000)

By advertising the subnet assigned to the links between the leaf nodes and the GPU/storage servers, communication between GPUs and the WEKA storage and WEKA management servers is possible across the fabric.

Figure 82: GPU Server to WEKA storage and WEKA Management Servers



All the devices are configured to perform ECMP load balancing, as explained later in the document.

GPU Backend Network Connectivity

The GPU Backend fabric is designed as a Layer 3 IP Fabric, where the links between the leaf and spine nodes are configured with /31 IP addresses and are running EBGP. The fabric consists of 2 spine nodes, and 8 spine nodes (per stripe).

There is a single 400GE link between each leaf node and the spine nodes.

Figure 83: GPU Backend Spine to GPU Backend Leaf Nodes Connectivity

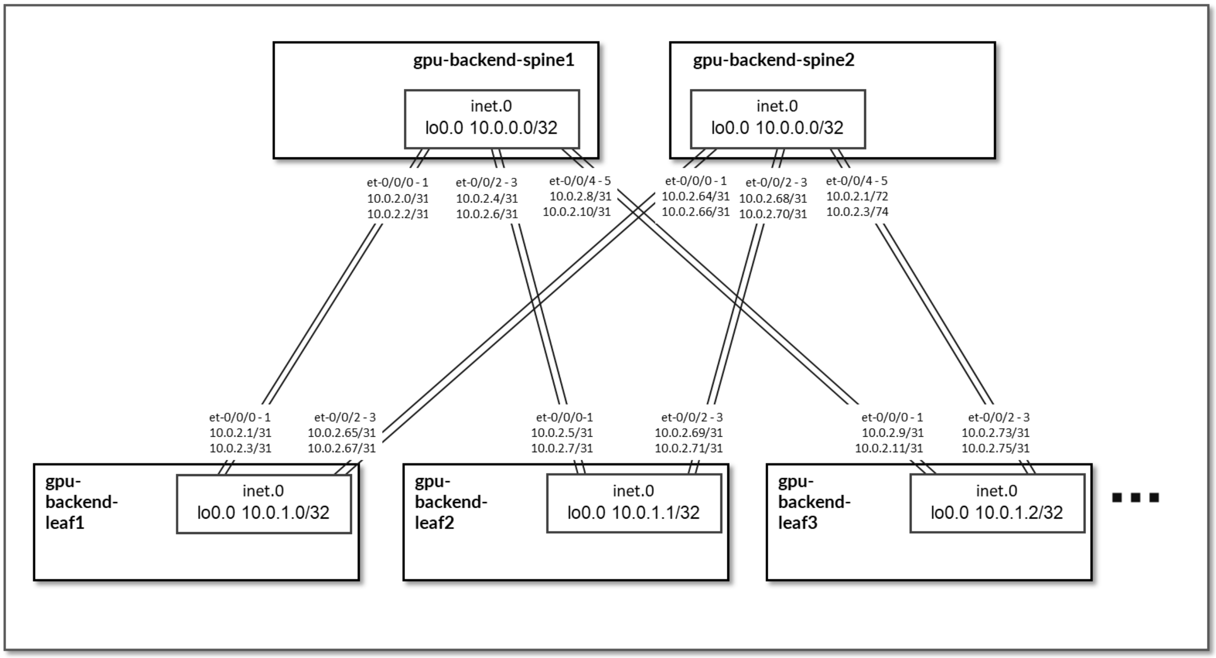


Table 35: GPU Backend Interface Addresses

Stripe #	Spine node	Leaf node	Spine IP address	Leaf IP address
1	*gpu-backend-spine 1*	*gpu-backend-leaf1*	10.0.2.0/31 10.0.2.2/31	10.0.2.1/31 10.0.2.3/31
1	*gpu-backend-spine 1*	*gpu-backend-leaf2*	10.0.2.4/31 10.0.2.6/31	10.0.2.5/31 10.0.2.7/31
1	*gpu-backend-spine 1*	*gpu-backend-leaf3*	10.0.2.8/31 10.0.2.10/31	10.0.2.9/31 10.0.2.11/31
. . .				

(Continued)

Stripe #	Spine node	Leaf node	Spine IP address	Leaf IP address
1	*gpu-backend-spine* 2	*gpu-backend-leaf1*	10.0.2.64/31 10.0.2.66/31	10.0.2.65/31 10.0.2.67/31
1	*gpu-backend-spine* 2	*gpu-backend-leaf2*	10.0.2.68/31 10.0.2.70/31	10.0.2.69/31 10.0.2.71/31
1	*gpu-backend-spine* 2	*gpu-backend-leaf3*	10.0.2.72/31 10.0.2.74/31	10.0.2.73/31 10.0.2.75/31

The loopback interfaces also have addresses automatically assigned by Apstra from a predefined pool.

NOTE: all IP addresses are assigned by Apstra (from predefined pools of resources) based on the intent.

Table 36: GPU Backend Loopback Addresses

Stripe #	Device	Loopback Interface Address
1	*gpu-backend-spine1*	10.0.0.0/32
1	*gpu-backend-spine2*	10.0.0.1/32
1	*gpu-backend-leaf1*	10.0.1.0/32
1	*gpu-backend-leaf2*	10.0.1.1/32
1	*gpu-backend-leaf3*	10.0.1.2/32

Each leaf node is assigned a /24 subnet out of 10.200/16 and a unique VLAN ID to provide connectivity to the GPU servers. Layer 3 connectivity is provided via an irb interface with an address out of the specific IP subnet, as shown in the table below.

Because each leaf node represents a rail, where all the GPUs with a given number connect, each rail in the cluster is mapped to a different /24 IP subnet.

Figure 84: GPU Backend Servers to Leaf Nodes Connectivity

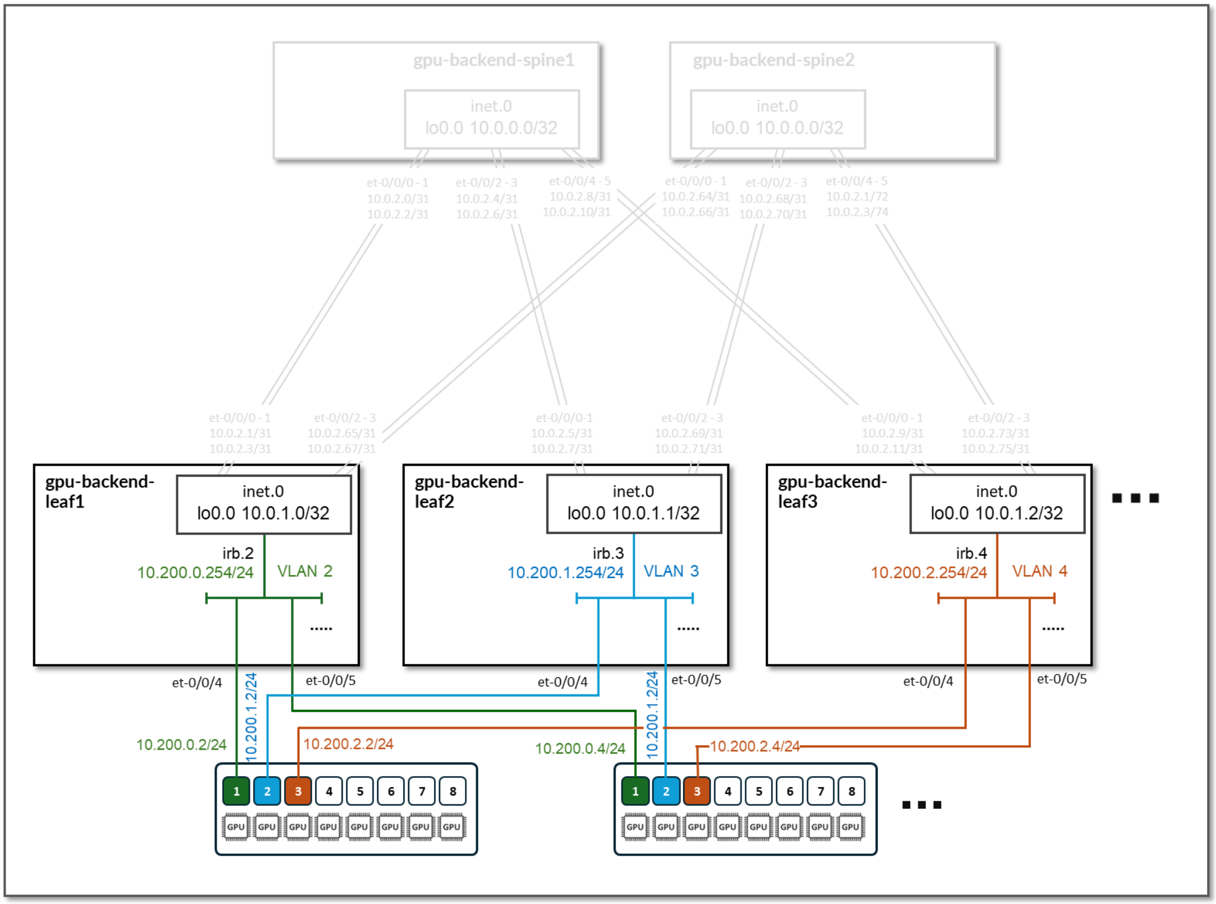


Table 37: GPU Backend Servers to Leaf Nodes Connectivity

Stripe #	Device	Rail #	VLAN #	Subnet	IRB on leaf	Connected device(s)
1	*gpu-backend-leaf* 1	1	2	10.200.0.0/24	10.200.0.254	GPU 1 from all 8 GPU servers
1	*gpu-backend-leaf* 2	2	3	10.200.1.0/24	10.200.1.254	GPU 2 from all 8 GPU servers
1	*gpu-backend-leaf* 3	3	4	10.200.2.0/24	10.200.2.254	GPU 3 from all 8 GPU servers

(Continued)

Stripe #	Device	Rail #	VLAN #	Subnet	IRB on leaf	Connected device(s)
. . .						

EBGP is configured between the IP addresses assigned to the spine-leaf nodes links, as shown in Figure 81. There will be 2 EBGP sessions between each ***gpu-backend-leaf #*** node and each ***gpu-backend-spine #***.

