

# Juniper Cloud Native Router User Guide

Published  
2023-02-24

Juniper Networks, Inc.  
1133 Innovation Way  
Sunnyvale, California 94089  
USA  
408-745-2000  
[www.juniper.net](http://www.juniper.net)

Juniper Networks, the Juniper Networks logo, Juniper, and Junos are registered trademarks of Juniper Networks, Inc. in the United States and other countries. All other trademarks, service marks, registered marks, or registered service marks are the property of their respective owners.

Juniper Networks assumes no responsibility for any inaccuracies in this document. Juniper Networks reserves the right to change, modify, transfer, or otherwise revise this publication without notice.

*Juniper Cloud Native Router User Guide*

Copyright © 2023 Juniper Networks, Inc. All rights reserved.

The information in this document is current as of the date on the title page.

## YEAR 2000 NOTICE

Juniper Networks hardware and software products are Year 2000 compliant. Junos OS has no known time-related limitations through the year 2038. However, the NTP application is known to have some difficulty in the year 2036.

## END USER LICENSE AGREEMENT

The Juniper Networks product that is the subject of this technical documentation consists of (or is intended for use with) Juniper Networks software. Use of such software is subject to the terms and conditions of the End User License Agreement ("EULA") posted at <https://support.juniper.net/support/eula/>. By downloading, installing or using such software, you agree to the terms and conditions of that EULA.

# Table of Contents

1

## **Juniper Cloud-Native Router (JCNR)**

**Juniper Cloud-Native Router - Overview | 2**

**Juniper Cloud-Native Router Controller (cRPD) | 9**

**JCNR-vRouter | 13**

**JCNR-CNI | 29**

2

## **Juniper Cloud-Native Router – Features**

**Cloud-Native Router Common Features | 34**

Juniper Cloud-Native Router Interface Types | 34

Logging and Notifications | 38

Juniper Cloud-Native Router Licensing | 41

Useful CLI Commands | 42

**Cloud-Native Router L2 Features | 45**

Juniper Cloud-Native Router Deployment Modes | 45

Juniper Cloud-Native Router L2 Interface Types | 46

L2 Metrics and Telemetry | 50

L2 ACLs (Firewall Filters) | 56

MAC Learning and Aging | 59

BUM Rate Limiting | 61

L2 API to Force Bond Link Switchover | 61

L2 Quality of Service (QoS) | 62

**Cloud-Native Router L3 Features | 65**

Juniper Cloud-Native Router Deployment Modes | 66

Juniper Cloud-Native Router Security Groups | 66

Juniper Cloud-Native Router Interface Types | 67

Security Groups | 70

L2 API to Force Bond Link Switchover | 71

MPLS Support in Juniper Cloud-Native Router | 71

## Juniper Cloud-Native Router (JCNR) - Examples

**L2 - Add User Pod with Kernel Access to a Cloud-Native Router Instance | 73**

Overview | 73

Before You Begin | 74

Detailed Steps | 75

**L2 - Add User Pod with virtio Trunk Ports to a Cloud-Native Router Instance | 83**

Overview | 83

Before You Begin | 84

Detailed Steps | 86

**L3 - Add User Pod to a Cloud-Native Router Instance | 93**

Overview | 93

Before You Begin | 94

Detailed Steps | 96

# 1

PART

## Juniper Cloud-Native Router (JCNR)

---

[Juniper Cloud-Native Router - Overview](#) | 2

[Juniper Cloud-Native Router Controller \(cRPD\)](#) | 9

[JCNR-vRouter](#) | 13

[JCNR-CNI](#) | 29

---

# Juniper Cloud-Native Router - Overview

## IN THIS CHAPTER

- [Overview | 2](#)
- [Benefits of Juniper Cloud-Native Router | 2](#)
- [Kubernetes | 3](#)
- [Juniper Cloud-Native Router Components | 4](#)
- [Ports Used by Cloud-Native Router | 7](#)

## Overview

The Juniper® Cloud-Native Router (cloud-native router) is a container-based software solution, orchestrated by Kubernetes. Cloud-native router combines the containerized routing protocol process (cRPD) and a Data Plane Development Kit (DPDK)-enabled Contrail® Networking™ vRouter (vRouter). With the cloud-native router, you can enable full Junos-based control plane with the enhanced forwarding capabilities of the DPDK-enabled vRouter.

## Benefits of Juniper Cloud-Native Router

- You can deploy the cloud-native router in either L2 (switch) or L3 (routing) mode
- Higher packet forwarding performance with DPDK-enabled vRouter
- Easy deployment on general-purpose compute devices
- Full routing and forwarding stacks in software
- Out-of-the-box software-based open radio access network (O-RAN) support
- IPv4 and IPv6 routing and forwarding
- Quick spin-up with containerized deployment on Kubernetes

- Highly scalable solution

## Kubernetes

**NOTE:** Juniper Networks refers to primary nodes and backup nodes in our documentation. Kubernetes refers to master nodes and worker nodes. References in this guide to primary and backup correlate with master and worker in the Kubernetes world.

Let's talk a little about Kubernetes in this section. Kubernetes is an orchestration platform for running containerized applications in a clustered computing environment. Kubernetes provides automatic deployment, scaling, networking, and management of containerized applications. Because Juniper Cloud-Native Router is a container-based solution, we've chosen Kubernetes as the orchestration platform. For complete details about Kubernetes, including installation, cluster creation, management, and maintenance, see <https://kubernetes.io/>.

The major components of a Kubernetes cluster are:

- **Nodes**

Kubernetes uses two types of nodes: a primary (control) node and a compute (worker) node. A Kubernetes cluster usually consists of one or more master nodes (in active/standby mode) and one or more worker nodes. You create a node on a physical computer or a virtual machine (VM).

**NOTE:** In Juniper Cloud-Native Router Release 22.X, you must provide a working, single-node Kubernetes cluster. Cloud-native router does not support multinode clusters, with primary and secondary nodes on separate VMs or bare-metal server (BMS).

- **Pods**

Pods live in nodes and provide a space for containerized applications to run. A Kubernetes pod consists of one or more containers, with each pod representing an instance of the application(s). A pod is the smallest unit that Kubernetes can manage. All containers in a pod share the same network namespace.

- **Namespaces**

In Kubernetes, pods operate within a namespace to isolate groups of resources within a cluster. All Kubernetes clusters have a *kube-system* namespace, which is for objects created by the Kubernetes system. Kubernetes also has a *default* namespace, which holds all objects that don't provide their own namespace. The last two preconfigured Kubernetes namespaces are *kube-public* and *kube-node-lease*. The **kube-public** namespace is used to allow unauthenticated users to read some aspects

of the cluster. Node leases allow the **kubelet** to send heartbeats so that the control plane can detect node failure.

In Juniper Cloud-Native Router Release 22.X, some of the pods run in the *kube-system* namespace while others provide their own namespace.

- **Kubelet**

The kubelet is the primary node agent that runs on each node. In the case of Juniper Cloud-Native Router, only a single kubelet runs on the cluster since we do not support multinode deployments.

- **Containers**

A container is a single package that consists of an entire runtime environment including the application and its:

- Configuration files
- Dependencies
- Libraries
- Other binaries

Software that runs in containers can, for the most part, ignore the differences in the those binaries, libraries, and configurations that may exist between the container environment and the environment that hosts the container. Common container types are docker, containerd, and Container Runtime Interface using Open Container Initiative compatible runtimes (CRI-O).

For Juniper Cloud-Native Router Release 22.X, docker is the only supported container type (container runtime).

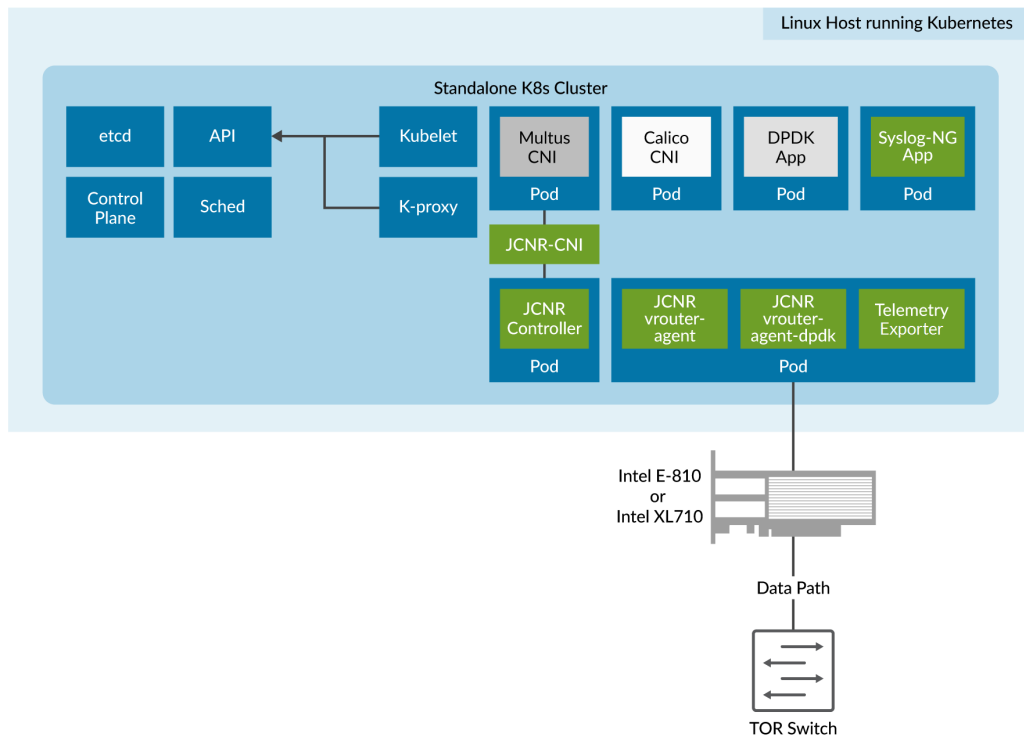
## Juniper Cloud-Native Router Components

The Juniper Cloud-Native Router solution consists of several components. This section provides a brief overview of the components of Juniper Cloud-Native Router.

[Figure 1 on page 5](#) shows the components of the Juniper Cloud-Native Router inside a Kubernetes cluster. The green-colored components are specific to the Juniper Cloud-Native Router, while the others are required third-party components.



**Figure 1: Cloud-Native Router Components**



- **Juniper Cloud-Native Router Controller (JCNr-controller or cRPD)**

The cRPD acts as the control plane for the cloud-native router. It performs management functions and maintains configuration information for the vRouter forwarding plane. cRPD is based on the Junos OS control plane. You can configure the JCNr-controller using:

- YAML-formatted Helm charts
- Third-party management platforms that use the NETCONF protocol
- API calls to the cRPD MGD
- Direct CLI access to the cRPD Pod

This section is only applicable to the L3 version of JCNr.

You can configure the requisite protocols (IGPs and BGP), using NETCONF or CLI, to the JCNr-controller to provide reachability over tunnels. JCNr-controller establishes adjacencies for various protocols, learns and programs the forwarding information base (FIB, also known as forwarding table) using its routing protocols to the JCNr-vRouter-agent through gRPC services. JCNr-vRouter provides a bidirectional gRPC channel for communication with the JCNr-controller.

A typical routing update for the underlay network follows the process shown below:

- JCNr-controller learns of a new underlay network route on its vhost0 interface
- JCNr-controller sends a gRPC-encoded route message (IPv4 or IPv6) to the vRouter agent using its gRPC interface
- The vRouter agent performs ARP or NDP look up for the next-hop
- The vRouter agent encapsulates the next-hop into vRouter and waits for ACK message in return
- The vRouter agent programs the underlay route into vRouter and waits for ACK message in return

Once a learned underlay route is no longer valid, JCNr-controller sends a route delete message to the vRouter agent which signals the vRouter to delete the route and next-hop as needed.

A typical routing update for the overlay (Pod-to-Pod) network follows the process shown below:

- A new remote Pod route (Pod-to-Pod) is learned by JCNr-controller through BGP.
- JCNr-controller resolves the next-hop over an SR-MPLS tunnel whose label stack is populated by ISIS. This creates next-hop information for the Pod's IP that contains the service label and the transport labels associated with the MPLS tunnel.
- JCNr-controller then sends a gRPC-encoded route message to the vRouter agent that contains the pod\_ip, vrf name, next-hop IP, service label and between 0 and 4 transport labels.
- The vRouter agent resolves the next-hop IP using NDP or ARP depending on whether the address is IPv6 or IPv4 respectively.
- The vRouter agent creates an MPLS tunnel next-hop (if not already present) and programs it to the vRouter.
- The vRouter creates the MPLS tunnel next-hop, adds it to the next-hop table and sends an ACK message to the vRouter agent in response.
- The vRouter agent programs the Pod route in the Pod VRF along with the next-hop created in the previous step.
- The vRouter adds the route entry for the Pod in the Pod VRF along with a pointer to the next-hop. vRouter then sends an ACK message to vRouter agent in response.