Figure 85: GPU Backend BGP Sessions

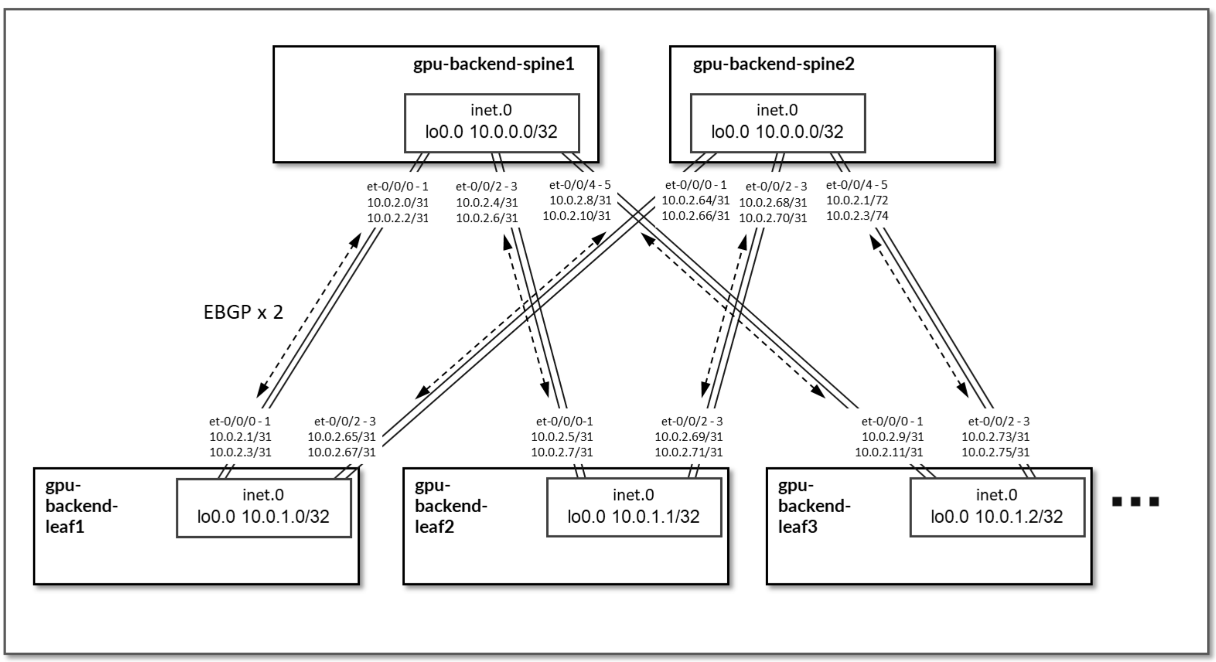


Table 38: GPU Backend Sessions

Stripe #	Spine Node	Leaf Node	Spine ASN	Leaf ASN	Spine IP Address	Leaf IP Address
1	*gpu-backend-spine1*	*gpu-backend-leaf1*	4201032100	4201032200	10.0.2.0/31 10.0.2.2/31	10.0.2.1/31 10.0.2.3/31

(Continued)

Stripe #	Spine Node	Leaf Node	Spine ASN	Leaf ASN	Spine IP Address	Leaf IP Address
1	*gpu-backend-spine1*	*gpu-backend-leaf2*		4201032201	10.0.2.4/31 10.0.2.6/31	10.0.2.5/31 10.0.2.7/31
1	*gpu-backend-spine1*	*gpu-backend-leaf3*		4201032202	10.0.2.8/31 10.0.2.10/31	10.0.2.9/31 10.0.2.11/31
	...					
1	*gpu-backend-spine2*	*gpu-backend-leaf1*	4201032101	4201032200	10.0.2.64/31 10.0.2.66/31	10.0.2.65/31 10.0.2.67/31
1	*gpu-backend-spine2*	*gpu-backend-leaf2*		4201032201	10.0.2.68/31 10.0.2.70/31	10.0.2.69/31 10.0.2.71/31
1	*gpu-backend-spine2*	*gpu-backend-leaf3*		4201032202	10.0.2.72/31 10.0.2.74/31	10.0.2.73/31 10.0.2.75/31
	...					

All the Autonomous System and community values are assigned by Apstra (from predefined pools of resources) based on the intent.

On the Leaf nodes, BGP policies are configured by Apstra to advertise the following routes to the spine nodes:

- Leaf node own loopback interface address
- leaf to spine interfaces subnets and
- irb interface subnet

Figure 86: GPU Backend Leaf Node BGP

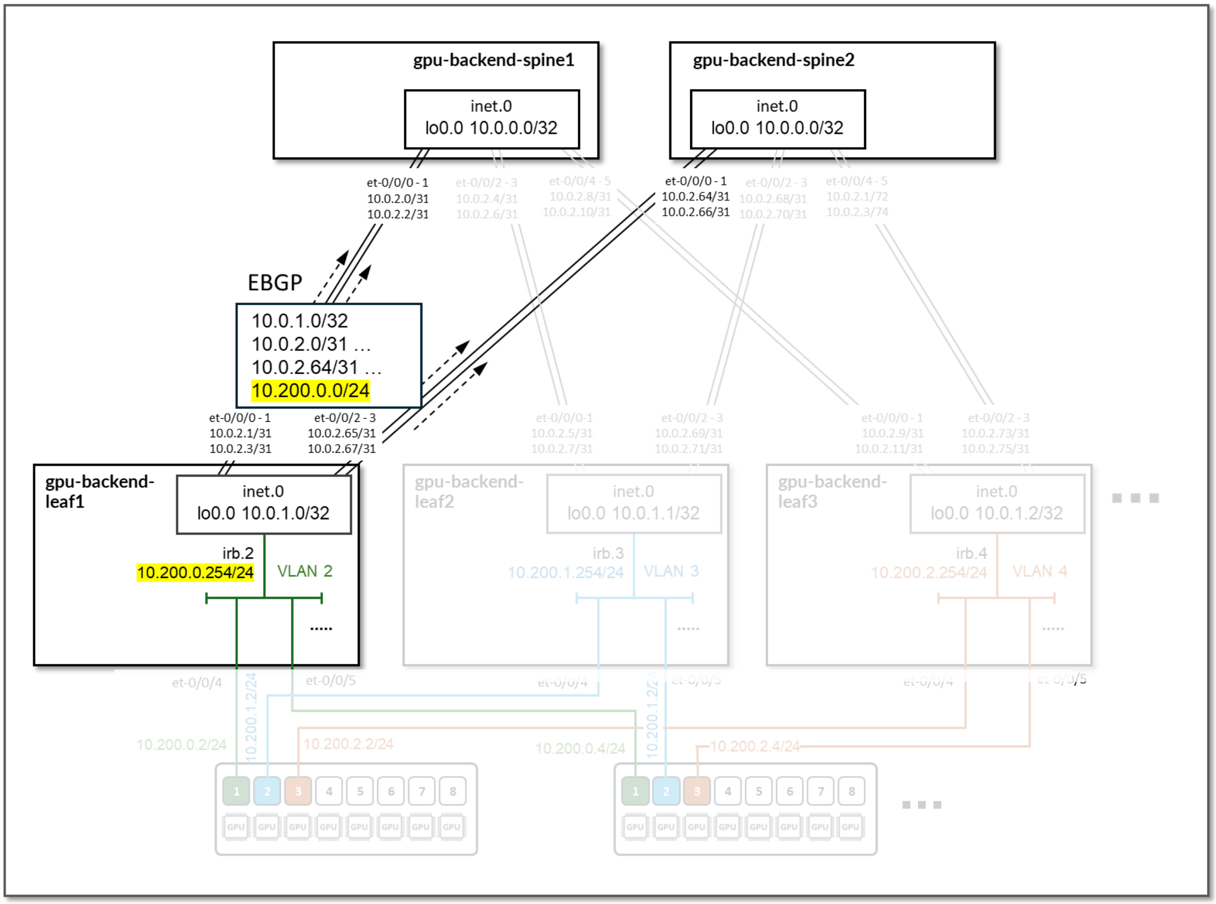


Table 39: GPU Backend Leaf Node Advertised Routes

Stripe #	Device	Advertised routes	BGP community
1	*gpu-backend-leaf* 1	10.0.1.0/32 10.0.2.0/31 10.0.2.64/31 10.200.0.0/24	3:20007 21001:26000

(Continued)

Stripe #	Device	Advertised routes	BGP community
1	*gpu-backend-leaf* 2	10.0.1.1/32 10.0.2.4/31 10.0.2.68/31 10.200.1.0/24	4:20007 21001:26000
1	*gpu-backend-leaf* 3	10.0.1.2/32 10.0.2.8/31 10.0.2.72/31 10.200.2.0/24	5:20007 21001:26000

On the Spine nodes, BGP policies are configured by Apstra to advertise the following routes to the leaf nodes:

- spine node own loopback interface address
- leaf nodes' loopback interface address
- spine to leaf interfaces subnets
- irb interface subnet, as shown below:

Figure 87: GPU Backend Spine Node BGP

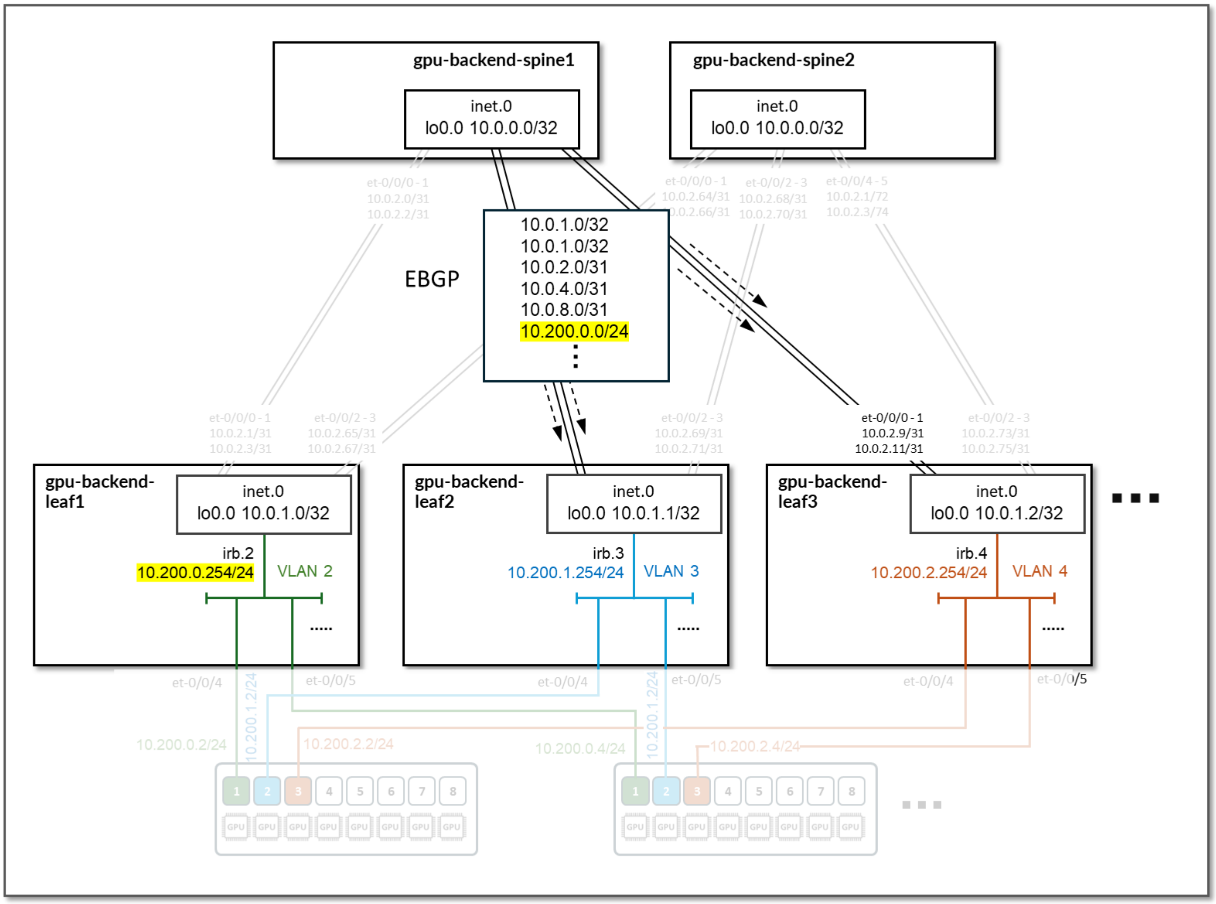
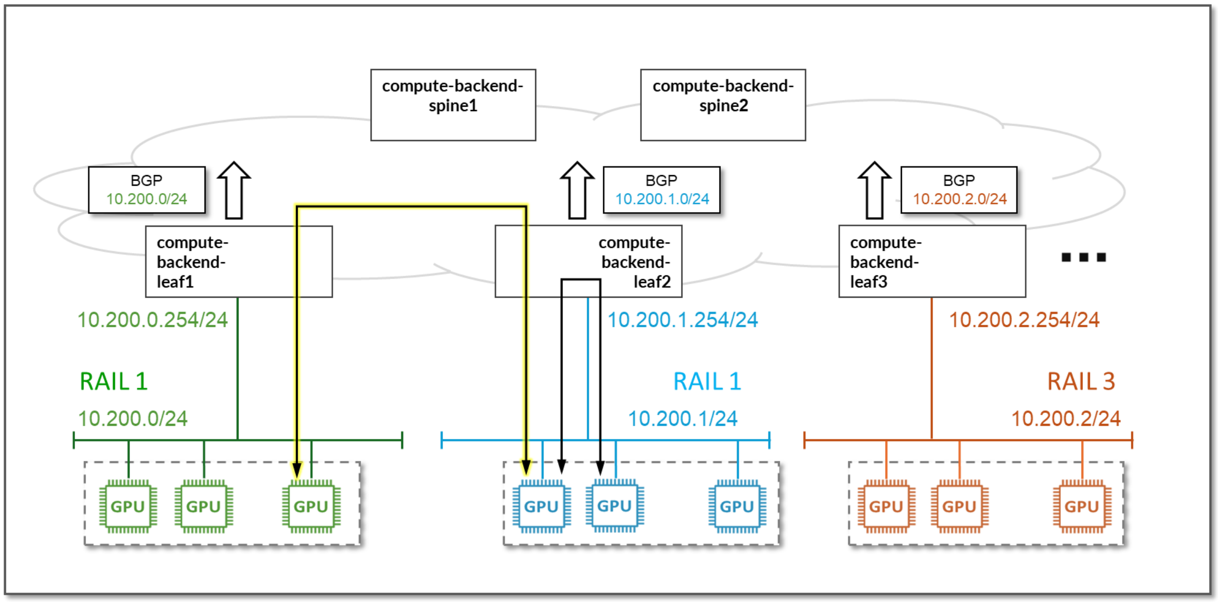


Table 40: GPU Backend Spine Node Advertised Routes

Stripe #	Spine Node	Advertised Routes	BGP Community
1	*gpu-backend-spine* 1	10.0.0.0/32 10.0.2.0/31 10.0.2.4/31 ... 10.200.1.0/24 ...	0:15 X:20007 21001:26000
1	*gpu-backend-spine* 2	10.0.0.1/32 10.0.2.64/31 10.0.2.68/31 ... 10.200.1.0/24 ...	0:15 X:20007 21001:26000

By advertising the irb interfaces subnet, communication between GPUs in different rails is possible across the fabric.

Figure 88: Communication Across Rails



All the devices are configured to perform ECMP load balancing, as explained later in the document.

Storage Backend Network Connectivity

The Storage Backend fabric is designed as a Layer 3 IP Fabric, where the links between the leaf and spine nodes are configured with /31 IP addresses as shown in the table below. The fabric consists of 2 spine nodes and 4 leaf nodes, where 2 leaf nodes are used to connect the storage servers (named ***storage-backend-weka-leaf #)*** and 2 are used to connect to the GPU servers (named ***storage-backend-gpu-leaf #).***

There are three 400GE links between each ***storage-backend-weka-leaf #*** node and the spine nodes and two 400GE links between each ***storage-backend-gpu-leaf #*** node and the spine nodes as shown in Figure 89.

Figure 89: Storage Backend Spine to Storage Backend GPU Leaf Nodes Connectivity

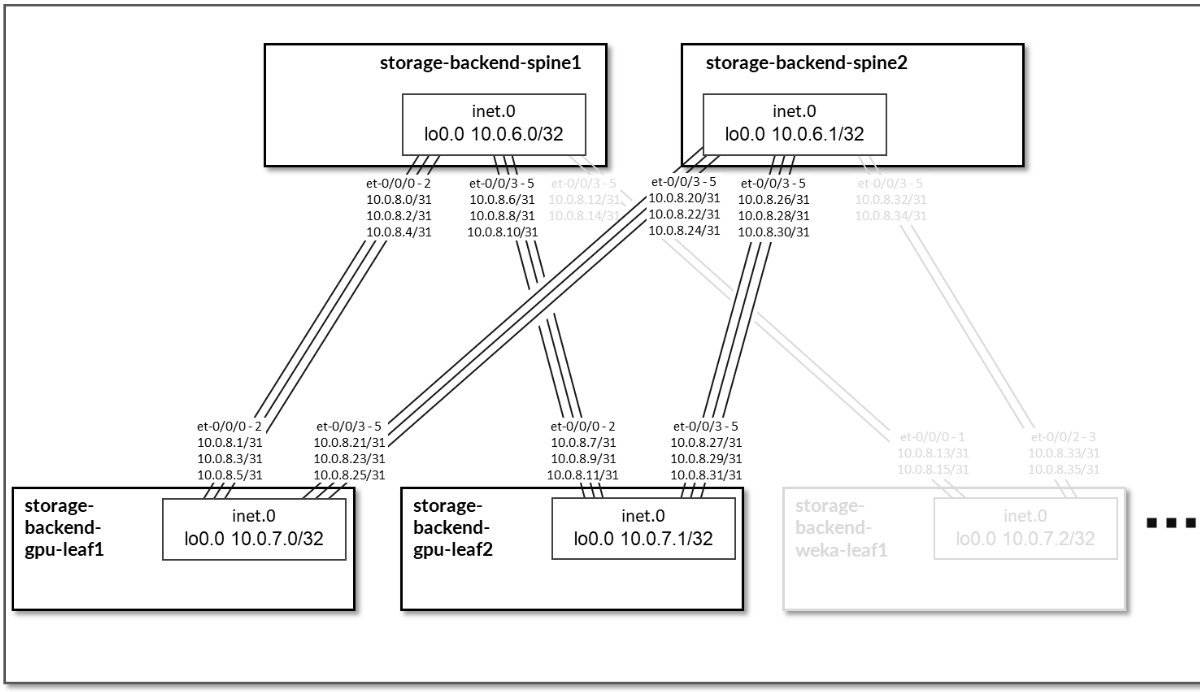


Figure 90: Storage Backend Spine to Storage Backend WEKA Storage Leaf Nodes Connectivity

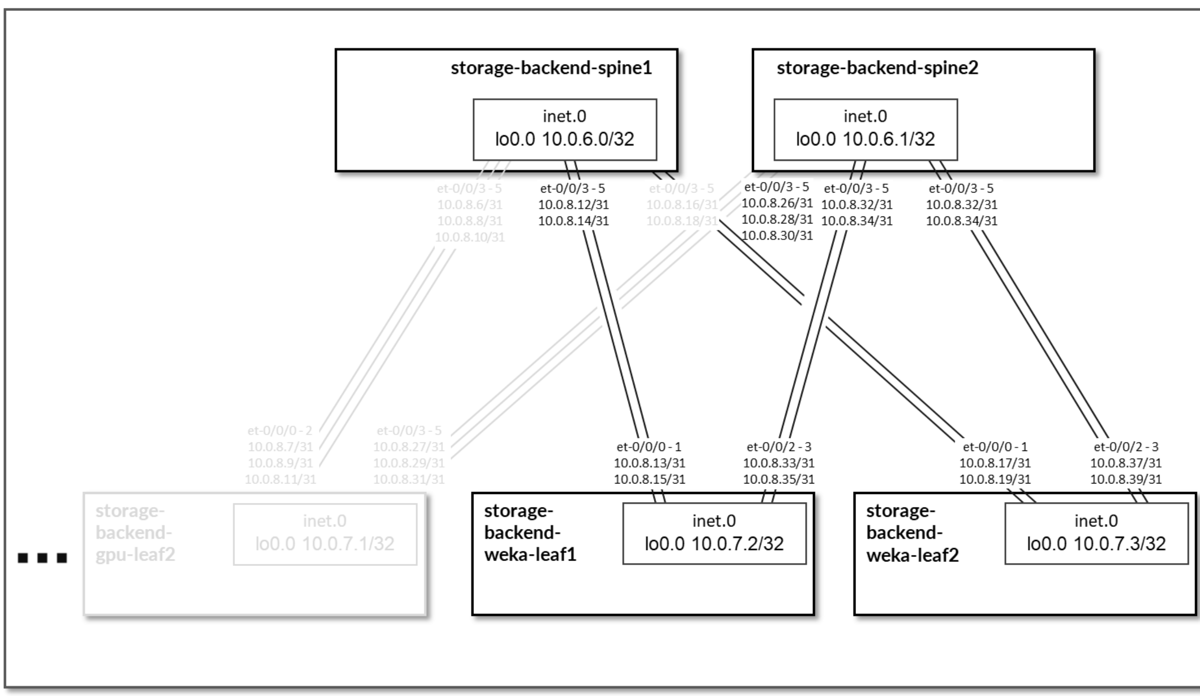


Table 41: Storage Backend Interface Addresses

Spine node	Leaf node	Spine IP Address	Leaf IP Address
storage-backend-spine 1	*storage-backend-gpu-leaf* 1	10.0.8.0/31 10.0.8.2/31 10.0.8.4/31	10.0.8.1/31 10.0.8.3/31 10.0.8.5/31
storage-backend-spine1	*storage-backend-gpu-leaf2*	10.0.8.6/31 10.0.8.8/31 10.0.8.10/31	10.0.8.7/31 10.0.8.9/31 10.0.8.11/31
storage-backend-spine1	*storage-backend-weka-leaf1*	10.0.8.12/31 10.0.8.14/31	10.0.8.13/31 10.0.8.15/31
storage-backend-spine1	*storage-backend-weka-leaf2*	10.0.8.16/31 10.0.8.18/31	10.0.8.17/31 10.0.8.19/31
storage-backend-spine2	*storage-backend-gpu-leaf1*	10.0.8.20/31 10.0.8.22/31 10.0.8.24/31	10.0.8.21/31 10.0.8.23/31 10.0.8.25/31
storage-backend-spine2	*storage-backend-gpu-leaf2*	10.0.8.26/31 10.0.8.28/31 10.0.8.30/31	10.0.8.27/31 10.0.8.29/31 10.0.8.31/31
storage-backend-spine2	*storage-backend-weka-leaf1*	10.0.8.32/31 10.0.8.34/31	10.0.8.33/31 10.0.8.35/31
storage-backend-spine2	*storage-backend-weka-leaf2*	10.0.8.36/31 10.0.8.38/31	10.0.8.37/31 10.0.8.39/31

NOTE: all IP addresses are assigned by Apstra (from predefined pools of resources) based on the intent.