When the overlay route is withdrawn, JCNr-controller sends a route delete message to vRouter agent which signals the vRouter to delete the route and next-hop information from the forwarding tables.

Access control lists (ACLs) are supported on JCNr-controller to configure the networking policy for application pods. The integration with JCNr-vRouter-agent means that these network policies are automatically shared with the JCNr-vRouter.

- **Juniper Cloud-Native Router -vRouter (JCNr-vRouter or vRouter)**

JCNR-vRouter acts as the forwarding, or data, plane for Juniper Cloud-Native Router. It interacts with the JCNR-controller through the vRouter-agent and receives and forwards packets through its various interfaces.

JCNR-vRouter enables applications built using the DPDK framework to send and receive packets directly between the application and vRouter without passing through the kernel.

The vRouter receives configuration and management information from JCNR-controller through the JCNR vRouter-agent using the gRPC protocol.

- **Juniper Cloud-Native Router-Container Network Interface (JCNR-CNI)**

JCNR-CNI is a Kubernetes CNI and is responsible for provisioning network interfaces for application Pods. vRouter acts as the data-plane for these application Pod interfaces. JCNR-CNI interacts with Kubernetes, JCNR-controller and JCNR-vRouter. JCNR-CNI manages the vRouter interface lifecycles and cRPD configuration. When you remove an application Pod, JCNR-CNI removes the corresponding interface configuration from cRPD and state information from the vRouter-DPDK forwarding plane.

## Ports Used by Cloud-Native Router

Juniper Cloud-Native Router listens on certain TCP and UDP ports. [Table 1 on page 7](#) shows the ports, protocols, and a description for each one.

**Table 1: Cloud-Native Router Listening Ports**

Protocol	Port	Description
TCP	8085	vRouter introspect–Used to gain internal statistical information about vRouter
TCP	8070	Telemetry information–Used to see telemetry data from cloud-native router
TCP	9091	vRouter health check–cloud-native router checks to ensure <b>contrail-vrouter-dpdk</b> process is running, etc.

Table 1: Cloud-Native Router Listening Ports *(Continued)*

Protocol	Port	Description
TCP	50052	gRPC port-JCNR listens on both IPv4 and IPv6
TCP	24	cRPD SSH
TCP	830	cRPD NETCONF
TCP	666	<b>rpd</b>
TCP	1883	Mosquito mqtt-Publish/subscribe messaging utility
TCP	9500	<b>agentd</b> on cRPD
TCP	21883	<b>na-mqttd</b>
TCP	50051	<b>jsd</b> on cRPD
TCP	51051	<b>jsd</b> on cRPD
UDP	50055	Syslog-NG

# Juniper Cloud-Native Router Controller (cRPD)

## IN THIS CHAPTER

- [Benefits of Juniper Cloud-Native Router Controller | 9](#)
- [Configuration Options | 10](#)
- [Access to the CLI | 10](#)

Read this chapter to understand the Juniper Cloud-Native Router-controller (cloud-native router controller or cRPD), which is the Juniper Cloud-Native Router control plane.

## Benefits of Juniper Cloud-Native Router Controller

The cRPD acts as the control plane in the Juniper Cloud-Native Router solution. The cRPD provides configuration interfaces to users (CLI) and applications (API) alike. You can use these interfaces to configure or program the Juniper Cloud-Native Router-vRouter forwarding plane. You can also configure the following using the Juniper Cloud-Native Router-controller:

- Virtual function (VF) fabric interfaces
- VF workload interfaces
- Trunk interfaces
- Access interfaces
- L2 ACLs (firewall rules)
- Bridge domains
- Ethernet switching
- VLANs

The cRPD performs the following functions:

- Supports JCNR-vRouter as the forwarding plane
- Maintains configuration for vRouter interfaces including trunk and access interfaces, virtual function interfaces (VFs), VLANs, and more
- Maintains configuration of bridge domains
- Maintains configuration for L2 firewall
- Maintains configuration for bridge domains, VLANs, virtual-switches, and so on
- Passes configuration information to the vRouter through the vRouter-agent
- Stores license key information

## Configuration Options

During deployment, you can configure the cRPD by changing the values of the *key:value* pairs contained within the **values.yaml** file that we include in the software distribution TAR file.

After deployment, we recommend that you use the NETCONF protocol with PyEZ to configure cRPD. See <https://www.juniper.net/documentation/us/en/software/junos-pyez/junos-pyez-developer/index.html> for details about PyEZ. Alternatively, you can SSH directly to the cRPD on TCP port 24 or using NETCONF on TCP port 830. Finally, you can also configure the cloud-native router by accessing the Junos CLI on the cRPD using Kubernetes commands to connect to the cRPD Pod.

## Access to the CLI

In this procedure, we provide CLI commands that you run on the host server. We do not show a prompt before the commands so that you can copy and paste the commands into your own cloud-native router.

```
kubect1 get pods -A
```

The output should look like:

NAMESPACE	NAME	READY	STATUS
RESTARTS	AGE		

contrail-deploy 0	contrail-k8s-deployer-7b5dd699b9-smqgn 37h	1/1	Running
contrail 0	contrail-vrouter-masters-htcv 37h	3/3	Running
default 0	delete-crpd-dirs--1-bjngd 37h	0/1	Completed
default 0	delete-vrouter-dirs--1-k5wgb 37h	0/1	Completed
default 0	odu-pktgen-trunkint 24h	1/1	Running
default 0	odu-subinterface-3003 7d	1/1	Running
kube-system 52 (8d ago)	calico-kube-controllers-57b9767bdb-76fvw 107d	1/1	Running
kube-system 18 (8d ago)	calico-node-pgljp 107d	1/1	Running
kube-system 38 (8d ago)	coredns-8474476ff8-2nbnv 107d	1/1	Running
kube-system 18 (8d ago)	dns-autoscaler-7f76f4dd6-8b4w5 107d	1/1	Running
kube-system 45 (8d ago)	kube-apiserver-nodem27.englab.juniper.net 107d	1/1	Running
kube-system 34 (8d ago)	kube-controller-manager-nodem27.englab.juniper.net 107d	1/1	Running
<b>kube-system 0</b>	<b>kube-crpd-worker-ds-89wzg 32h</b>	<b>1/1</b>	<b>Running</b>
kube-system 0	kube-multus-ds-amd64-f2pls 8d	1/1	Running
kube-system 18 (8d ago)	kube-proxy-vrqjm 107d	1/1	Running
kube-system 35 (8d ago)	kube-scheduler-nodem27.englab.juniper.net 107d	1/1	Running
kube-system 43 (8d ago)	node-local-dns-hm56k 107d	1/1	Running
kube-system 0	syslog-ng-54749b7b77-tqvpk 37h	1/1	Running

The command to access the cRPD CLI is in the form: `kubectl exec -n kube-system -it <full cRPD Pod name> -- bash`. If we use the output mentioned earlier in this section, the command appears as: `kubectl exec -n kube-system -it kube-crpd-worker-ds-89wzg -- bash`.

The output from the command (when you use the full name of your cRPD Pod) should look like:

```
Defaulted container "kube-crpd-worker" out of: kube-crpd-worker, jcnr-crpd-config (init),
install-cni (init)

===>
      Containerized Routing Protocols Daemon (CRPD)
      Copyright (C) 2020-2021, Juniper Networks, Inc. All rights reserved.
                                     <===
```

This output indicates that you have accessed the cRPD CLI. At this point, your access level is root and you are in shell mode. Just as when you connect as root to any Junos OS-based device, you must enter the `cli` command to access the Junos CLI in operation mode.



## CHAPTER 3

# JCNR-vRouter

## IN THIS CHAPTER

- Benefits of JCNR vRouter | 13
- Access vRouter CLI | 15
- L2 Mode Packet Flow in vRouter | 16
- Monitoring vRouter with CLI Commands | 18
- The **dropstats** Command | 22
- The **dppdkinfo** Command | 23
- Troubleshooting vRouter | 27

Read this chapter to understand more about JCNR-vRouter, which is the JCNR DPDK-based forwarding plane.

## Benefits of JCNR vRouter

- Integration of the DPDK into the JCNR-vRouter:
  - Forwarding plane provides faster forwarding capabilities than kernel-based forwarding
  - Forwarding plane is more scalable than kernel-based forwarding
  - Support for the following NICs:
    - Intel E810 (Columbiaville) with Intel Adapter Virtual Function (IAVF) and Dynamic Device Personalization (DDP)
    - Intel XL710 (Fortville) with Intel Adapter Virtual Function (IAVF)

**NOTE:** Dynamic Device Personalization (DDP) is not supported on the Intel XL710 NIC

- Support for the following drivers on Intel XL710 NICs:
  - Intel Adapter Virtual Function (IAVF)
  - Linux base driver i40E
- **Interface Support:**
  - **Pod interfaces using virtio**

You define these DPDK-enabled vhost-based user socket interfaces as workload interfaces in the `values.yaml` file. The system maintains the socket details for all active interfaces of this type in the `/dpdk` directory of the workload container. You usually define this type of pod interfaces as trunk interfaces rather than access interfaces.
  - **Pod interfaces using kernel veth pair**

You define pod interfaces with kernel veth pairs in the `values.yaml` file as access interfaces so the pod can pass traffic through the kernel rather than using DPDK.
  - **DPDK Virtual Function (VF) workload interfaces**

You define the DPDK VF fabric trunk interfaces in the `values.yaml` file that is used in the vRouter deployment. This makes JCNr aware of the names of the interfaces, their MAC addresses, and their PCI slot ID.
  - **DPDK VF fabric trunk interfaces**

You define the DPDK VF fabric trunk interfaces in the `values.yaml` file that is used in the vRouter deployment. This makes JCNr aware of the names of the interfaces, their MAC addresses, and their PCI slot ID. To keep traffic flows manageable, we apply VLAN filtering to the physical interfaces. When you configure VLANs for use with the cloud-native router, only the configured VLANs can pass through the physical interfaces.
  - **Physical Function (PF) workload interfaces**

You define PF workload interfaces in the `values_I3.yaml` file. The system equips each PF workload interfaces with only one receive and one transmit queue. The system assigns one forwarding CPU core to the task of polling the interface for traffic.
  - **PF fabric interfaces**

You define PF fabric interfaces in the `values.yaml` file. The system equips each PF fabric interface with as many receive and transmit queues as you assign forwarding CPU cores to handle the polling. For example, if you assign three forwarding CPU cores to the PF fabric interface, the system allocates three receive and three transmit queues to the PF fabric interface.
  - No `vhost0` interface when run in L2 mode

vRouter-agent detects L2 mode in **values.yaml**, so does not wait for the `vhost0` interface to come up before completing installation. There is no `vhost` interface add message sent so the vRouter doesn't create the `vhost0` interface.

- **Interface Bonding**

DPDK vRouter supports interface bonding in active/standby mode on DPDK VF fabric interfaces. The **values.yaml** file specifies the interface names, mode value, and primary and secondary interface designations. DPDK contains a library with its own bonding driver that it uses for bonding. In operation, the vRouter uses the primary interface to pass traffic. If the primary link goes down, the secondary interface in the pair passes traffic until the primary interface reconnects.

- **Pod DPDK Interfaces**

JCNR-vRouter supports virtio communication to the POD application. The JCNR-CNI allocates unique socket directories that it passes to Pod applications and to vRouter. JCNR-CNI ensures that one Pod cannot access the resources of another Pod through isolation of `vhost` sockets and Pod volume mounts.

- **Pod Kernel Interfaces**

JCNR supports the `veth` interface type to communicate with pod applications that use the Linux Kernel's networking stack.

## Access vRouter CLI

```
kubectl get pods -n contrail
```

The output of the command looks like:

NAME	READY	STATUS	RESTARTS	AGE
contrail-vrouter-masters-97v8z	3/3	Running	0	6d1h

To access the vRouter-agent CLI, you use the full pod name from your system in the following command:

```
kubectl exec -n contrail -it contrail-vrouter-masters-97v8z -- bash
```

The output of the command looks like: Defaulted container "contrail-vrouter-agent" out of: contrail-vrouter-agent, contrail-vrouter-agent-dpdk, contrail-vrouter-telemetry-exporter, contrail-init (init), contrail-vrouter-kernel-init-dpdk (init).

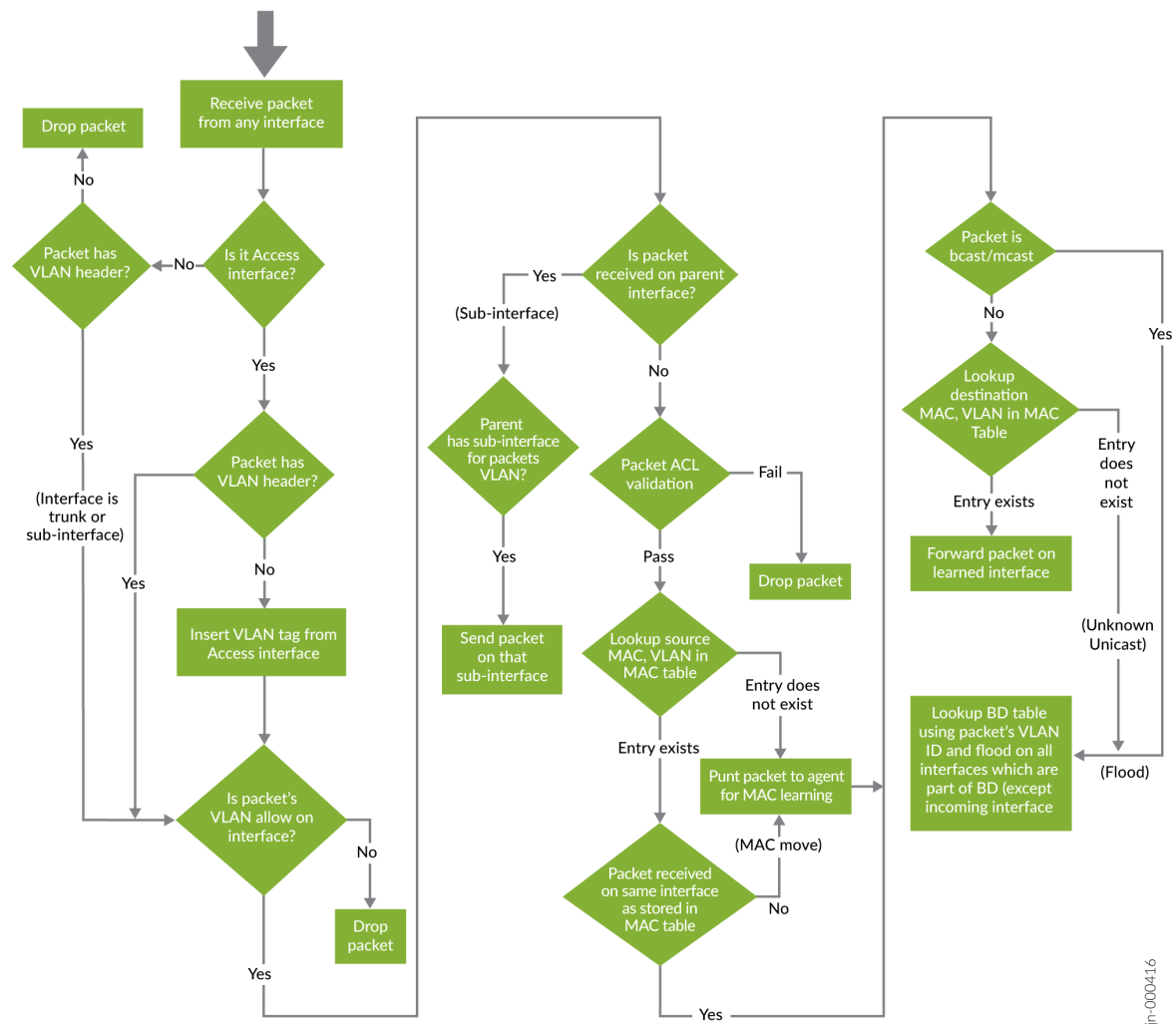
After you access the vRouter CLI, there are a number of commands that you can run to monitor and troubleshoot the system. We illustrate some of the available commands in ["Monitoring vRouter with CLI Commands" on page 18](#).

## L2 Mode Packet Flow in vRouter

To understand a switch or router, you must know what happens to packets as they flow through. This section describes the life of a packet in the vRouter when deployed in L2 mode. In this section, we show how the vRouter MAC and bridge domain (BD) tables are populated and introduces some of the CLI commands that you can use to see various parts of the vRouter from inside.

The flowchart [Figure 2 on page 17](#) illustrates one possible generic packet flow through cloud-native router. It does not cover all possible interactions with the packet.

Figure 2: The Life of a Packet in vRouter



The vRouter makes a lot of decisions about a received packet to ensure that the packet is handled correctly. Let's look at what the vRouter does with a packet. You can see in [Figure 2 on page 17](#) that there are several choices made based on the VLAN or BD. One of the tables that vRouter consults for making forwarding decisions is the [Table 2 on page 17](#).

Table 2: BD Table

VLAN ID (KEY)	Interface ID List (Value)
1024	2,3,4

**Table 2: BD Table (Continued)**

VLAN ID (KEY)	Interface ID List (Value)
1042	4,5
1022	1
1044	6

The BD table tells the vRouter which interfaces can carry traffic with a specific VLAN ID. Thus, the VLAN ID serves as the key for the table while the Interface ID List serves as the values for each entry.

The [Table 3 on page 18](#) is closely related to the BD table. The MAC table uses the MAC address and VLAN ID as a key pair. Then, the Interface ID and hit count serve as the values for each entry in the table as shown in this section.

**Table 3: MAC Table**

MAC Address	VLAN ID	Interface ID	Hit Count
00:11:22:33:44:55	1024	2	123234
00:22:33:44:55:66	1042	4	823948
00:33:44:55:66:77	1022	1	45980
00:44:55:66:77:88	1044	6	86578

The primary purpose of the MAC table is to map which MAC addresses can be reached through which interface. The vRouter makes entries in and consults the MAC table while processing packets.

## Monitoring vRouter with CLI Commands

In the vRouter, CLI commands are useful for troubleshooting and monitoring purposes. As mentioned in ["Access vRouter CLI" on page 15](#), you can access the CLI of the vRouter. By executing commands in that CLI, you can learn about various aspects of the running vRouter. The following examples assume that you have already connected to the vRouter CLI. The commands that we show in this section do not show a command prompt so that you can copy and paste them into your own vRouter.

We use the **purel2cli** command in most of the examples in this section. The command has more options than we show in the examples. In addition, the **purel2cli** has a help command that you can use to see the available options.

### The purel2cli Command

To see all the options of the **purel2cli** command in the vRouter CLI, execute the command with the **--help** option.

```
purel2cli --help
```

```
Usage: purel2cli [--mac show]
      [--vlan show]
      [--vlan get <VLAN_ID>]
      [--acl show <VLAN_ID>]
      [--acl reset-counters <VLAN_ID>]
      [--l2stats get <VIF_ID> <VLAN_ID>]
      [--clear VLAN_ID]
      [--sock-dir <sock dir>]
      [--help]
```

### See the Current Status of Your Running vRouter

To see the status of the vRouter, enter the following command in the vRouter CLI:

```
ps -eaf | grep dpdk
```

The output from the command above looks like: root 127 93 99 Jul29 ? 82-20:31:49 /contrail-vrouter-dpdk --no-daemon --socket-mem=1024 1024 --allow=0000:01:10.1 --allow=0000:01:10.0 --l2\_table\_size=10240 --yield\_option 0 --ddp --l2\_mode

The output contains several elements.

**Table 4: vRouter Status**

Flag	Meaning
--l2_mode	The vRouter is running in L2 mode.

Table 4: vRouter Status *(Continued)*

Flag	Meaning
--l2_table_size	The current number of entries in the MAC table. The default size is 10240 entries.
--allow=<PCI Id>	The PCI ID of fabric and fabric workload interfaces. More than one ID can appear in the output. These IDs serve as an allowlist.
--ddp	Enable Intel DDP support.  We enable DDP by default in the <b>values.yaml</b> file in the vRouter.  <b>NOTE:</b> The Intel XL710 NIC does not support DDP.

### Show MAC Table

The following command shows the MAC addresses that the vRouter has dynamically learned.

```
purel2cli --mac show
```

The output from the above command looks like:

```
=====
||  MAC            vlan    port    hit_count||
=====
00:01:01:01:01:03  1221    2       1101892
00:01:01:01:01:02  1221    2       1101819
00:01:01:01:01:04  1221    2       1101863
00:01:01:01:01:01  1221    2       1101879
5a:4c:4c:75:90:fe  1250    5        12
Total Mac entries 5
```



### Show Bridge Domain Table

The following command shows the VLAN to port mapping in the vRouter.

```
purel2cli --vlan show
```

The output from the above command looks like:

```
VLAN      PORT
=====
1201      1,2,3,4,
1202      1,2,3,4,
1203      1,2,3,4,
1204      1,2,3,4,
1205      1,2,3,4,
```

You can use the following form of the **purel2cli** command to see the bridge domain table entry for a specific VLAN: `purel2cli --vlan get <vlan-id>`

### Show L2 Statistics

There are several variations of the **purel2cli** command that allow you to display and filter L2 statistics in the vRouter. The base form of the command is: `purel2cli -- l2stats get <virtual_interface_ID> <VLAN_ID>`. The table [Table 5 on page 21](#) shows the available command options and what they do. This section also provides a sample output using one of the options.

**Table 5: purel2cli Command Options for L2 Statistics**

Sample Command	Function
<code>purel2cli --l2stats get '*' '*'</code>	Get statistics for all virtual interfaces (vif) and all VLAN IDs.
<code>purel2cli --l2stats get '*' 100</code>	Get statistics for all vif that are part of VLAN 100
<code>purel2cli --l2stats get 1 '*'</code>	Get statistics for all VLANs for which interface 1 is a member
<code>purel2cli --l2stats get 1 100</code>	Get statistics for interface 1 and VLAN 100

The following command is an example of the L2 statistics for interface 2 and VLAN 1221:

```
purel2cli --l2stats get 2 1221
```

```
Vlan id count: 1
```

```
-----  
Statistics for vif 2 vlan 1221  
-----
```

	Rx Pkts	Rx Bytes	Tx Pkts	Tx Bytes
Unicast	245344824	48152682842	835552	1667761792
Broadcast	0	0	0	0
Multicast	0	0	0	0
Flood	0	0	0	0

```
-----
```

### Clear L2 Statistics

The following example shows commands that allow you to clear L2 statistics information from the vRouter.

You can clear the statistics from the vRouter with the purel2cli command in the form: purel2cli --clear <VLAN\_ID>.

```
purel2cli --clear '*'
```

Clears all statistics from all VLANs in the vRouter.

```
purel2cli --clear 100
```

Clears all statistics for VLAN id 100.

## The dropstats Command

The vRouter tracks the packets that it drops and includes the reason for dropping them. [Table 6 on page 23](#) shows the common reasons for vRouter to drop a packet. When you execute the **dropstats** command, the vRouter does not show a counter if the count for that counter is 0.

Table 6: Dropstats Counters

Counter Name	Meaning
L2 bd table drop	No interfaces in bridge domain
L2 untag pkt drop	Untagged packet arrives on trunk or sub-interface
L2 Invalid Vlan	Packet VLAN does not match interface VLAN
L2 Mac Table Full	No more entries available in the MAC table
L2 ACL drop	Packet matched firewall filter (ACL) drop rule
L2 Src Mac lookup fail	Unable to match (or learn) the source MAC address

Example output from the **dropstats** command looks like:

```
dropstats
```

```

L2 bd table Drop          43
L2 untag pkt drop        716
L2 Invalid Vlan          7288253
Rate limit exceeded      673179706
L2 Mac Table Full        41398787
L2 ACL drop              8937037
L2 Src Mac lookup fail   247046

```

## The dpdkinfo Command

The **dpdkinfo** command provides insight into the status and statistics of DPDK. The **dpdkinfo** command has many options. The following sections describe the available options and the example output from the **dpdkinfo** command. You can run the **dpdkinfo** command only from within the vRouter-agent CLI.