The loopback interfaces also have addresses automatically assigned by Apstra from a predefined pool.

Table 42: Storage Backend Loopback Interfaces

Device	Loopback Interface Address
storage-backend-spine1	10.0.6.0/32
storage-backend-spine2	10.0.6.1/32
storage-backend-gpu-leaf1	10.0.7.0/32
storage-backend-gpu-leaf2	10.0.7.1/32
storage-backend-weka-leaf1	10.0.7.2/32
storage-backend-weka-leaf2	10.0.7.3/32

The H100 GPU Servers and A100 GPU Servers are connected to the storage backend leaf switches as summarized in the following table.

Table 43: Storage GPU Backend Servers to Leaf Nodes Connectivity

GPU servers	Leaf Node
H100-1	*storage-backend-gpu-leaf1*
H100-2	
A100-1	
A100-2	
A100-3	
A100-4	
H100-3	*storage-backend-gpu-leaf2*
H100-4	
A100-5	

(Continued)

GPU servers	Leaf Node
A100-6	
A100-7	
A100-8	

The links between the GPU servers and ***storage-backend-gpu-leaf*** 1 are assigned /31 subnets out of 10.100.1/24, while the links between the GPU servers and ***storage-backend-gpu-leaf*** 2 are assigned /31 subnets out of 10.100.2/24, as shown in Figure 91.

Figure 91: GPU Servers to Storage Backend GPU Leaf nodes Connectivity

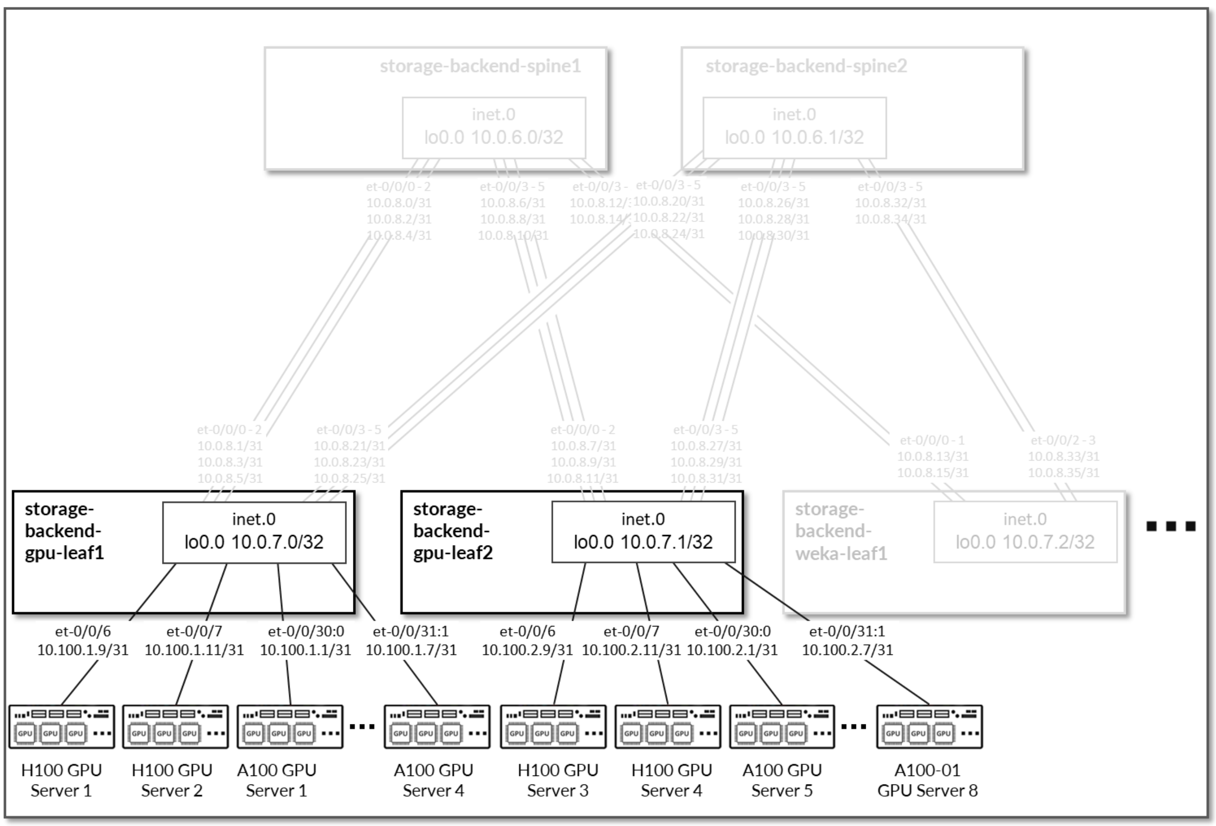
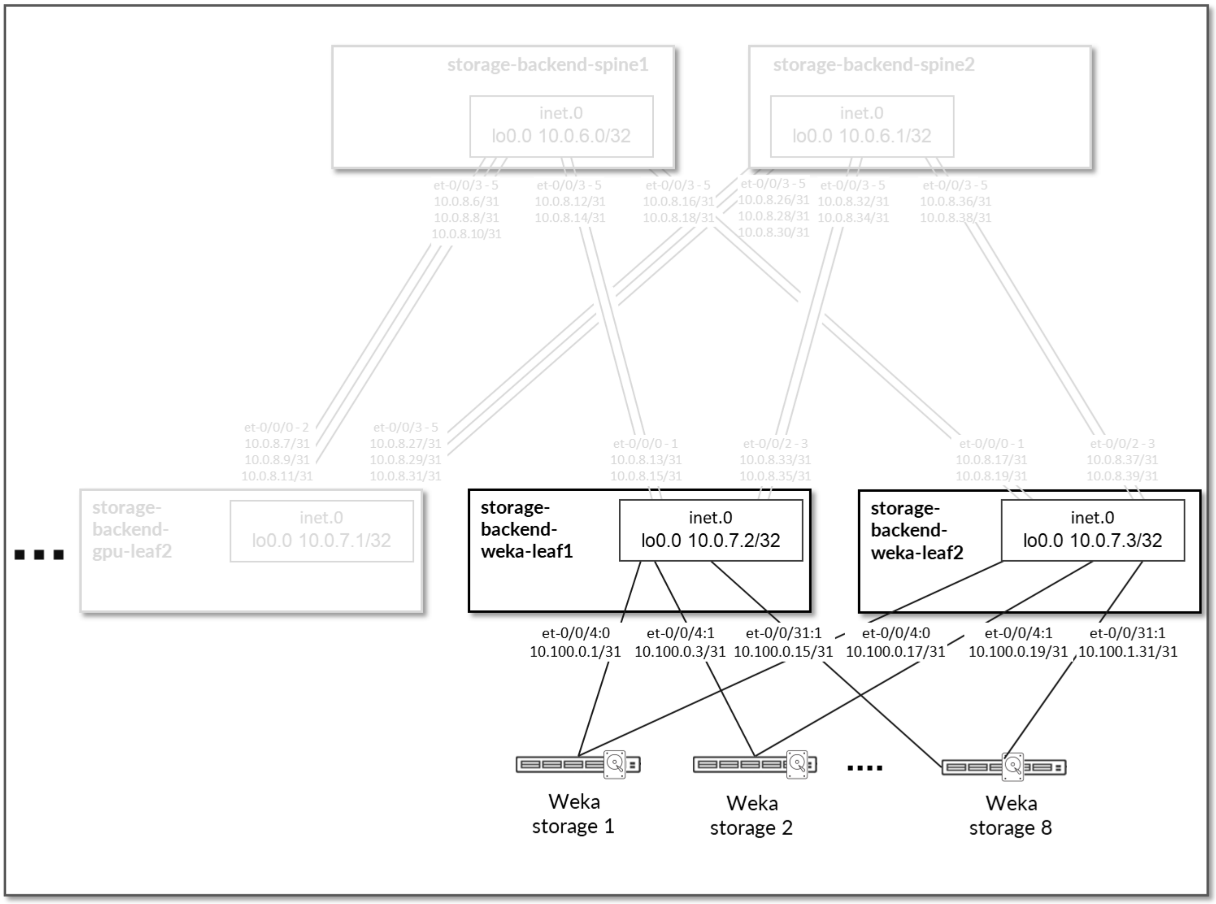


Table 44: GPU Servers to Storage GPU Backend Interface Addresses

GPU Server	Leaf Node	GPU Server IP Address	Leaf IP Address
H100 GPU Server 1	*storage-backend-gpu-leaf* 1	10.100.1.8/31	10.100.1.9/31
H100 GPU Server 2	*storage-backend-gpu-leaf* 1	10.100.1.10/31	10.100.1.11/31
A100 GPU Server 1	*storage-backend-gpu-leaf* 1	10.100.1.0/31	10.100.1.1/31
A100 GPU Server 2	*storage-backend-gpu-leaf* 1	10.100.1.2/31	10.100.1.3/31
A100 GPU Server 3	*storage-backend-gpu-leaf* 1	10.100.1.4/31	10.100.1.5/31
A100 GPU Server 4	*storage-backend-gpu-leaf* 1	10.100.1.6/31	10.100.1.7/31
H100 GPU Server 3	*storage-backend-gpu-leaf* 2	10.100.2.8/31	10.100.2.9/31
H100 GPU Server 4	*storage-backend-gpu-leaf* 2	10.100.2.10/31	10.100.2.11/31
A100 GPU Server 5	*storage-backend-gpu-leaf* 2	10.100.2.0/31	10.100.2.1/31
A100 GPU Server 6	*storage-backend-gpu-leaf* 2	10.100.2.2/31	10.100.2.3/31
A100 GPU Server 7	*storage-backend-gpu-leaf* 2	10.100.2.4/31	10.100.2.5/31
A100 GPU Server 8	*storage-backend-gpu-leaf* 2	10.100.2.6/31	10.100.2.7/31

Like the GPU servers, the WEKA storage servers are connected to the two *storage-backend-weka-leaf #* nodes as shown Figure 92.

Figure 92: WEKA Storage servers to Leaf Nodes Connectivity



Each GPU server to leaf node connection is assigned a /31 subnet out of 10.100.0.0/24, as shown in the following table.