**dpdkinfo Command Usage**

```
dpdkinfo

Usage: dpdkinfo [--help]
                --version|-v                               Show
DPDK Version
                --bond|-b                                   Show
Master/Slave bond information
                --lacp|-l    <all/conf>                   Show
LACP information from DPDK
                --mempool|-m  <all/<mempool-name>>         Show
Mempool information
                --stats|-n    <vif index value>            Show
Stats information
                --xstats|-x    <vif index value>            Show
Extended Stats information
                --lcore|-c                                       Show
Lcore information
                --app|-a                                       Show
App information
                --ddp|-d    <list> <list-flow>             Show DDP
information for X710 NIC
                --rx_vlan|-z  <value>                       Show
VLAN information
Optional: --buffsz    <value>                               Send
output buffer size (less than 1000Mb)
```

**dpdkinfo Lcore Information**

This command shows the Lcores assigned to DPDK VF fabric interfaces and the queue ID for each interface.

```
dpdkinfo -c

No. of forwarding lcores: 4

Lcore 10:
  Interface: 0000:18:01.1      Queue ID: 0
  Interface: 0000:18:0d.1      Queue ID: 0
  Interface: 0000:86:00.0      Queue ID: 0
```

```
Lcore 11:
  Interface: 0000:18:01.1      Queue ID: 1
  Interface: 0000:18:0d.1     Queue ID: 1
  Interface: 0000:86:00.0     Queue ID: 1

Lcore 12:
  Interface: 0000:18:01.1      Queue ID: 2
  Interface: 0000:18:0d.1     Queue ID: 2
  Interface: 0000:86:00.0     Queue ID: 2

Lcore 13:
  Interface: 0000:18:01.1      Queue ID: 3
  Interface: 0000:18:0d.1     Queue ID: 3
  Interface: 0000:86:00.0     Queue ID: 3
```

**dpdkinfo Memory Pool Information**

This command shows all of the memory pool information.

```
dpdkinfo -m all
```

-----				
Name	Size	Used	Available	
-----				
rss_mempool		16384	1549	14835
frag_direct_mempool	4096	0	4096	
frag_indirect_mempool	4096	0	4096	
packet_mbuf_pool	8192	2	8190	

**dpdkinfo Statistics Information**

This command displays statistical information for a specific interface.

```
dpdkinfo -n 3
```

```
Interface Info(0000:18:0d.1):
RX Device Packets:6710, Bytes:1367533, Errors:0, Nombufs:0
Dropped RX Packets:0
```

```
TX Device Packets:0, Bytes:0, Errors:0
```

```
Queue Rx:
```

```
    Tx:
```

```
    Rx Bytes:
```

```
    Tx Bytes:
```

```
    Errors:
```

### **dpdkinfo Extended Statistics Information**

This command displays extended statistical information for a specific interface.

```
dpdkinfo -x 3
```

```
Driver Name:net_iavf
```

```
Interface Info:0000:18:0d.1
```

```
Rx Packets:
```

```
    rx_good_packets: 6701
```

```
    rx_unicast_packets: 0
```

```
    rx_multicast_packets: 2987
```

```
    rx_broadcast_packets: 3714
```

```
    rx_dropped_packets: 0
```

```
Tx Packets:
```

```
    tx_good_packets: 0
```

```
    tx_unicast_packets: 0
```

```
    tx_multicast_packets: 0
```

```
    tx_broadcast_packets: 0
```

```
    tx_dropped_packets: 0
```

```
Rx Bytes:
```

```
    rx_good_bytes: 1365696
```

```
Tx Bytes:
```

```
    tx_good_bytes: 0
```

```
Errors:
```

```
    rx_missed_errors: 0
```

```
    rx_errors: 0
```

```
    tx_errors: 0
```

```
    rx_mbuf_allocation_errors: 0
```

```
    inline_ipsec_crypto_ierrors: 0
```

```
    inline_ipsec_crypto_ierrors_sad_lookup: 0
```

```
    inline_ipsec_crypto_ierrors_not_processed: 0
```

```
    inline_ipsec_crypto_ierrors_icv_fail: 0
```

```
    inline_ipsec_crypto_ierrors_length: 0
```

Others:

inline\_ipsec\_crypto\_ipackets: 0

-----

## Troubleshooting vRouter

For vRouter-agent debugging, we use Introspect. You can access the Introspect data at **http://<host server IP>:8085**. [Table 7 on page 27](#) shows a sample of the Introspect data..

**NOTE:** The table, [Table 7 on page 27](#) shows grouped output. The cloud-native router does not group or sort the output on live systems.

The **http://host server IP address:8085** page displays only a list of HTML links.

**Table 7: Modules shown in contrail-vrouter-agent debug output**

Link	and Description
<a href="#">agent.xml</a>	Shows agent operational data. Using this introspect, you can see the list of interfaces, VMs, VNs, VRFs, security groups, ACLs and mirror configurations.
<a href="#">agent_ksync.xml</a>	Shows agent ksync layer for data objects such as interfaces and bridge ports.
<a href="#">agent_profile.xml</a>	shows agent <b>operdb</b> , tasks, flows, and statistics summary.
<a href="#">agent_stats_interval.xml</a>	View and set collection period for statistics.
<a href="#">controller.xml</a>	Shows the connection status of the jcnr-controller (cRPD)
<a href="#">cpuinfo.xml</a>	Shows the CPU load and memory usage on the compute node.

Table 7: Modules shown in contrail-vrouter-agent debug output (*Continued*)

Link	and Description
<b>ifmap_agent.xml</b>	Shows the current configuration data received from <b>ifmap</b> .
<b>kstate.xml</b>	Shows data configured in the vRouter data path.
<b>mac_learning.xml</b>	Shows entries in vRouter-agent MAC learning table.
<b>sandesh_trace.xml</b>	Gives the different agent module traces such as <b>oper</b> , <b>ksync</b> , <b>mac learning</b> , and <b>grpc</b> .
<b>sandesh_uve.xml</b>	Lists all the user visible entities (UVEs) in the vRouter-agent. The UVEs are used for analytics and telemetry.
<b>stats.xml</b>	Shows vRouter-agent slow path statistics such as error packets, trapped packets, and debug statistics.
<b>task.xml</b>	Shows vRouter-agent worker task details.



# JCNR-CNI

## IN THIS CHAPTER

- [Benefits of JCNR-CNI | 30](#)
- [JCNR-CNI Inside Cloud-Native Router | 30](#)
- [JCNR-CNI Role in pod Creation | 31](#)
- [Network Attachment Definitions | 32](#)

Read this chapter to learn about JCNR-CNI, which is the primary container network interface for JCNR.

The JCNR-CNI manages the secondary interfaces that the pods use. It creates the required interfaces based on the configuration in YAML-formatted network attachment definition (NAD) files. The JCNR-CNI configures some interfaces before passing them to their final location or connection point and provides an API for further interface configuration options.

- JCNR-CNI instantiates different kinds of pod interfaces.
- Creates virtio-based high performance interfaces for pods that leverage the DPDK data plane
- Creates veth pair interfaces that allow pods to communicate using the Linux Kernel networking stack
- Creates pod interfaces in access or trunk mode
- Attaches pod interfaces to bridge domains
- Supports IPAM plug-in for Dynamic IP address allocation
- Allocates unique socket interfaces for virtio interfaces
- Applies L2 access control lists (ACLs) to JCNR-vRouter
- Attaches pod interfaces to a bridge domain
- Manages the networking tasks in pods such as assigning IP addresses and setting up of interfaces between the pod and host in a Kubernetes cluster
- Applies Kubernetes network policies that are translated to firewall filter rules. The JCNR-CNI sends the firewall policies to JCNR-vRouter for application in the data plane.

- Connects pod interface to network: pod-to-pod and pod-to-network
- Integrates with JCNR-vRouter for offloading packet processing

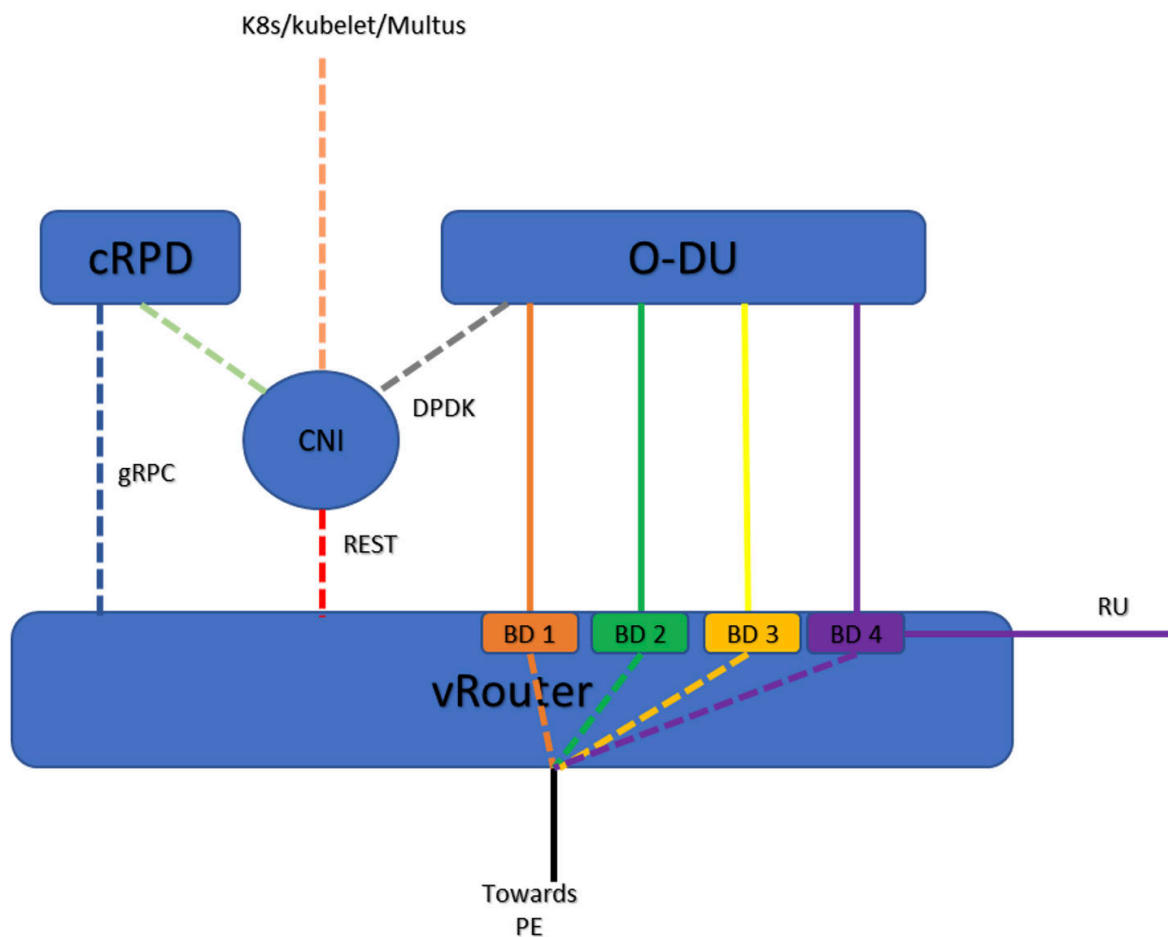
## Benefits of JCNR-CNI

- Improved pod interface management
- Customizable administrative and monitoring capabilities
- Improved application security
- Increased performance through tight integration with cRPD and vRouter components

## JCNR-CNI Inside Cloud-Native Router

JCNR-CNI is a specialized container network interface that can make a variety of network connections. It operates together with the Multus CNI. [Figure 3 on page 31](#) shows how JCNR-CNI interacts with the other components in Juniper Cloud-Native Router.

Figure 3: JCNr-CNI in an L2 Deployment



## JCNr-CNI Role in pod Creation

When you create a pod for use in the cloud-native router, the Kubernetes component known as **kubelet** calls the Multus CNI to set up pod networking and interfaces. Multus reads the annotations section of the **pod.yaml** file to find the NADs. If a NAD points to JCNr-CNI as the CNI plug in, Multus calls the JCNr-CNI to set up the pod interface. JCNr-CNI creates the interface as specified in the NAD. JCNr-CNI then generates and pushes a configuration into cRPD.

## Network Attachment Definitions

The NAD files are YAML files that the Multus CNI uses during the interface creation phase of pod creation. A NAD specifies the interface MAC addresses and allocates IP addresses. Each pod can use one or more NAD, typically one per pod interface. In the pod YAML file, the NAD to use for pod creation is listed under the network annotations section. In addition to creating the interface on pods, NADs can create virtual switches. The NAD attaches pod interfaces to L2 switching instances. The table, [Table 8 on page 32](#) describes the L2 interface types and modes supported.

**Table 8: NAD - L2 Interface Modes**

Interface Mode	Characteristics	Comments
Access	Allows untagged packets to traverse the link to the pod	Must be explicitly bound to a bridge domain
		Virtual switches use access mode for non-DPDK interfaces and applications like SSH and syslog
Trunk	Allows packets within specifically configured VLAN range	Implicitly part of one or more bridge domains
	No IP address allocation by CNI	Virtual switches in trunk mode carry DU user-plane traffic
	If IP address is needed, the pod must have its own allocation method such as DHCP	Dynamically add and remove network slices in 5G environments without restarting the Pod

# 2

PART

## Juniper Cloud-Native Router – Features

---

Cloud-Native Router Common Features | 34

Cloud-Native Router L2 Features | 45

Cloud-Native Router L3 Features | 65

---

# Cloud-Native Router Common Features

## SUMMARY

Read this chapter to learn about the common features of the Juniper Cloud-Native Router. We discuss cloud-native router interface types and other features that are present in both L2 and L3 deployment mode.

## IN THIS SECTION

- [Juniper Cloud-Native Router Interface Types | 34](#)
- [Logging and Notifications | 38](#)
- [Juniper Cloud-Native Router Licensing | 41](#)
- [Useful CLI Commands | 42](#)

## Juniper Cloud-Native Router Interface Types

Juniper Cloud-Native Router supports the following types of interfaces:

- **Agent interface**

vRouter has only one agent interface. The agent interface enables communication between the vRouter-agent and the vRouter. On the vRouter CLI when you issue the `vif --list` command, the agent interface looks like this:

```
vif0/0      Socket: unix
            Type: Agent HWaddr:00:00:5e:00:01:00
            Vrf:65535 Flags:L2 QOS:-1 Ref:3
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0
            RX packets:0 bytes:0 errors:0
            TX packets:650 bytes:99307 errors:0
            Drops:0
```

- **Data Plane Development Kit (DPDK) Virtual Function (VF) workload interfaces**

These interfaces connect to the radio units (RUs) or millimeter-wave distributed units (mmWave-DUs) On the vRouter CLI when you issue the `vif --list` command, the DPDK VF workload interface looks like this:

```
vif0/5      PCI: 0000:ca:19.1 (Speed 10000, Duplex 1)
            Type: Workload HWaddr: 9e:52:29:9e:97:9b
            Vrf: 0 Flags: L2Vof QOS: -1 Ref: 9
            RX queue packets: 29087 errors: 0
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0 0
            Fabric Interface: 0000:ca:19.1 Status: UP Driver: net_iavf
            Vlan Mode: Access Vlan Id: 1250 OVlan Id: 1250
            RX packets: 29082 bytes: 6766212 errors: 5
            TX packets: 0 bytes: 0 errors: 0
            Drops: 29896
```

- **DPDK VF fabric interfaces**

DPDK VF fabric interfaces, which are associated with the physical network interface card (NIC) on the host server, accept traffic from multiple VLANs. On the vRouter CLI when you issue the `vif --list` command, the DPDK VF fabric interface looks like this:

```
vif0/1      PCI: 0000:31:01.0 (Speed 10000, Duplex 1)
            Type: Physical HWaddr: d6:22:c5:42:de:c3
            Vrf: 65535 Flags: L2Vof QOS: -1 Ref: 12
            RX queue packets: 11813 errors: 1
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 1 0
            Fabric Interface: 0000:31:01.0 Status: UP Driver: net_iavf
            Vlan Mode: Trunk Vlan: 1001-1100
            RX packets: 0 bytes: 0 errors: 49962
            TX packets: 18188356 bytes: 2037400554 errors: 0
            Drops: 49963
```

- **Active or standby bond interfaces**

Bond interfaces accept traffic from multiple VLANs. A bond interface runs in the active or standby mode (mode 0).

On the vRouter CLI when you issue the `vif --list` command, the bond interface looks like this:

```
vif0/2    PCI: 0000:00:00.0 (Speed 10000, Duplex 1)
          Type:Physical HWaddr:32:f8:ad:8c:d3:bc
          Vrf:65535 Flags:L2Vof QOS:-1 Ref:8
          RX queue  packets:1882 errors:0
          RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0
          Fabric Interface: eth_bond_bond0 Status: UP Driver: net_bonding
          Slave Interface(0): 0000:81:01.0 Status: UP Driver: net_iavf
          Slave Interface(1): 0000:81:03.0 Status: UP Driver: net_iavf
          Vlan Mode: Trunk Vlan: 751-755
          RX packets:8108366000 bytes:486501960000 errors:4234
          TX packets:65083776 bytes:4949969408 errors:0
          Drops:8108370394
```

- **Pod interfaces using virtio and the DPDK data plane**

Virtio interfaces accept traffic from multiple VLANs and are associated with pod interfaces that use virtio on the DPDK data plane.

On the vRouter CLI when you issue the `vif --list` command, the virtio with DPDK data plane interface looks like this:

```
vif0/3    PMD: vhost242ip-93883f16-9ebb-4acf-b
          Type:Virtual HWaddr:00:16:3e:7e:84:a3
          Vrf:65535 Flags:L2 QOS:-1 Ref:13
          RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0
          Vlan Mode: Trunk Vlan: 1001-1003
          RX packets:0 bytes:0 errors:0
          TX packets:10604432 bytes:1314930908 errors:0
          Drops:0
          TX port packets:0 errors:10604432
```

- **Pod interfaces using virtual Ethernet (veth) pairs and the DPDK data plane**

Pod interfaces that use **veth** pairs and the DPDK data plane are access interfaces rather than trunk interfaces. This type of a pod interface allows traffic from only one VLAN to pass.



On the vRouter CLI when you issue the `vif --list` command, the veth pair with DPDK data plane interface looks like this:

```
vif0/4      Ethernet: jvknet1-88c44c3
            Type:Virtual HWaddr:02:00:00:3a:8f:73
            Vrf:0 Flags:L2Vof QOS:-1 Ref:10
            RX queue packets:524 errors:0
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0 0
            Vlan Mode: Access Vlan Id: 3001 OVlan Id: 3001
            RX packets:9 bytes:802 errors:515
            TX packets:0 bytes:0 errors:0
            Drops: 525
```

- **VLAN sub-interfaces**

Starting in Juniper Cloud-Native Router Release 22.4, the cloud-native router supports the use of VLAN sub-interfaces. VLAN sub-interfaces are like logical interfaces on a physical switch or router. When you run the cloud-native router in L2 mode, you must associate each sub-interface with a specific VLAN. On the JCNr-vRouter, a VLAN sub-interface look like this:

```
vif0/5      Virtual: vhostnet1-71cd7db1-1a5e-49.3003 Vlan(o/i)(,S): 3003/3003 Parent:vif0/4
            Type:Virtual(Vlan) HWaddr:00:99:99:99:33:09
            Vrf:0 Flags:L2 QOS:-1 Ref:3
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0
            RX packets:0 bytes:0 errors:0
            TX packets:0 bytes:0 errors:0
            Drops:0
```

- **Physical Function (PF) workload interfaces**
- **PF fabric interfaces**
- **The vhost0 interface**

The `vhost0` interface is an L3-only interface. When you run the cloud-native router in L3 mode, you must map the `vhost0` interface to a kernel-based physical interface such as `eth0`, `en1`, etc. You make the mapping by adjusting the value of the `vrouter_dpdk_physical_interface:` key in the file **Juniper\_Cloud\_Native\_Router\_version/helmchart/values\_L3.yaml** prior to deployment. In this configuration, the system uses the same physical interface for both IPv4 and IPv6 traffic.

Alternatively, you can choose specific interfaces for IPv4 and IPv6 traffic by entering the appropriate physical interface name in the `vhost_interface_ipv4:` and `vhost_interface_ipv6:` keys respectively.

**NOTE:** vRouter does not support the `vhost0` interface when run in L2 mode.

The vRouter-agent detects L2 mode in **values.yaml** during deployment, so does not wait for the `vhost0` interface to come up before completing installation. The vRouter-agent does not send a `vhost` interface add message so the vRouter doesn't create the `vhost0` interface.

Pods are the Kubernetes element that contains the interfaces used in cloud-native router. You control interface creation by manipulating the value portion of the *key:value* pairs in YAML configuration files. The cloud-native router uses a pod-specific file and a network attachment device (NAD)-specific file for pod and interface creation. During pod creation, Kubernetes consults the pod and NAD configuration files and creates the needed interfaces from the values contained within the NAD configuration file.

You can see example NAD and pod YAML files in the ["L2 - Add User Pod with Kernel Access to a Cloud-Native Router Instance" on page 73](#) and ["L2 - Add User Pod with virtio Trunk Ports to a Cloud-Native Router Instance" on page 83](#) examples.

## Logging and Notifications

### IN THIS SECTION

- [File Locations | 38](#)
- [Notifications | 39](#)

Read this topic to learn about logging and notification functions in Juniper Cloud-Native Router. We discuss the location of log files, what you can log, and various log levels. You can also learn about the available notifications and how the notifications are implemented in the cloud-native router.

### File Locations

The Juniper Cloud-Native Router pods and containers use syslog as their logging mechanism. You can determine the location of the log files at the deployment time by retaining or changing the value of the **log\_path** key in the **values.yaml** file. By default, the location of the log files is **/var/log/jcnr**. The system stores log files from all the cloud-native router pods and containers in the **log\_path** directory.

In addition, a syslog-ng pod stores event notification data in JSON format on the host server. The syslog-ng pod stores the JSON-formatted notifications in the directory specified by the **syslog\_notifications** key in the **values.yaml** file. By default, the file location is **/var/log/jcncr** and the filename is **jcncr\_notifications.json**. You can change the location and filename by changing the value of the **syslog\_notifications** key before the cloud-native router deployment.

When you use the default file locations, the **/var/log/jcncr** directory displays the following files:

```
[root@jcncr-1 /var/log/jcncr]# ls
contrail-vrouter-agent.log    contrail-vrouter-agent.log.2    contrail-vrouter-dpdk.log
jcncr_notifications.json
contrail-vrouter-agent.log.1  contrail-vrouter-dpdk-init.log  jcncr-cni.log
vrouter-kernel-init.log
```

**NOTE:** The host server must manage the log rotation for the **contrail-vrouter-dpdk.log** and the **jcncr-cni.log** files.

## Notifications

The syslog-ng pod continuously monitors the preceding log files for notification events such as interface up, interface down, interface add, and so on. When these events appear in a log file, syslog-ng converts the log events into notification events and stores the events in JSON format within the **syslog\_notifications** file configured in the **values.yaml** file.

As of Juniper Cloud-Native Router Release 22.2, syslog-ng stores the following notifications:

**Table 9: Supported Notifications**

Notification	Source Pod
License Near Expiry	cRPD
License Expired	cRPD
License Invalid	cRPD
License OK	cRPD

**Table 9: Supported Notifications** *(Continued)*

Notification	Source Pod
JCNR Init Success	Deployer
JCNR Init Failure	Deployer
Upstream Fabric Bond Member Link Up	vRouter
Upstream Fabric Bond Member Link Down	vRouter
Upstream Fabric Bond Link Up	vRouter
Upstream Fabric Bond Link Down	vRouter
Downstream Fabric Link Up	vRouter
Downstream Fabric Link Down	vRouter
Appliance Link Up	vRouter
Appliance Link Down	vRouter
Any JCNR Application Critical Errors	vRouter
JCNR MAC Table Limit Reached	vRouter
JCNR CLI Start	cRPD or vRouter-Agent
JCNR CLI Stop	cRPD or vRouter-Agent
JCNR Kernel App Interface Up	vRouter
JCNR Kernel App Interface Down	vRouter

Table 9: Supported Notifications *(Continued)*

Notification	Source Pod
JCNR Virtio User Interface Up	vRouter
JCNR Virtio User Interface Down	vRouter

SEE ALSO

No Link Title

## Juniper Cloud-Native Router Licensing

IN THIS SECTION

- [Licensing in the Juniper Cloud-Native Router | 41](#)

Read this section to learn about Juniper Cloud-Native Router licensing.

### Licensing in the Juniper Cloud-Native Router

Starting in Juniper Cloud-Native Router Release 22.2, we've enabled our Juniper Agile Licensing (JAL) model. JAL ensures that features are used in compliance with Juniper's end-user license agreement. You can purchase licenses for the Juniper Cloud-Native Router software through your Juniper Account Team. You can apply the licenses by using the CLI of the cloud-native router controller. For details about managing multiple license files for multiple cloud-native router deployments, see [Juniper Agile Licensing Overview](#)

If your cRPD pod displays its state as *running* when you issue the command `kubectl get pods -A` on the host server, then you have properly applied your license file.

**NOTE:** In Juniper Cloud-Native Router Releases 22.3 and 22.4, we only monitor license compliance. We do not enforce license compliance.

After configuring, apply your firewall filters to a bridge domain using a cRPD configuration command similar to: `set routing-instances vswitch bridge-domains bd3001 forwarding-options filter input filter1`. Then, commit the configuration for the firewall filter to take effect.

To see the how many packets matched the filter (per VLAN), you can use the cRPD CLI and issue the command:

```
show firewall filter filter1
```

The output from the above command looks like:

```
Filter : filter1    vlan-id : 3001
Term               Packet
t1                 0
```

In this example, we applied the filter to the bridge domain bd3001. The filter has not yet matched any packets.

## Useful CLI Commands

This section provides some example CLI commands and their outputs. We also provide some command completion example outputs. These outputs allow you to see the available command hierarchy which you can explore on your cloud-native router system.

You can see the bridge command hierarchy with the `show bridge ?` command shown as follows.

```
show bridge ?
Possible completions:
mac-table      Show media access control table
statistics     Show bridge statistics information
```

If you look further into the hierarchy, you see:

```
show bridge mac-table ?
Possible completions:
  <[Enter]>      Execute this command
  count          Number of MAC address
  mac-address     MAC address in the format XX:XX:XX:XX:XX:XX
  vlan-id         Display MAC address learned on a specified VLAN or 'all-vlan'
  |              Pipe through a command
```

If you use the <[Enter]> option, you see something like:

```
show bridge mac-table
Routing Instance : default-domain:default-project:ip-fabric:__default__
Bridging domain VLAN id : 3002
MAC              MAC              Logical
address          flags            interface

00:00:5E:00:53:01    D                bond0
```

The `show bridge mac-table` command displays the L2 MAC table which is dynamically learned by the vRouter.