Table 45: WEKA Storage Servers to Leaf Nodes Interface Addresses

WEKA Server	Leaf Node	WEKA Server IP Address	Leaf IP Address
WEKA storage Server 1	*storage-backend-weka-leaf* 1	10.100.0.0/31	10.100.0.1/31
WEKA storage Server 2	*storage-backend-weka-leaf* 1	10.100.0.2/31	10.100.0.3/31
WEKA storage Server 3	*storage-backend-weka-leaf* 1	10.100.0.4/31	10.100.0.5/31
WEKA storage Server 4	*storage-backend-weka-leaf* 1	10.100.0.5/31	10.100.0.7/31

(Continued)

WEKA Server	Leaf Node	WEKA Server IP Address	Leaf IP Address
WEKA storage Server 5	*storage-backend-weka-leaf* 1	10.100.0.8/31	10.100.0.9/31
WEKA storage Server 6	*storage-backend-weka-leaf* 1	10.100.0.10/31	10.100.0.11/31
WEKA storage Server 7	*storage-backend-weka-leaf* 1	10.100.0.12/31	10.100.0.13/31
WEKA storage Server 8	*storage-backend-weka-leaf* 1	10.100.0.14/31	10.100.0.15/31
WEKA storage Server 1	*storage-backend-weka-leaf* 1	10.100.0.16/31	10.100.0.17/31
WEKA storage Server 2	*storage-backend-weka-leaf* 1	10.100.0.18/31	10.100.0.19/31
WEKA storage Server 3	*storage-backend-weka-leaf* 1	10.100.0.20/31	10.100.0.21/31
WEKA storage Server 4	*storage-backend-weka-leaf* 1	10.100.0.22/31	10.100.0.23/31
WEKA storage Server 5	*storage-backend-weka-leaf* 1	10.100.0.24/31	10.100.0.25/31
WEKA storage Server 6	*storage-backend-weka-leaf* 1	10.100.0.26/31	10.100.0.27/31
WEKA storage Server 7	*storage-backend-weka-leaf* 1	10.100.0.28/31	10.100.0.29/31
WEKA storage Server 8	*storage-backend-weka-leaf* 1	10.100.0.30/31	10.100.0.31/31

Notice that the leaf nodes in this case are using physical interfaces to connect to the storage servers. Thus, no irb interface or vlan id are used for this connectivity.

EBGP is configured between the IP addresses assigned to the links between the spine and the leaf nodes as shown in Figure 93.

There will be 3 EBGP sessions between each ***storage-backend-weka-leaf #*** node and the spine nodes. Similarly, there will be 2 EBGP sessions between each ***storage-backend-gpu-leaf #*** node.

Figure 93: Storage Backend Spine to Storage Backend Leave for GPU Servers EBGP

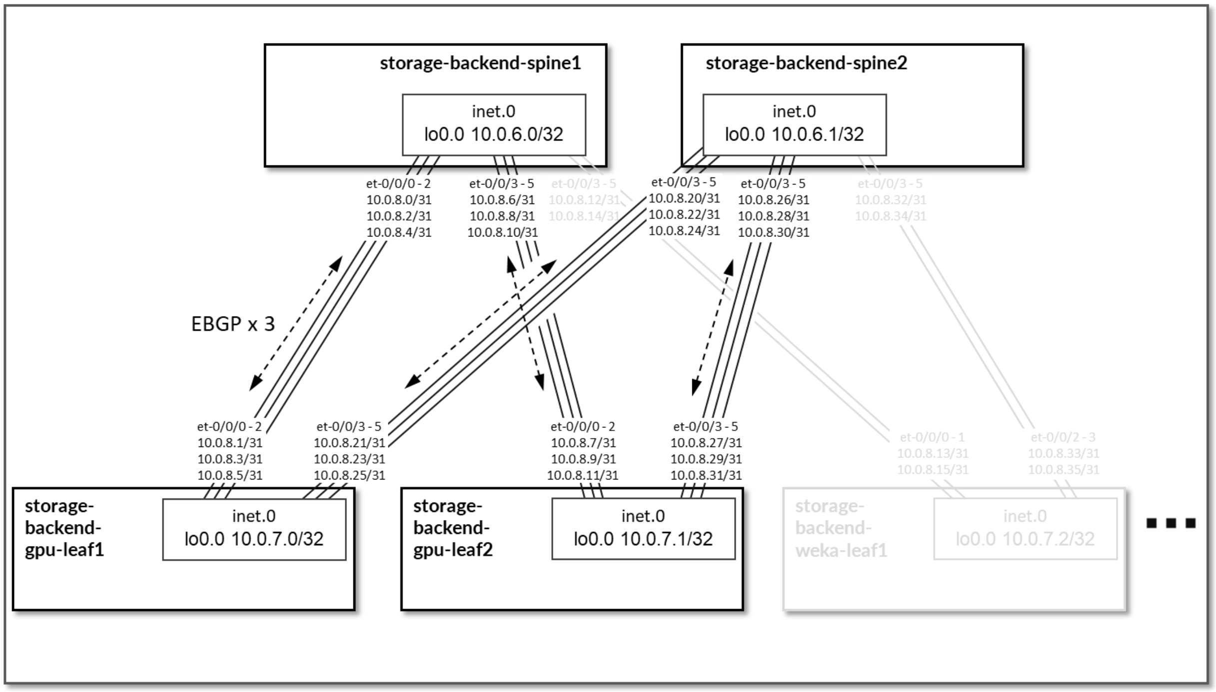


Figure 94: Storage Backend Spine to Storage Backend Leave for WEKA Servers EBGP

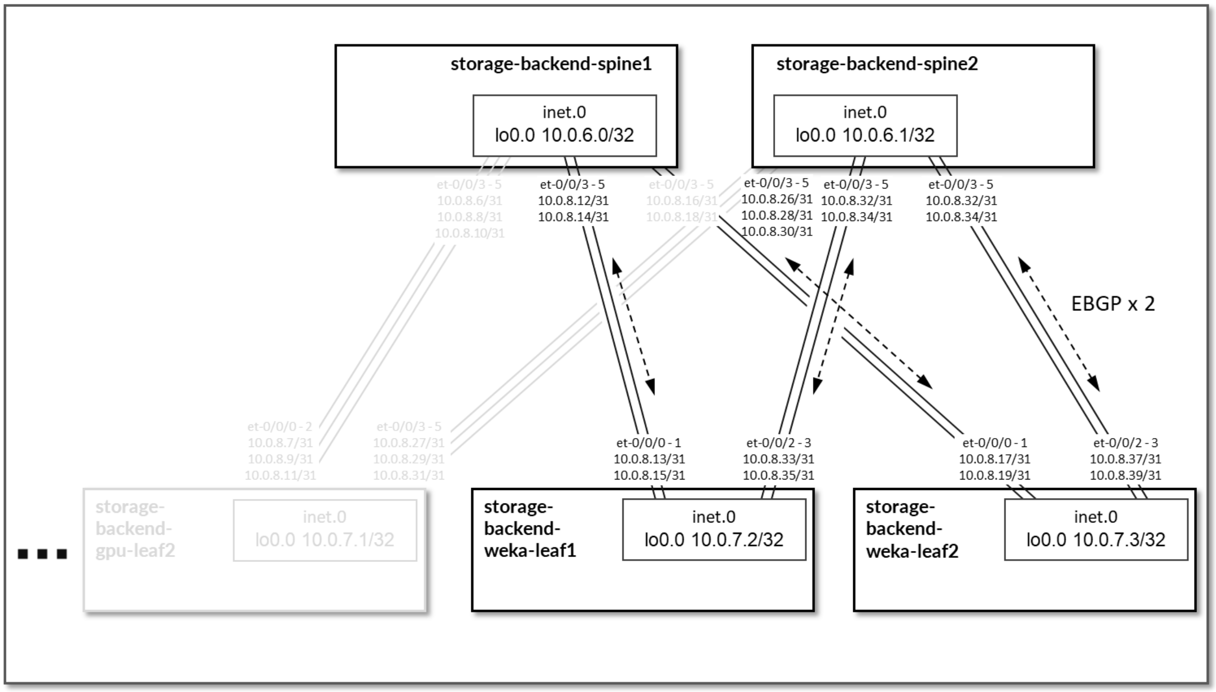


Table 46: Storage Backend Sessions

Spine Node	Leaf Node	Spine ASN	Leaf ASN	Spine IP Address	Leaf IP Address
storage-backend-spine1	*storage-backend-gpu-leaf1*	4201032500	4201032600	10.0.8.0/31 10.0.8.2/31 10.0.8.4/31	10.0.8.1/31 10.0.8.3/31 10.0.8.5/31
storage-backend-spine1	*storage-backend-gpu-leaf2*		4201032601	10.0.8.6/31 10.0.8.8/31 10.0.8.10/31	10.0.8.7/31 10.0.8.9/31 10.0.8.11/31
storage-backend-spine1	*storage-backend-weka-leaf1*		4201032602	10.0.8.12/31 10.0.8.14/31	10.0.8.13/31 10.0.8.15/31

(Continued)

Spine Node	Leaf Node	Spine ASN	Leaf ASN	Spine IP Address	Leaf IP Address
storage-backend-spine1	*storage-backend-weka-leaf2*		4201032603	10.0.8.16/31 10.0.8.18/31	10.0.8.17/31 10.0.8.19/31
storage-backend-spine2	*storage-backend-gpu-leaf1*	4201032501	4201032600	10.0.8.20/31 10.0.8.22/31 10.0.8.24/31	10.0.8.21/31 10.0.8.23/31 10.0.8.25/31
storage-backend-spine2	*storage-backend-gpu-leaf2*		4201032601	10.0.8.26/31 10.0.8.28/31 10.0.8.30/31	10.0.8.27/31 10.0.8.29/31 10.0.8.31/31
storage-backend-spine2	*storage-backend-weka-leaf1*		4201032602	10.0.8.32/31 10.0.8.34/31	10.0.8.33/31 10.0.8.35/31
storage-backend-spine2	*storage-backend-weka-leaf2*		4201032603	10.0.8.36/31 10.0.8.38/31	10.0.8.37/31 10.0.8.39/31

On the Leaf nodes BGP policies are configured by Apstra to advertise the following routes to the spine nodes:

NOTE: all the Autonomous System and community values are assigned by Apstra (from predefined pools of resources) based on the intent.

- Leaf node own loopback interface address,
- leaf to spine interfaces subnets and
- GPU/WEKA storage server to leaf node link subnets.

Figure 95: Storage Backend Leaf BGP

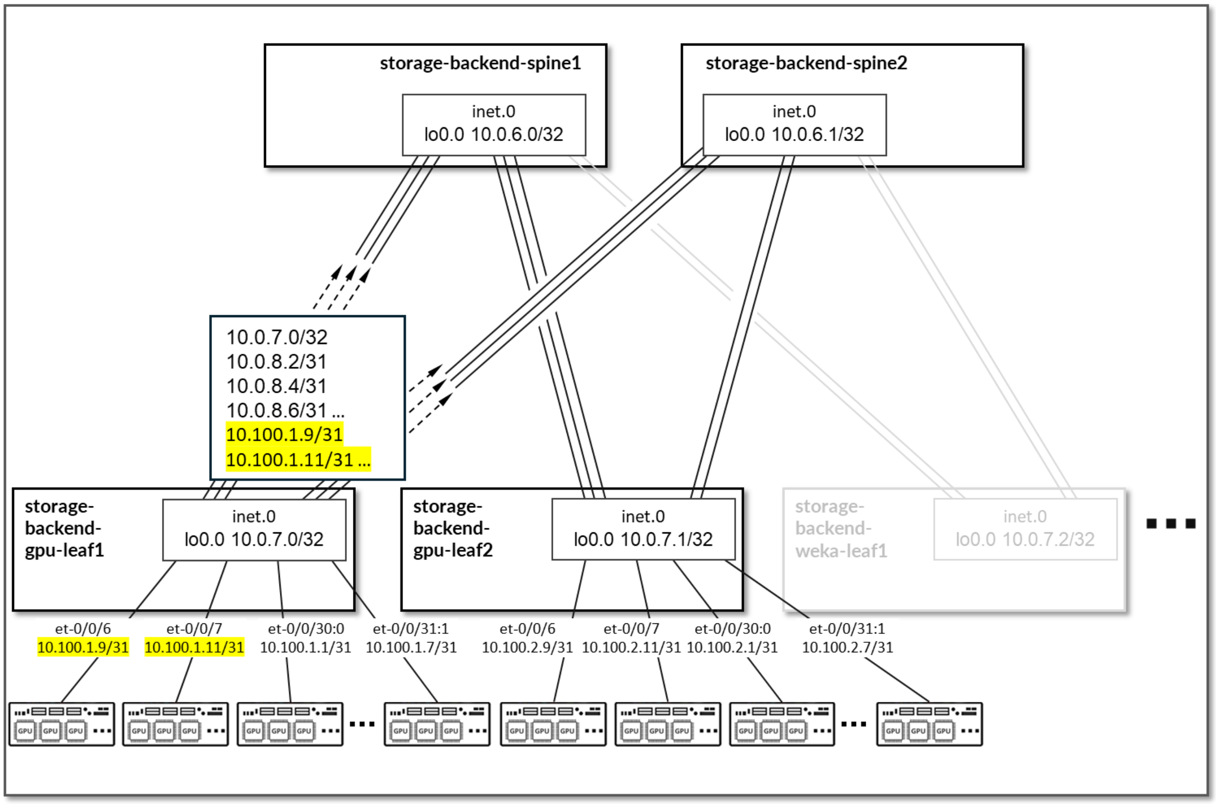


Table 47: Storage Backend Leaf Node Advertised Routes

Leaf Node	Peer	Advertised Routes		BGP Communities
storage-backend-gpu-leaf1	*storage-backend-spine1 &* *storage-backend-spine2*	10.0.7.0/32 10.0.8.0/31 10.0.8.2/31 10.0.8.4/31 10.0.8.20/31 ...	10.100.1.0/31 10.100.1.2/31 ...	3:20007 21001:26000
storage-backend-gpu-leaf2	*storage-backend-spine1 &* *storage-backend-spine2*	10.0.7.1/32 10.0.8.6/31 10.0.8.8/31 10.0.8.10/31 10.0.8.26/31 ...	10.100.2.0/31 10.100.2.2/31 ...	4:20007 21001:26000

(Continued)

Leaf Node	Peer	Advertised Routes		BGP Communities
storage-backend-weka-leaf1	*storage-backend-spine1 &* *storage-backend-spine2*	10.0.7.2/32 10.0.8.12/31 10.0.8.14/31 10.0.8.32/31 ...	10.100.0.16/31 10.100.0.18/31 ...	5:20007 21001:26000
storage-backend-weka-leaf2	*storage-backend-spine1 &* *storage-backend-spine2*	10.0.7.3/32 10.0.8.16/31 10.0.8.17/31 10.0.8.36/31 ...	10.100.0.16/31 10.100.0.18/31 ...	6:20007 21001:26000

On the Spine nodes, BGP policies are configured by Apstra to advertise the following routes to the leaf nodes:

- spine node own loopback interface address
- leaf nodes' loopback interface address
- spine to leaf interfaces subnets
- GPU/WEKA storage server to leaf node link subnets.

Figure 96: Storage Backend Spine BGP

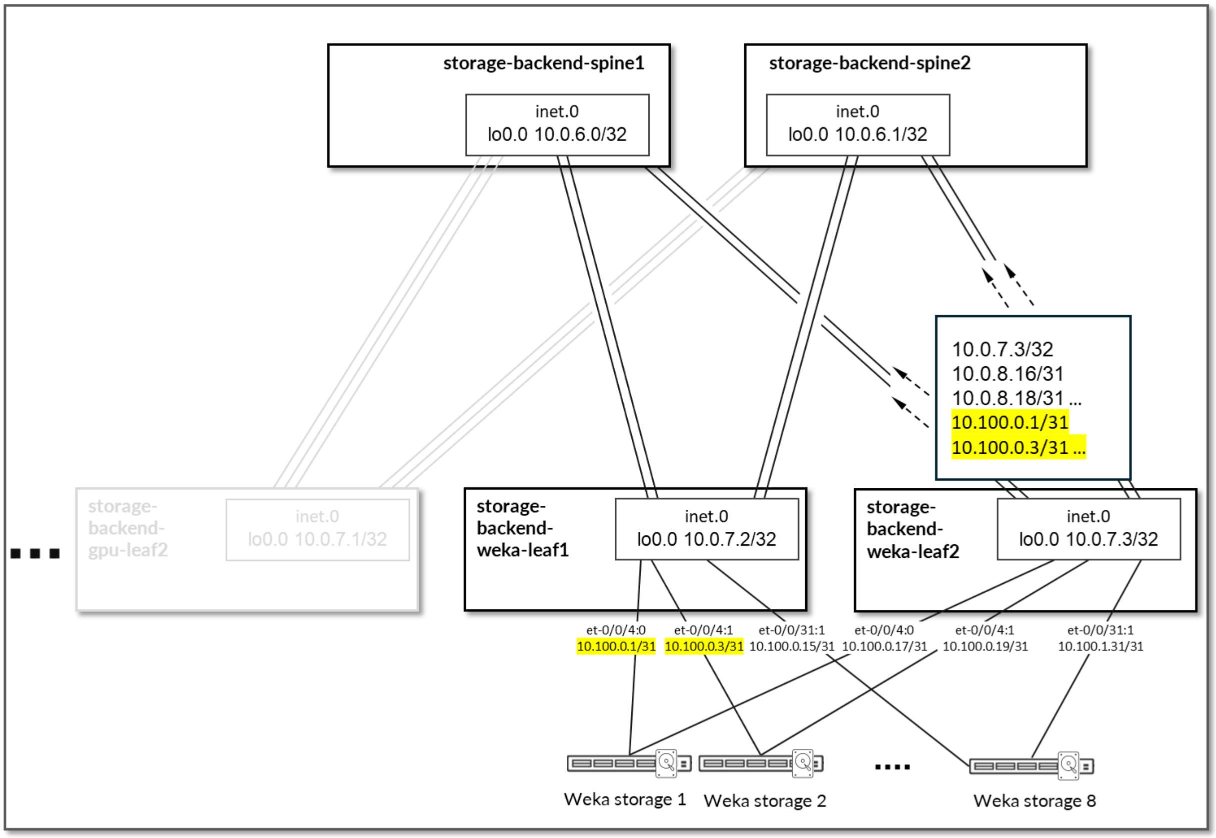


Table 48: Storage Backend Spine Node Advertised Routes

Spine Node	Peer	Advertised Routes			BGP Communities
storage-backend-spine1	*storage-backend-gpu-leaf1*	10.0.6.0/32 10.0.7.1/32 10.0.7.2/32 10.0.7.3/32	10.0.8.6/31 10.0.8.8/31 10.0.8.10/31 10.0.8.12/31 10.0.8.14/31 ...	10.100.0.0/31 10.100.0.2/31 ... 10.100.2.0/31 10.100.2.2/31 ...	3:20007 21001:26000

(Continued)

Spine Node	Peer	Advertised Routes			BGP Communities
	storage-backend-gpu-leaf2	10.0.6.0/32 10.0.7.0/32 10.0.7.2/32 10.0.7.3/32	10.0.8.0/31 10.0.8.2/31 10.0.8.4/31 10.0.8.12/31 10.0.8.14/31 ...	10.100.0.0/31 10.100.0.2/31 ... 10.100.1.0/31 10.100.1.2/31 ...	
	storage-backend-weka-leaf 1	10.0.6.0/32 10.0.7.0/32 10.0.7.1/32 10.0.7.3/32	10.0.8.0/31 10.0.8.2/31 10.0.8.4/31 ...	10.100.0.0/31 10.100.0.2/31 ... 10.100.1.0/31 10.100.1.2/31 ... 10.100.2.0/31 10.100.2.2/31 ...	

(Continued)

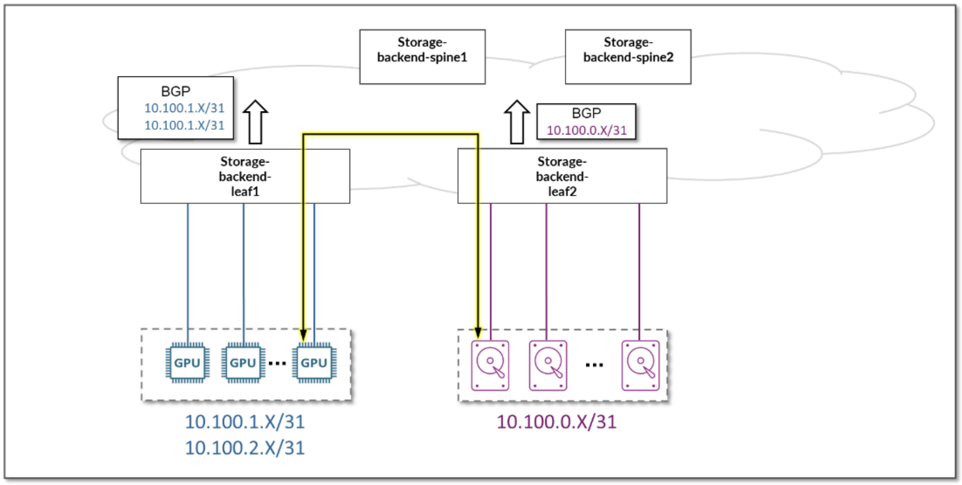
Spine Node	Peer	Advertised Routes			BGP Communities
	storage-backend-weka-leaf 2	10.0.6.0/32 10.0.7.0/32 10.0.7.1/32 10.0.7.2/32	10.0.8.0/31 10.0.8.2/31 10.0.8.4/31 10.0.8.20/31 ...	10.100.0.0/31 10.100.0.2/31 ... 10.100.1.0/31 10.100.1.2/31 ... 10.100.2.0/31 10.100.2.2/31 ...	
storage-backend-spine2	*storage-backend-gpu-leaf1*	10.0.6.1/32 10.0.7.1/32 10.0.7.2/32 10.0.7.3/32	10.0.8.6/31 10.0.8.8/31 10.0.8.10/31 10.0.8.12/31 10.0.8.14/31 ...	10.100.0.0/31 10.100.0.2/31 ... 10.100.2.0/31 10.100.2.2/31 ...	4:20007 21001:26000
	storage-backend-gpu-leaf2	10.0.6.1/32 10.0.7.0/32 10.0.7.2/32 10.0.7.3/32	10.0.8.0/31 10.0.8.2/31 10.0.8.4/31 10.0.8.12/31 10.0.8.14/31 ...	10.100.0.0/31 10.100.0.2/31 ... 10.100.2.0/31 10.100.2.2/31 ...	