If you look at the other option, statistics, you see:

```
show bridge statistics ?
Possible completions:
  <[Enter]>      Execute this command
  vlan-id         Display statistics for a particular vlan (1..4094)
  |              Pipe through a command
```

If you use the <[Enter]> option, you see:

```
show bridge statistics
Bridge domain vlan-id: 100
  Local interface: bond0
    Broadcast packets Tx : 0      Rx : 0
    Multicast packets Tx : 0      Rx : 0
    Unicast packets Tx   : 0      Rx : 0
    Broadcast bytes Tx   : 0      Rx : 0
```

```

Multicast bytes Tx   : 0           Rx : 0
Unicast bytes Tx    : 0           Rx : 0
Flooded packets     : 0
Flooded bytes       : 0
Local interface: ens1f0v1
Broadcast packets Tx : 0           Rx : 0
Multicast packets Tx : 0           Rx : 0
Unicast packets Tx   : 0           Rx : 0
Broadcast bytes Tx   : 0           Rx : 0
Multicast bytes Tx   : 0           Rx : 0
Unicast bytes Tx     : 0           Rx : 0
Flooded packets     : 0
Flooded bytes       : 0
Local interface: ens1f3v1
Broadcast packets Tx : 0           Rx : 0
Multicast packets Tx : 0           Rx : 0
Unicast packets Tx   : 0           Rx : 0
Broadcast bytes Tx   : 0           Rx : 0
Multicast bytes Tx   : 0           Rx : 0
Unicast bytes Tx     : 0           Rx : 0
Flooded packets     : 0

```

The `show bridge statistics` command displays the L2 VLAN traffic statistics per interface within a bridge domain.

To see the firewall (ACL) configuration:

```

show configuration firewall:firewall
family {
  bridge {
    filter filter1 {
      term t1 {
        from {
          destination-mac-address 10:30:30:30:30:31;
          source-mac-address 10:30:30:30:30:30;
          ether-type oam;
        }
        then {
          discard;
        }
      }
    }
  }
}

```



```
}
}
```

## Cloud-Native Router L2 Features

### SUMMARY

Read this chapter to learn about the features of the Juniper Cloud-Native Router running in L2 mode. We discuss L2 metrics and telemetry, L2 ACLs (firewall filters), MAC learning and aging, and L2 BUM traffic rate limiting.

### IN THIS SECTION

- [Juniper Cloud-Native Router Deployment Modes | 45](#)
- [Juniper Cloud-Native Router L2 Interface Types | 46](#)
- [L2 Metrics and Telemetry | 50](#)
- [L2 ACLs \(Firewall Filters\) | 56](#)
- [MAC Learning and Aging | 59](#)
- [BUM Rate Limiting | 61](#)
- [L2 API to Force Bond Link Switchover | 61](#)
- [L2 Quality of Service \(QoS\) | 62](#)

## Juniper Cloud-Native Router Deployment Modes

Starting in Juniper Cloud-Native Router Release 22.4, you can deploy and operate Juniper Cloud-Native Router in either L2 or L3 mode. You control the deployment mode by editing the appropriate `values.yaml` file prior to deployment.

To deploy the cloud-native router in L2 mode, retain or modify the values in the file **`Juniper_Cloud_Native_Router_version-number/helmchart/values.yaml`**.

Throughout the rest of this chapter we identify those features that are only available in L2 mode by beginning the feature name with L2.

In L2 mode, the cloud-native router behaves like a switch and so performs no routing functions and runs no routing protocols. The pod network uses VLANs to direct traffic to various destinations.

To deploy the cloud-native router in L3 mode, retain or modify the values in the file `Juniper_Cloud_Native_Router_version-number/helmchart/values_L3.yaml`,

In L3 mode, the cloud-native router behaves like a router and so performs routing functions and runs routing protocols such as ISIS, BGP, OSPF, and segment routing-MPLS. In L3 mode, the pod network is divided into an IPv6 underlay network and an IPv4 or IPv6 overlay network. The IPv6 underlay network is used for control plane traffic.

## Juniper Cloud-Native Router L2 Interface Types

Juniper Cloud-Native Router supports the following types of interfaces:

- **Agent interface**

vRouter has only one agent interface. The agent interface enables communication between the vRouter-agent and the vRouter. On the vRouter CLI when you issue the `vif --list` command, the agent interface looks like this:

```
vif0/0      Socket: unix
            Type: Agent HWaddr:00:00:5e:00:01:00
            Vrf:65535 Flags:L2 QOS:-1 Ref:3
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0
            RX packets:0 bytes:0 errors:0
            TX packets:650 bytes:99307 errors:0
            Drops:0
```

- **Data Plane Development Kit (DPDK) Virtual Function (VF) workload interfaces**

These interfaces connect to the radio units (RUs) or millimeter-wave distributed units (mmWave-DUs) On the vRouter CLI when you issue the `vif --list` command, the DPDK VF workload interface looks like this:

```
vif0/5      PCI: 0000:ca:19.1 (Speed 10000, Duplex 1)
            Type: Workload HWaddr:9e:52:29:9e:97:9b
            Vrf:0 Flags:L2Vof QOS:-1 Ref:9
            RX queue packets:29087 errors:0
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0
            Fabric Interface: 0000:ca:19.1 Status: UP Driver: net_iavf
```

```
Vlan Mode: Access Vlan Id: 1250 OVlan Id: 1250
RX packets:29082 bytes:6766212 errors:5
TX packets:0 bytes:0 errors:0
Drops:29896
```

- **DPDK VF fabric interfaces**

DPDK VF fabric interfaces, which are associated with the physical network interface card (NIC) on the host server, accept traffic from multiple VLANs. On the vRouter CLI when you issue the `vif --list` command, the DPDK VF fabric interface looks like this:

```
vif0/1    PCI: 0000:31:01.0 (Speed 10000, Duplex 1)
          Type:Physical HWaddr:d6:22:c5:42:de:c3
          Vrf:65535 Flags:L2Vof QOS:-1 Ref:12
          RX queue packets:11813 errors:1
          RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 1 0
          Fabric Interface: 0000:31:01.0 Status: UP Driver: net_iavf
          Vlan Mode: Trunk Vlan: 1001-1100
          RX packets:0 bytes:0 errors:49962
          TX packets:18188356 bytes:2037400554 errors:0
          Drops:49963
```

- **Active or standby bond interfaces**

Bond interfaces accept traffic from multiple VLANs. A bond interface runs in the active or standby mode (mode 0).

On the vRouter CLI when you issue the `vif --list` command, the bond interface looks like this:

```
vif0/2    PCI: 0000:00:00.0 (Speed 10000, Duplex 1)
          Type:Physical HWaddr:32:f8:ad:8c:d3:bc
          Vrf:65535 Flags:L2Vof QOS:-1 Ref:8
          RX queue packets:1882 errors:0
          RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0
          Fabric Interface: eth_bond_bond0 Status: UP Driver: net_bonding
          Slave Interface(0): 0000:81:01.0 Status: UP Driver: net_iavf
          Slave Interface(1): 0000:81:03.0 Status: UP Driver: net_iavf
          Vlan Mode: Trunk Vlan: 751-755
          RX packets:8108366000 bytes:486501960000 errors:4234
```

```
TX packets:65083776  bytes:4949969408  errors:0
Drops:8108370394
```

- **Pod interfaces using virtio and the DPDK data plane**

Virtio interfaces accept traffic from multiple VLANs and are associated with pod interfaces that use virtio on the DPDK data plane.

On the vRouter CLI when you issue the `vif --list` command, the virtio with DPDK data plane interface looks like this:

```
vif0/3    PMD: vhost242ip-93883f16-9ebb-4acf-b
          Type:Virtual HWaddr:00:16:3e:7e:84:a3
          Vrf:65535 Flags:L2 QOS:-1 Ref:13
          RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0 0 0
          Vlan Mode: Trunk Vlan: 1001-1003
          RX packets:0 bytes:0 errors:0
          TX packets:10604432 bytes:1314930908 errors:0
          Drops:0
          TX port packets:0 errors:10604432
```

- **Pod interfaces using virtual Ethernet (veth) pairs and the DPDK data plane**

Pod interfaces that use **veth** pairs and the DPDK data plane are access interfaces rather than trunk interfaces. This type of a pod interface allows traffic from only one VLAN to pass.

On the vRouter CLI when you issue the `vif --list` command, the veth pair with DPDK data plane interface looks like this:

```
vif0/4    Ethernet: jvknet1-88c44c3
          Type:Virtual HWaddr:02:00:00:3a:8f:73
          Vrf:0 Flags:L2Vof QOS:-1 Ref:10
          RX queue packets:524 errors:0
          RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0 0 0
          Vlan Mode: Access Vlan Id: 3001 OVlan Id: 3001
          RX packets:9 bytes:802 errors:515
          TX packets:0 bytes:0 errors:0
          Drops: 525
```

- **VLAN sub-interfaces**

Starting in Juniper Cloud-Native Router Release 22.4, the cloud-native router supports the use of VLAN sub-interfaces. VLAN sub-interfaces are like logical interfaces on a physical switch or router. When you run the cloud-native router in L2 mode, you must associate each sub-interface with a specific VLAN. On the JCNR-vRouter, a VLAN sub-interface look like this:

```
vif0/5      Virtual: vhostnet1-71cd7db1-1a5e-49.3003 Vlan(o/i)(,S): 3003/3003 Parent:vif0/4
Type:Virtual(Vlan) HWaddr:00:99:99:99:33:09
Vrf:0 Flags:L2 QoS:-1 Ref:3
RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0
RX packets:0 bytes:0 errors:0
TX packets:0 bytes:0 errors:0
Drops:0
```

- **Physical Function (PF) workload interfaces**
- **PF fabric interfaces**

**NOTE:** vRouter does not support the vhost0 interface when run in L2 mode.

The vRouter-agent detects L2 mode in **values.yaml** during deployment, so does not wait for the vhost0 interface to come up before completing installation. The vRouter-agent does not send a vhost interface add message so the vRouter doesn't create the vhost0 interface.

In L3 mode, the vhost0 interface is present and functional.

Pods are the Kubernetes element that contains the interfaces used in cloud-native router. You control interface creation by manipulating the value portion of the *key:value* pairs in YAML configuration files. The cloud-native router uses a pod-specific file and a network attachment device (NAD)-specific file for pod and interface creation. During pod creation, Kubernetes consults the pod and NAD configuration files and creates the needed interfaces from the values contained within the NAD configuration file.

You can see example NAD and pod YAML files in the ["L2 - Add User Pod with Kernel Access to a Cloud-Native Router Instance" on page 73](#) and ["L2 - Add User Pod with virtio Trunk Ports to a Cloud-Native Router Instance" on page 83](#) examples.

## L2 Metrics and Telemetry

### IN THIS SECTION

- [Viewing L2 Metrics | 50](#)

Read this topic to learn how to view Layer 2 (L2) metrics from an instance of Juniper Cloud-Native Router.

### Viewing L2 Metrics

Juniper Cloud-Native Router comes with telemetry capabilities that enable you to see performance metrics and telemetry data. The container **contrail-vrouter-telemetry-exporter** provides you this visibility. This container runs along side the other vRouter containers in the **contrail-vrouter-masters** pod.

The telemetry exporter periodically queries the Introspect agent on the vRouter-agent for statistics and reports metrics information in response to the Prometheus scrape requests. You can directly view the telemetry data by using the following URL: **`http://host server IP address:8070`**. The following table shows a sample output.

**NOTE:** We've grouped the output shown in the following table. The cloud-native router does not group or sort the output on live systems.