(Continued)

Spine Node	Peer	Advertised Routes			BGP Communities
	storage-backend-weka-leaf 1	10.0.6.1/32 10.0.7.0/32 10.0.7.1/32 10.0.7.3/32	10.0.8.0/31 10.0.8.2/31 10.0.8.4/31 ...	10.100.0.0/31 10.100.0.2/31 ... 10.100.1.0/31 10.100.1.2/31 ... 10.100.2.0/31 10.100.2.2/31 ...	
	storage-backend-weka-leaf 2	10.0.6.0/32 10.0.7.1/32 10.0.7.2/32 10.0.7.3/32	10.0.8.6/31 10.0.8.8/31 10.0.8.10/31 10.0.8.12/31 10.0.8.14/31 ...	10.100.0.0/31 10.100.0.2/31 ... 10.100.2.0/31 10.100.2.2/31 ...	

By advertising the subnet assigned to the links between the leaf nodes and the GPU/storage servers, communication between GPUs and the storage servers is possible across the fabric.

Figure 97: Storage Subnet Advertisement



NOTE: All the devices are configured to perform ECMP load balancing, as explained later in the document.

WEKA Storage Solution

IN THIS SECTION

The WEKA Data Platform is a software-based solution built to modernize enterprise data stacks. Its advanced AI-native, data pipeline-oriented architecture delivers high performance at scale, so AI workloads run faster and work more efficiently.

We selected the WEKA Data Platform as part of the AI JVD design due to the following benefits:

- **High Performance**: Weka's architecture is designed for extreme performance, making it suitable for AI/ML workloads, big data analytics, and high-performance computing (HPC) environments.
- **Scalability**: Weka can scale from a few terabytes to exabytes of data, allowing customers to grow their storage capacity without compromising performance. WEKA's distributed architecture differs from typical scale-up style storage systems, appliances, and hypervisor-based, software-defined storage solutions. It overcomes traditional storage scaling and file-sharing limitations that can be a bottleneck to large-scale AI deployments making one of the preferred choices for customers.
- **Unified Storage**: Weka provides a single storage solution that can support multiple protocols (e.g., NFS, SMB, POSIX, S3), providing flexibility to access and manage the data and allowing Nvidia's GPUDirect Storage access.
- **Data Resilience**: Weka offers advanced data protection features, including erasure coding, which ensures data resilience and protection against hardware failures. With a minimum configuration of six storage servers the cluster can survive two-server failure.
- **Ease of Management**: Weka's software-defined storage solution is easy to deploy and manage, with a user-friendly interface and automated management features. It can be installed on any standard AMD EPYC™ or Intel Xeon™ Scalable Processor-based hardware with the appropriate memory, CPU processor, networking, and NVMe solid-state drives.
- **Support for GPUs**: Weka is optimized for GPU acceleration, making it an ideal storage solution for environments that heavily rely on GPU computing, such as AI and machine learning applications.
- **Low Latency**: The architecture of Weka allows for very low-latency access to data, which is crucial for applications that require real-time data processing.

Weka storage cluster in the AI JVD lab

We built the WEKA storage cluster with eight SuperMicro-based servers connected to the Storage Backend fabric providing **242TB** of usable storage. WEKA recommends eight cluster nodes and requires a minimum of six nodes for production deployment.

Each WEKA Server has the following specifications

- AMD EPYC 9454P processors
- 384GB System Memory
- OS drives: 2x 1.92TB M.2 NVMe Data Center SSD (PCIe 4.0)
- Data drives: 7x 7.68TB U.2 NVMe Data Center SSD (PCIe 4.0)
- Onboard OOB network connection (RJ45) and the following additional interface cards:

    - 1 x NVIDIA Mellanox ConnectX-6 DX Adapter Card, 100GE, dual-port QSFP28, PCIe 4.0 x16
    - 2 x NVIDIA Mellanox ConnectX-6 VPI Adapter Card, HDR IB & 200GE, dual-port QSFP56, OCP 3.0
- Software:
    - The operating system installed is Ubuntu 22.04 LTS.
    - WEKA release version tested in this design is 4.2.5.
    - WEKA Flash Tier license w/SnapShot and high-performance protocol services
    - (POSIX, NFS-W, S3 and SMB-W)

Common Setting Changes Required

WEKA strongly recommends certain BIOS settings, and that Mellanox drivers are matched across all nodes. For convenience, these changes are documented here.

NOTE: WEKA makes available a Weka Management Service (WMS) tool that can be used to automate the BIOS settings changes, verify your configuration, including driver revisions, and deploy the WEKA version you have. This can be downloaded from the WEKA website, located here: https://get.weka.io/ui/wms/download. Juniper highly recommends utilizing the WMS for configuring the WEKA cluster. All the devices are configured to perform ECMP load balancing, as explained later in the document.

BIOS settings:

The BIOS settings can be changed by applying the bios_settings.yml:

-
```
Supermicro:

  AMD:

    ACPISRATL3CacheAsNUMADomain#0099: Disabled

    IOMMU#00EA: Disabled

    NUMANodesPerSocket#703F: Auto

    SMTControl#00CB: Disabled

    SR-IOVSupport#0067: Enabled
```

```
DFCstates#7104: Disabled

GlobalC-stateControl#00CD: Disabled
```

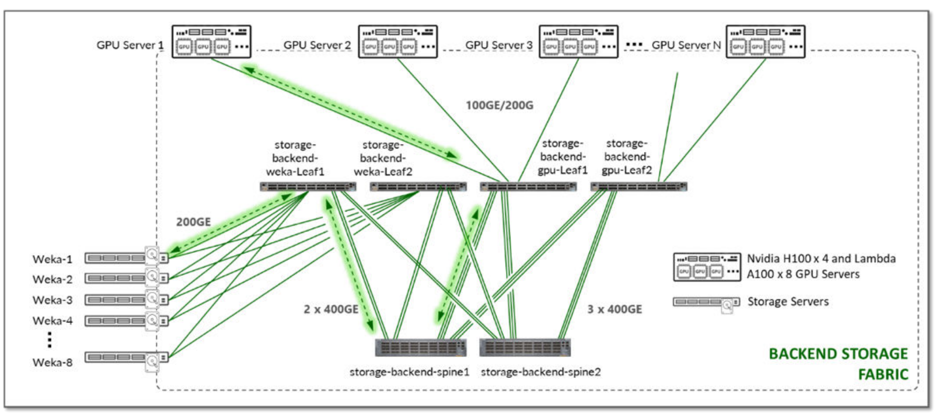
This is an AMD CPU-powered cluster; the settings may be different for Intel based CPUs.

For more details on how to apply these changes refer to: GitHub - weka/bios_tool: A tool for viewing/ setting bios_settings for Weka servers

Network Configuration for the Juniper WEKA Cluster

As described in the Storage Backend sections, the WEKA servers are dual-homed, and are connected to separate storage backend switches (***storage-backend-weka-leaf*** 1 and ***storage-backend-weka-leaf*** 2) using 200GE ports in the NVIDIA Mellanox ConnectX-6 VPI Adapter Card. The additional QSFP28 100Gbe ports are not used in this JVD but can be used for front-end ingress/egress traffic, staging and management.

Figure 98: Storage Interface Connectivity



The ports on the switch side must be configured with no auto negotiation and set to 200G speed.

OFED Drivers:

WEKA recommends following Nvidia's recommendation for OFED (Mellanox) drivers when using Connect-X cards. NVIDIA Documentation - Installing Mellanox OFED.

Driver Release Should be 5.8 or Later

Ensure that all versions for OFED drivers are aligned across all nodes in the WEKA cluster (i.e. ensure weka01 has the appropriate OFED installed).

For Ubuntu, the following command is recommended:

./mlnxofedinstall --force --dkms --all.

The following script can also be run (as root) on all machines to set the appropriate Mellanox firmware settings.

- ```
  #!/bin/bash


  mst start

  for MLXDEV in /dev/mst/* ; do

      mlxconfig -d ${MLXDEV} -y s ADVANCED_PCI_SETTINGS=1 PCI_WR_ORDERING=1

      mlxfwreset -y -d ${MLXDEV} reset

  done

  netplan apply

  mst stop
  ```

Best Practices for WEKA Data Platform with Juniper Switches

Our cluster is configured using the WEKA distributed POSIX client, which requires some tuning to be integrated to the rest of the design.

We recommend the following:

- Set the MTU to 9000
- If the back-end storage fabric is shared with another resource, set up appropriate CoS prioritization to ensure the AI ingest and checkpoint traffic is not interrupted by other applications network I/O requests.
  ```

If GPU Direct Storage is being used instead of the WEKA distributed POSIX client, congestion management and mitigation capability on the network utilizing Explicit Congestion Notification (ECN) and Priority Flow Control (PFC) must be set up.

WEKA also provides tools that can be used to test and measure network activity from a WEKA system perspective.

The command line tool ‘weka stats’ reports a percentage output of ‘good’ network performance.