Table 10: Sample Telemetry Output

Group	Sample Output
Memory usage per vRouter	<pre> # TYPE virtual_router_system_memory_cached_bytes gauge # HELP virtual_router_system_memory_cached_bytes Virtual router system memory cached virtual_router_system_memory_cached_bytes{vrouter_name="jcnr.example.com"} 2635970448 # TYPE virtual_router_system_memory_buffers gauge # HELP virtual_router_system_memory_buffers Virtual router system memory buffer virtual_router_system_memory_buffers{vrouter_name="jcnr.example.com"} 32689 # TYPE virtual_router_system_memory_bytes gauge # HELP virtual_router_system_memory_bytes Virtual router total system memory virtual_router_system_memory_bytes{vrouter_name="jcnr.example.com"} 2635970448 # TYPE virtual_router_system_memory_free_bytes gauge # HELP virtual_router_system_memory_free_bytes Virtual router system memory free virtual_router_system_memory_free_bytes{vrouter_name="jcnr.example.com"} 2635969296 # TYPE virtual_router_system_memory_used_bytes gauge # HELP virtual_router_system_memory_used_bytes Virtual router system memory used virtual_router_system_memory_used_bytes{vrouter_name="jcnr.example.com"} 32689 # TYPE virtual_router_virtual_memory_kilobytes gauge # HELP virtual_router_virtual_memory_kilobytes Virtual router virtual memory virtual_router_virtual_memory_kilobytes{vrouter_name="jcnr.example.com"} 0 # TYPE virtual_router_resident_memory_kilobytes gauge # HELP virtual_router_resident_memory_kilobytes Virtual router resident memory virtual_router_resident_memory_kilobytes{vrouter_name="jcnr.example.com"} 32689 # TYPE virtual_router_peak_virtual_memory_bytes gauge # HELP virtual_router_peak_virtual_memory_bytes Virtual router peak virtual memory virtual_router_peak_virtual_memory_bytes{vrouter_name="jcnr.example.com"} 2894328001 </pre>

Table 10: Sample Telemetry Output (Continued)

Group	Sample Output
Packet count per interface	<pre> # TYPE virtual_router_phys_if_input_packets_total counter # HELP virtual_router_phys_if_input_packets_total Total packets received by physical interface virtual_router_phys_if_input_packets_total{vrouter_name="jcnr.example.com",interface_name="bond0"} 1483 # TYPE virtual_router_phys_if_output_packets_total counter # HELP virtual_router_phys_if_output_packets_total Total packets sent by physical interface virtual_router_phys_if_output_packets_total{vrouter_name="jcnr.example.com",interface_name="bond0"} 32969 # TYPE virtual_router_phys_if_input_bytes_total counter # HELP virtual_router_phys_if_input_bytes_total Total bytes received by physical interface virtual_router_phys_if_input_bytes_total{interface_name="bond0",vrouter_name="jcnr.example.com"} 125558 # TYPE virtual_router_phys_if_output_bytes_total counter # HELP virtual_router_phys_if_output_bytes_total Total bytes sent by physical interface virtual_router_phys_if_output_bytes_total{vrouter_name="jcnr.example.com",interface_name="bond0"} 4597076 virtual_router_phys_if_input_bytes_total{vrouter_name="jcnr.example.com",interface_name="bond0"} 228300499320 virtual_router_phys_if_output_bytes_total{interface_name="bond0",vrouter_name="jcnr.example.com"} 228297889634 virtual_router_phys_if_input_packets_total{interface_name="bond0",vrouter_name="jcnr.example.com"} 1585421179 virtual_router_phys_if_output_packets_total{vrouter_name="jcnr.example.com",interface_name="bond0"} 1585402623 virtual_router_phys_if_output_packets_total{interface_name="bond0",vrouter_name="jcnr.example.com"} 1585403344 </pre>



Table 10: Sample Telemetry Output *(Continued)*

Group	Sample Output
CPU usage per vRouter	<pre># TYPE virtual_router_cpu_1min_load_avg gauge # HELP virtual_router_cpu_1min_load_avg Virtual router CPU 1 minute load average virtual_router_cpu_1min_load_avg{vrouter_name="jcnr.example.com"} 0.11625 # TYPE virtual_router_cpu_5min_load_avg gauge # HELP virtual_router_cpu_5min_load_avg Virtual router CPU 5 minute load average virtual_router_cpu_5min_load_avg{vrouter_name="jcnr.example.com"} 0.109687 # TYPE virtual_router_cpu_15min_load_avg gauge # HELP virtual_router_cpu_15min_load_avg Virtual router CPU 15 minute load average virtual_router_cpu_15min_load_avg{vrouter_name="jcnr.example.com"} 0.110156</pre>
Drop packet count per vRouter	<pre># TYPE virtual_router_dropped_packets_total counter # HELP virtual_router_dropped_packets_total Total packets dropped virtual_router_dropped_packets_total{vrouter_name="jcnr.example.com"} 35850</pre>

Table 10: Sample Telemetry Output (Continued)

Group	Sample Output
Packet count per interface per VLAN	<pre> # TYPE virtual_router_interface_vlan_multicast_input_packets_total counter # HELP virtual_router_interface_vlan_multicast_input_packets_total Total number of multicast packets received on interface VLAN virtual_router_interface_vlan_multicast_input_packets_total{interface_id="1",vlan_id=" 100"} 0 # TYPE virtual_router_interface_vlan_broadcast_output_packets_total counter # HELP virtual_router_interface_vlan_broadcast_output_packets_total Total number of broadcast packets sent on interface VLAN virtual_router_interface_vlan_broadcast_output_packets_total{interface_id="1",vlan_id= "100"} 0 # TYPE virtual_router_interface_vlan_broadcast_input_packets_total counter # HELP virtual_router_interface_vlan_broadcast_input_packets_total Total number of broadcast packets received on interface VLAN virtual_router_interface_vlan_broadcast_input_packets_total{interface_id="1",vlan_id=" 100"} 0 # TYPE virtual_router_interface_vlan_multicast_output_packets_total counter # HELP virtual_router_interface_vlan_multicast_output_packets_total Total number of multicast packets sent on interface VLAN virtual_router_interface_vlan_multicast_output_packets_total{interface_id="1",vlan_id= "100"} 0 # TYPE virtual_router_interface_vlan_unicast_input_packets_total counter # HELP virtual_router_interface_vlan_unicast_input_packets_total Total number of unicast packets received on interface VLAN virtual_router_interface_vlan_unicast_input_packets_total{interface_id="1",vlan_id="10 0"} 0 # TYPE virtual_router_interface_vlan_flooded_output_bytes_total counter # HELP virtual_router_interface_vlan_flooded_output_bytes_total Total number of output bytes flooded to interface VLAN virtual_router_interface_vlan_flooded_output_bytes_total{interface_id="1",vlan_id="100 "} 0 # TYPE virtual_router_interface_vlan_multicast_output_bytes_total counter # HELP virtual_router_interface_vlan_multicast_output_bytes_total Total number of multicast bytes sent on interface VLAN virtual_router_interface_vlan_multicast_output_bytes_total{interface_id="1",vlan_id="1 00"} 0 # TYPE virtual_router_interface_vlan_unicast_output_packets_total counter # HELP virtual_router_interface_vlan_unicast_output_packets_total Total number of unicast packets sent on interface VLAN virtual_router_interface_vlan_unicast_output_packets_total{interface_id="1",vlan_id="1 00"} 0 </pre>

Table 10: Sample Telemetry Output (*Continued*)

Group	Sample Output
	<pre> # TYPE virtual_router_interface_vlan_broadcast_input_bytes_total counter # HELP virtual_router_interface_vlan_broadcast_input_bytes_total Total number of broadcast bytes received on interface VLAN virtual_router_interface_vlan_broadcast_input_bytes_total{interface_id="1",vlan_id="10 0"} 0 # TYPE virtual_router_interface_vlan_multicast_input_bytes_total counter # HELP virtual_router_interface_vlan_multicast_input_bytes_total Total number of multicast bytes received on interface VLAN virtual_router_interface_vlan_multicast_input_bytes_total{vlan_id="100",interface_id=" 1"} 0 # TYPE virtual_router_interface_vlan_unicast_input_bytes_total counter # HELP virtual_router_interface_vlan_unicast_input_bytes_total Total number of unicast bytes received on interface VLAN virtual_router_interface_vlan_unicast_input_bytes_total{interface_id="1",vlan_id="100" } 0 # TYPE virtual_router_interface_vlan_flooded_output_packets_total counter # HELP virtual_router_interface_vlan_flooded_output_packets_total Total number of output packets flooded to interface VLAN virtual_router_interface_vlan_flooded_output_packets_total{interface_id="1",vlan_id="1 00"} 0 # TYPE virtual_router_interface_vlan_broadcast_output_bytes_total counter # HELP virtual_router_interface_vlan_broadcast_output_bytes_total Total number of broadcast bytes sent on interface VLAN virtual_router_interface_vlan_broadcast_output_bytes_total{interface_id="1",vlan_id="1 00"} 0 # TYPE virtual_router_interface_vlan_unicast_output_bytes_total counter # HELP virtual_router_interface_vlan_unicast_output_bytes_total Total number of unicast bytes sent on interface VLAN virtual_router_interface_vlan_unicast_output_bytes_total{interface_id="1",vlan_id="100 "} 0 ... </pre>

Prometheus is an open-source systems monitoring and alerting toolkit. You can use Prometheus to retrieve telemetry data from the cloud-native router host servers and view that data in the HTTP format. A sample of Prometheus configuration looks like this:

```

- job_name: "prometheus-JCNR-1a2b3c"

# metrics_path defaults to '/metrics'
# scheme defaults to 'http'.

```

```
static_configs:
- targets: ["<host-server-IP>:8070"]
```

## SEE ALSO

No Link Title

## L2 ACLs (Firewall Filters)

### IN THIS SECTION

- [L2 Firewall Filters | 56](#)
- [Firewall Filter Example | 57](#)
- [L2 Firewall Filter \(ACL\) Troubleshooting | 58](#)

Read this topic to learn about Layer 2 access control lists (L2 ACLs) in the cloud-native router.

## L2 Firewall Filters

Starting with Juniper Cloud-Native Router Release 22.2 we've included a limited firewall filter capability. You can configure the filters using the Junos OS CLI within the cloud-native router controller, using NETCONF, or the cloud-native router APIs.

During deployment, the system defines and applies firewall filters to block traffic from passing directly between the router interfaces. You can dynamically define and apply more filters. Use the firewall filters to:

- Define firewall filters for bridge family traffic.
- Define filters based on one or more of the following fields: source MAC address, destination MAC address, or EtherType.
- Define multiple terms within each filter.
- Discard the traffic that matches the filter.
- Apply filters to bridge domains.

## Firewall Filter Example

Below you can see an example of a firewall filter configuration from a cloud-native router deployment.

```
root@jcnr01> show configuration firewall:firewall
family {
  bridge {
    filter example {
      term t1 {
        from {
          destination-mac-address 10:10:10:10:10:11;
          source-mac-address 10:10:10:10:10:10;
          ether-type arp;
        }
        then {
          discard;
        }
      }
    }
  }
}
```

**NOTE:** You can configure up to 16 terms in a single firewall filter.

The only *then* action you can configure in a firewall filter is the discard action.

After configuration, you must apply your firewall filters to a bridge domain using a cRPD configuration command similar to: `set routing-instances vswitch bridge-domains bd3001 forwarding-options filter input filter1`. Then you must commit the configuration for the firewall filter to take effect.

To see how many packets matched the filter (per VLAN), you can issue the following command on the cRPD CLI:

```
show firewall filter filter1
```

The command output looks like this:

```
Filter : filter1    vlan-id : 3001
```

Term	Packet
t1	0

In the preceding example, we applied the filter to the bridge domain bd3001. The filter has not yet matched any packets.

## L2 Firewall Filter (ACL) Troubleshooting

The following table lists some of the potential problems that you might face when you implement firewall rules or ACLs in the cloud-native router. You run most of these commands on the host server. The "Command" column indicates whether the command shown needs to run somewhere else.

**Table 11: L2 Firewall Filter or ACL Troubleshooting**

Problem	Possible Causes and Resolution	Command
Firewall filters or ACLs not working	gRPC connection (port 50052) to the vRouter is down.  Check the gRPC connection.	<code>netstat -antp grep 50052</code>
	The ui-pubd process is not running.  Check whether ui-pubd is running.	<code>ps aux grep ui-pubd</code>
Firewall filter or ACL show commands not working	The gRPC connection (port 50052) to the vRouter is down.  Check the gRPC connection.	<code>netstat -antp grep 50052</code>
	The firewall service is not running.	<code>ps aux grep firewall</code>
		<code>show log filter.log</code>  You must run this command in the JCNR-controller (cRPD) CLI.

SEE ALSO

No Link Title
No Link Title
No Link Title

MAC Learning and Aging

IN THIS SECTION

- MAC Learning | 59
- MAC Entry Aging | 60

Juniper Cloud-Native Router provides automated learning and aging of MAC addresses. Read this topic for an overview of the MAC learning and aging functionality in the cloud-native router.

MAC Learning

MAC learning enables the cloud-native router to efficiently send the received packets to their respective destinations. The cloud-native router maintains a table of MAC addresses grouped by interface. The table includes MAC addresses, VLANs, and the interface on which the vRouter learns each MAC address and VLAN. The MAC table informs the vRouter about the MAC addresses that each interface can reach.

The cloud-native router caches the source MAC address for a new packet flow to record the incoming interface into the MAC table. The router learns the MAC addresses for each VLAN or bridge domain. The cloud-native router creates a key in the MAC table from the MAC address and VLAN of the packet. Queries sent to the MAC table return the interface associated with the key. To enable MAC learning, the cloud-native router performs these steps:

- Records the incoming interface into the MAC table by caching the source MAC address for a new packet flow.
- Learns the MAC addresses for each VLAN or bridge domain.
- Creates a key in the MAC table from the MAC address and VLAN of the packet.