- ```
  weka stats --start-time -24h --end-time -1m --show-internal  --stat
  GOODPUT_TX_RATIO,GOODPUT_RX_RATIO
  ```

When the output is shown as a percentage, anything below 85% indicates potential issues that require further examination.

Examples:

- ```
  NODE  CATEGORY  TIMESTAMP              STAT                      VALUE
  all   network   2024-06-14T12:58:00  GOODPUT_RX_RATIO  99.7636 %
  all   network   2024-06-14T12:58:00  GOODPUT_TX_RATIO  99.7636 %
  all   network   2024-06-14T12:57:00  GOODPUT_RX_RATIO  99.7663 %
  all   network   2024-06-14T12:57:00  GOODPUT_TX_RATIO  99.7663 %
  all   network   2024-06-14T12:56:00  GOODPUT_RX_RATIO  99.752 %
  all   network   2024-06-14T12:56:00  GOODPUT_TX_RATIO  99.752 %
  all   network   2024-06-14T12:55:00  GOODPUT_RX_RATIO  99.7578 %
  all   network   2024-06-14T12:55:00  GOODPUT_TX_RATIO  99.7578 %
  all   network   2024-06-14T12:54:00  GOODPUT_RX_RATIO  99.7795 %
  all   network   2024-06-14T12:54:00  GOODPUT_TX_RATIO  99.7795 %
  all   network   2024-06-14T12:53:00  GOODPUT_RX_RATIO  99.7685 %
  ```

```
all   network   2024-06-14T12:53:00  GOODPUT_TX_RATIO  99.7685 %

all   network   2024-06-14T12:52:00  GOODPUT_RX_RATIO  99.775 %

all   network   2024-06-14T12:52:00  GOODPUT_TX_RATIO  99.775 %
```

weka stats --category=network --show-internal --stat DROPPED_PACKETS --start-time -24h --end-time -1m -Z

- ```
  NODE  CATEGORY  TIMESTAMP               STAT               VALUE

  all   network   2024-06-14T13:06:00  DROPPED_PACKETS  0 Packets/Sec

  all   network   2024-06-14T13:05:00  DROPPED_PACKETS  0 Packets/Sec

  all   network   2024-06-14T13:04:00  DROPPED_PACKETS  0 Packets/Sec

  all   network   2024-06-14T13:03:00  DROPPED_PACKETS  0 Packets/Sec

  all   network   2024-06-14T13:02:00  DROPPED_PACKETS  0 Packets/Sec

  all   network   2024-06-14T13:01:00  DROPPED_PACKETS  0 Packets/Sec

  all   network   2024-06-14T13:00:00  DROPPED_PACKETS  0 Packets/Sec

  all   network   2024-06-14T12:59:00  DROPPED_PACKETS  0 Packets/Sec

  all   network   2024-06-14T12:58:00  DROPPED_PACKETS  0 Packets/Sec

  all   network   2024-06-14T12:57:00  DROPPED_PACKETS  0 Packets/Sec

  all   network   2024-06-14T12:56:00  DROPPED_PACKETS  0 Packets/Sec

  all   network   2024-06-14T12:55:00  DROPPED_PACKETS  0 Packets/Sec

  all   network   2024-06-14T12:54:00  DROPPED_PACKETS  0 Packets/Sec

  all   network   2024-06-14T12:53:00  DROPPED_PACKETS  0 Packets/Sec
  ```

If the weka stats command reports dropped packets as shown, further investigation is warranted.
  ```

More details and additional tools can be found on the WEKA website Manually prepare the system for WEKA configuration | W E K A.

Test Objectives

The primary objectives of the JVD testing can be summarized as:

- Qualification of the complete AI fabric design functionality including the Frontend, GPU Backend, and Storage Backend fabrics, and connectivity between NVIDIA GPUs and WEKA Storage.
- Qualification of the deployment steps based on Juniper Apstra.
- Ensure the design is well-documented and will produce a reliable, predictable deployment for the customer.

The qualification objectives included validating:

- validation of blueprint deployment, device upgrade, incremental configuration pushes/provisioning, Telemetry/Analytics checking, failure mode analysis, congestion avoidance and mitigation, and verification of host, storage, and GPU traffic.

Test Goals

The AI JVD testing for the described network included the following:

- Design and blueprint deployment through Apstra of three distinct fabrics
- Fabric operation and monitoring through Apstra analytics and telemetry dashboard
- Congestion management with PFC and ECN, including failure scenarios
- End-to-end traffic flow, with Dynamic Load Balancing
- System health, ARP, ND, MAC, BGP (route, next hop), interface traffic counters, and so on
- Software operation verification (no anomalies, or issues found)
- AI fabric with Juniper Apstra successfully performing under the following required scenarios (must):
    - Node failure (reboot)
    - Interface failures (interface down/up, Laser on/off):

Under these scenarios the following were evaluated/validated:

- Completion of AI Job models within MLCommons Training benchmarks
- Traffic recovery was validated after all failure scenarios.
- impact to the fabric and check anomalies reporting in Apstra.

Other features tested:

- Mellanox Connect-X NIC card default settings.
- DSCP and CNP configuration on the NICs
- Connectivity between fabric-connected hosts created by Apstra towards NSX-managed hosts.
- BERT/DLRM test completion times
- Llama2 Inference against existing infrastructure.

Refer to the test report for more information.

Tested Optics

Table 49: Frontend Fabric Optics

Frontend Fabric				
Part number	**Optics Name**	**Device Role**	**Device Model**	**Interface/NIC type**
740-085351	QSFP56-DD-400GBASE-DR4	SPINE	QFX5130-32CD	QSFP-DD
740-085351	QSFP56-DD-400GBASE-DR4	LEAF	QFX5130-32CD	QSFP-DD
740-061405	QSFP-100GBASE-SR4-T2	LEAF	QFX5130-32CD	QSFP28
740-046565	QSFP+-40G-SR4 w/ 4x10G breakout cable.	LEAF	QFX5130-32CD	QSFP+

AFBR-709SMZ	AVAGO 10GBASE-SR SFP+ 300m	Serv er	SuperMicro Headend Server	Intel Corporation Ethernet Controller X710 for 10GbE SFP+ (rev 01)
AFBR-89CDDZ	AVAGO 100GbE QSFP28 300m	Serv er	Weka Storage Server	ConnectX-6 Dx
AFBR-89CDDZ	AVAGO 100GbE QSFP28 300m	Serv er	SuperMicro A100 HGX Server	ConnectX-6 Dx
AFBR-89CDDZ	AVAGO 100GbE QSFP28 300m	Serv er	NVIDIA H100 DGX Server	ConnectX-7

Table 50: Storage Fabric Optics

Storage Fabric				
Part number	**Optics Name**	**Device Role**	**Device Model**	**Interface/NIC type**
740-0853 51	QSFP56-DD-400GBASE-DR4	SPINE	QFX5220-32CD	QSFP-DD
740-0853 51	QSFP56-DD-400GBASE-DR4	LEAF	QFX5220-32CD	QSFP-DD
740-0587 34	QSFP-100GBASE-SR4	LEAF	QFX5220-32CD	QSFP28
720-1287 30	QSFP56-DD-2x200GBASE-CR4-CU-2.5M w/ 400G DAC Breakout into 2X200G	LEAF	QFX5220-32CD	QSFP-DD
740-0614 05	QSFP-100GBASE-SR4	LEAF	QFX5220-32CD	QSFP28
720-1287 30	QSFP56-DD-2x200GBASE-CR4-CU-2.5M	Server	Weka Storage Server	ConnectX-6

720-1287 30	QSFP56-DD-2x200GBASE-CR4-CU-2.5M	Server	SuperMicro A100 HGX Server	ConnectX-6
740-1590 03	QSFP56-DD-2x200G-AOCBO-7M	Server	NVIDIA H100 DGX Server	ConnectX-7

Table 51: Backend GPU Fabric - Cluster 1

Backend GPU Fabric - Cluster 1 (HGX-A100)				
Part number	**Optics Name**	**Device Role**	**Device Model**	**Interface/NIC type**
740-0853 51	QSFP56-DD-400GBASE-DR4	SPINE	QFX5230-64CD	QSFP-DD
740-0853 51	QSFP56-DD-400GBASE-DR4	SPINE	PTX10008	QSFP-DD
740-0853 51	QSFP56-DD-400GBASE-DR4	LEAF	QFX5230-64CD	QSFP-DD
740-0465 65	QSFP+-40G-SR4 w/ 4x10G breakout cable.	LEAF	QFX5230-64CD	QSFP+
740-1590 02	QSFP56-DD-2x200G-BOAOC-5M	LEAF	QFX5230-64CD	QSFP-DD
720-1287 30	QSFP56-DD-2x200GBASE-CR4-CU-2.5M w/ 400G DAC Breakout into 2X200G.	LEAF	QFX5230-64CD	QSFP-DD
740-0853 51	QSFP56-DD-400GBASE-DR4	LEAF	QFX5220-32CD	QSFP-DD
720-1287 30	QSFP56-DD-2x200GBASE-CR4-CU-2.5 w/ 400G DAC Breakout into 2X200G	LEAF	QFX5220-32CD	QSFP-DD

720-128730	QSFP56-DD-2x200GBASE-CR4-CU-2.5M	Server	SuperMicro A100 HGX Server	ConnectX-7

Table 52: Backend GPU Fabric - Cluster 2

Backend GPU Fabric - Cluster 2 (DGX-H100)				
Part number	**Optics Name**	**Device Role**	**Device Model**	
740-174933	OSFP-800G-DR8	SPINE	QFX5240-64OD QFX5241-64OD	OSPF800
740-174933	OSFP-800G-DR8	LEAF	QFX5240-64OD QFX5241-64OD	OSPF800
MMS4X00-NS-FLT	NVIDIA 800Gbps Twin-port OSFP 2x400Gb_s Single Mode 2xDR4 100m	Server	NVIDIA H100 DGX Server	ConnectX-7

Results Summary and Analysis

For a detailed test results report, contact your Juniper representative.

Recommendations

The AI Data Center Network with Juniper Apstra, NVIDIA GPUs, and WEKA Storage JVD follows an industry-standard dedicated IP Fabric design. Three distinct fabrics provide maximum efficiency while maintaining focus on AI model scale, expedited completion times, and rapid evolution with the advent of AI technologies.

To follow best practice recommendations:

- A minimum of 4 spines in each fabric is suggested.

Though the design for cluster 1 in this document only includes only 2 spines, we found that under certain dual failure scenarios, combined with congestion, the fabric becomes susceptible to PFC storms (not vendor-unique). We recommend deploying the solution with 4 spines as described for the QFX5240/QFX5241 fabric (cluster 2) even when using different switch models.

- Follow a rail-optimized fabric and maintain a 1:1 relation with bandwidth subscription and Leaf to GPU symmetry.
- Implement Dynamic Load Balancing instead of traditional ECMP for optimal load distribution.
- Implement DCQCN (PFC and ECN) to ensure a lossless fabric in the GPU Backend Fabric, and possibly in the Storage Backend Fabric as required per vendor recommendation.
- The minimum recommended Junos OS releases for this JVD are:
- Junos OS Release 23.4R2-S3 is for the Juniper QFX5130-32CD
- Junos OS Release 23.4X100-D20 for the Juniper QFX5220-32CD
- Junos OS Release 23.4X100-D20 for the Juniper QFX5230-64CD
- Junos OS Release 23.4X100-D20 for the Juniper QFX5240-64CD
- Junos OS Release 23.4X100-D42 for the Juniper QFX5241-64CD
- Junos OS Release 23.4R2-S3 for the Juniper PTX10008
- Configure DCQCN (PFC and ECN) parameters on the Nvidia servers and change the NCCL_SOCKET interface to be the management (frontend) interface.

The Juniper hardware listed in the Juniper Hardware and Software Components section are the best-suited switch platforms regarding features, performance, and the roles specified in this JVD.

Revision History

Table 53: Revision History

Date	Version	Description
Dec 2025	JVD-AICLUSTERDC-AIML-02-09	Added QFX5241 and GLB configuration. Update Rail Optimized Section.

December 2024	JVD-AICLUSTERDC-AIML-02-08	Added PTX as spine.
November 2024	JVD-AICLUSTERDC-AIML-02-05	Utilized Junos OS Evolved Release 23.4X100-D20 for the leaf and spine switches.