If the destination MAC address and VLAN are missing (lookup failure), the cloud-native router floods the packet out all the interfaces (except the incoming interface) in the bridge domain.

By default:

- MAC table entries time out after 60 seconds.
- The MAC table size is limited to 10,240 entries.

You can configure the aging timeout and MAC table size during deployment by editing the **values.yaml** file under the **jcnr-vrouter** directory on the host server. We recommend that you do not change the default values.

You can see the MAC table entries by using:

- Introspect agent at **[http://host\\_server\\_IP:8085/mac\\_learning.xml#Snh\\_FetchL2MacEntry](http://host_server_IP:8085/mac_learning.xml#Snh_FetchL2MacEntry)**.
- The command **show bridge mac-table** on the cRPD CLI.
- The command **purel2cli --mac show** on the CLI of the **contrail-tools** pod.

If you exceed the MAC address limit, the counter **pkt\_drop\_due\_to\_mactable\_limit** increments. You can see this counter by using the introspect agent at **[http://host\\_server\\_IP:8085/Snh\\_AgentStatsReq](http://host_server_IP:8085/Snh_AgentStatsReq)**.

If you delete or disable an interface, the cloud-native router deletes all the MAC entries associated with that interface from the MAC table.

## MAC Entry Aging

The aging timeout for cached MAC entries is 60 seconds. You can configure the aging timeout at deployment time by editing the **values.yaml** file. The minimum timeout is 60 seconds and the maximum timeout is 10,240 seconds. You can see the time that is left for each MAC entry through introspect at **[http://host\\_server\\_IP:8085/mac\\_learning.xml#Snh\\_FetchL2MacEntry](http://host_server_IP:8085/mac_learning.xml#Snh_FetchL2MacEntry)**. We show an example of the output below:

```
l2_mac_entry_list
vrf_id      vlan_id      mac              index      packets
time_since_add  last_stats_change
0           1001         00:10:94:00:00:01  5644      615123154
12:55:14.248785  00:00:00.155450
0           1001         00:10:94:00:00:65  6480      615108294
12:55:14.247765  00:00:00.155461
0           1002         01:10:94:00:00:02  5628      615123173
12:55:14.248295  00:00:00.155470
```



## BUM Rate Limiting

The rate limiting feature controls the rate of egress broadcast, unknown unicast, and multicast (BUM) traffic on fabric interfaces. You specify the rate limit in bytes per second by adjusting **stormControlProfiles** in the **values.yaml** file before deployment. The system applies the configured profiles to all specified fabric interfaces in the cloud-native router. The maximum per-interface rate limit value you can set is 1,000,000 bytes per second.

If the unknown unicast, broadcast, or multicast traffic rate exceeds the set limit on a specified fabric interface, the vRouter drops the traffic. You can see the drop counter values by running the `dropstats` command in the vRouter CLI. You can see the per-interface rate limit drop counters by running the vRouter CLI command `vif --get fabric_vif_id --get-drop-stats` For example:

```
dropstats
L2 untag pkt drop      8832
L2 Src Mac lookup fail  880
Rate limit exceeded 29312474
```

When you configure a rate limit profile on a fabric interface, you can see the configured limit in bytes per second when you run either `vif --list` or `vif --get fabric_vif_id`.

## L2 API to Force Bond Link Switchover

When you run cloud-native router in L2 mode with cascaded nodes you can configure those nodes to use bond interfaces. If you also configure the bond interfaces as `BONDING_MODE_ACTIVE_BACKUP`, the vRouter-agent exposes the REST API call: `curl -X POST http://127.0.0.1:9091/bond-switch/bond0` on localhost port 9091. You can use this REST API call to force traffic to switch from the active interface to the standby interface.

The vRouter contains two CLI commands that allow you to see the active interface in a bonded pair and to see the traffic statistics associated with your bond interfaces. These commands are: `dpdinfo -b` and `dpdinfo -n` respectively.

## L2 Quality of Service (QoS)

### IN THIS SECTION

- [QoS Overview | 62](#)
- [QoS Example Configuration | 64](#)
- [Viewing the QoS Configuration | 64](#)

Starting in Juniper Cloud-Native Router Release 22.4, you can configure quality of service (QoS) parameters including classification, marking, and queuing. The cloud-native router performs classification and marking operations in vRouter and queuing (scheduling) operations in the physical network interface card (NIC). Scheduling is only supported on the E810 NIC.

### QoS Overview

You enable QoS prior to the deploy time by editing the `values.yaml` file in **Juniper-Cloud-Native-Router-*version-number*/helmchart** directory and changing the `qosEnable` value to `true`. The default value for the QoS feature is `false` (disabled).

**NOTE:** You can only enable the QoS feature if the host server on which you install your cloud-native router contains an Intel E810 NIC that is running lldp.

You enable lldp on the NIC using the `lldptool` which runs on the host server as a CLI application. Issue the following command to enable lldp on the E810 NIC. For example, you could use the following command:

```
lldptool -T -i INTERFACE -V ETS-CFG willing=no  
tsa=0:strict,1:strict,2:strict,3:strict,4:strict,  
5:strict,6:strict,7:strict  
up2tc=0:0,1:1,2:2,3:3,4:0,5:1,6:2,7:3
```

The details of the above command are:

- **ETS**—Enhanced Transmission Selection

- **willing**–The willing attribute determines whether the system uses locally configured packet forwarding classification (PFC) or not. If you set `willing` to `no`(the default setting), the cloud-native router applies local PFC configuration. If you set `willing` to `yes`, and the cloud-native router receives TLV from the peer router, the cloud-native router applies the received values.
- **tsa**–The transmission selection algorithm is a comma separated list of traffic class to selection algorithm maps. You can choose `ets`, `strict`, or `vendor` as selection algorithms.
- **up2tc**–Comma-separated list that maps user priorities to traffic classes

The list below provides an overview of the classification, marking, and queueing operations performed by cloud-native router.

- Classification:
  - vRouter classifies packets by examining the priority bits in the packet
  - vRouter derives traffic class and loss priority
  - vRouter can apply traffic classifiers to fabric, traffic, and workload interface types
  - vRouter maintains 16 entries in its classifier map
- Marking (Re-write):
  - vRouter performs marking operationsMarking is done in Vrouter. •Re-write of p-bits done in egress path. •At egress based on traffic class and drop priority new priority is derived. •Marking can be applied to Fabric interface only.
  - vRouter performs rewriting of p-bits in the egress path
  - vRouter derives new traffic priority based on traffic class and drop priority at egress
  - vRouter can apply marking to packets only on fabric interfaces
  - vRouter maintains 8 entries in its marking map
- Queueing (Scheduling):
  - Cloud-native router performs strict priority scheduling in hardware (E810 NIC)
  - Cloud-native router maps each traffic class to one queue
  - Cloud-native router limits the maximum number of traffic queue to 4
  - Cloud-native router maps 8 possible priorities to 4 traffic classes; It also maps each traffic class 1 hardware queue
  - Cloud-native router can apply scheduling to fabric interface only

- Virtual functions (VFs) leverage the queues that you configure in the physical functions (interfaces)
- vRouter maintains 8 entries in its scheduler map

## QoS Example Configuration

You configure QoS classifiers, rewrite rules, and schedulers in the cRPD using Junos set commands or remotely using NETCONF. We display a Junos-based example configuration below.

```
set class-of-service classifiers ieee-802.1 class1
  forwarding-class assured-forwarding loss-priority
  high code-points 011
set class-of-service rewrite-rules ieee-802.1 Rule_1
  forwarding-class assured-forwarding loss-priority
  high code-point 110
set class-of-service schedulers sch1 priority high
set class-of-service scheduler-maps sch1 forwarding-class
  assured-forwarding scheduler sch1
set class-of-service interfaces enp175s1 scheduler-map sch1
set class-of-service interfaces enp175s1 unit 0 rewrite-rules ieee-802.1 Rule_1
set class-of-service interfaces vhostnet123-3546aefd-7af8-4fe5 unit 0 classifiers ieee-802.1
class1
```

## Viewing the QoS Configuration

You view the QoS configuration in the cRPD CLI using show commands in Junos operation mode. The show commands reveal the configuration of classifiers, rewrite rules, or scheduler maps individually. We display three examples below; one example for each operation.

- **Show Classifier**

```
user@jcnr1> show class-of-service classifier

Classifier: class1, Code point type: ieee802.1p
Code point      Forwarding class      Loss priority
011             assured-forwarding    high
```

- **Show Rewrite-Rule**

```
user@jcnr1> show class-of-service rewrite-rule

Rewrite rule: Rule_1, Code point type: ieee802.1p
Forwarding class      Loss priority      Code point
assured-forwarding    high              110
```

- **Show Scheduler-Map**

```
show class-of-service scheduler-map sch1
Scheduler map: sch1
  Scheduler: sch1, Forwarding class: assured-forwarding
    Transmit rate: unspecified, Rate Limit: none, Priority: high
```

## Cloud-Native Router L3 Features

### SUMMARY

Read this chapter to learn about operation, and monitoring of the Juniper Cloud-Native Router running in L3 mode. We discuss cloud-native router deployment modes, interface types, and segment routing MPLS tunnels.

### IN THIS SECTION

- [Juniper Cloud-Native Router Deployment Modes | 66](#)
- [Juniper Cloud-Native Router Security Groups | 66](#)
- [Juniper Cloud-Native Router Interface Types | 67](#)
- [Security Groups | 70](#)
- [L2 API to Force Bond Link Switchover | 71](#)
- [MPLS Support in Juniper Cloud-Native Router | 71](#)

## Juniper Cloud-Native Router Deployment Modes

Starting in Juniper Cloud-Native Router Release 22.4, you can deploy and operate Juniper Cloud-Native Router in either L2 or L3 mode. You control the deployment mode by editing the appropriate `values.yaml` file prior to deployment.

To deploy the cloud-native router in L2 mode, retain or modify the values in the file `Juniper_Cloud_Native_Router_version-number/helmchart/values.yaml`.

Throughout the rest of this chapter we identify those features that are only available in L2 mode by beginning the feature name with L2.

In L2 mode, the cloud-native router behaves like a switch and so performs no routing functions and runs no routing protocols. The pod network uses VLANs to direct traffic to various destinations.

To deploy the cloud-native router in L3 mode, retain or modify the values in the file `Juniper_Cloud_Native_Router_version-number/helmchart/values_L3.yaml`,

In L3 mode, the cloud-native router behaves like a router and so performs routing functions and runs routing protocols such as ISIS, BGP, OSPF, and segment routing-MPLS. In L3 mode, the pod network is divided into an IPv6 underlay network and an IPv4 or IPv6 overlay network. The IPv6 underlay network is used for control plane traffic.

## Juniper Cloud-Native Router Security Groups

Starting in Juniper Cloud-Native Router Release 22.4, you control the types of traffic with the use of security groups when you use cloud-native router in L3 mode.

A security group is a container for security group rules. Security groups and security group rules allow administrators to specify the type of traffic that passes through an interface port. When you create a pod in a virtual network (VN), you can associate a security group with the pod and its virtual machine interface (VMI). The VMI is the interface connecting the pod to the vrouter-dpdk. When the cloud-native router launches the pod, it applies the rules in the security group to the pod's VMI port. If you do not specify a security group for the pod, the cloud-native router associates a default security group with the pod's VMI. The default security group rule is to allow all traffic to and from the port. The default security group allows both ingress and egress traffic. Security rules can be added to the default security group to change the traffic behavior.

You can apply each rule in the security group to either ingress or egress traffic. Ingress traffic is the traffic coming to the pod's VMI. Egress traffic is the traffic that leaves the pod through the VMI.

## Juniper Cloud-Native Router Interface Types

Juniper Cloud-Native Router supports the following types of interfaces:

- **Agent interface**

vRouter has only one agent interface. The agent interface enables communication between the vRouter-agent and the vRouter. On the vRouter CLI when you issue the `vif --list` command, the agent interface looks like this:

```
vif0/0      Socket: unix
            Type: Agent HWaddr:00:00:5e:00:01:00
            Vrf:65535 Flags:L2 QOS:-1 Ref:3
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0
            RX packets:0 bytes:0 errors:0
            TX packets:650 bytes:99307 errors:0
            Drops:0
```

- **Data Plane Development Kit (DPDK) Virtual Function (VF) workload interfaces**

These interfaces connect to the radio units (RUs) or millimeter-wave distributed units (mmWave-DUs) On the vRouter CLI when you issue the `vif --list` command, the DPDK VF workload interface looks like this:

```
vif0/5      PCI: 0000:ca:19.1 (Speed 10000, Duplex 1)
            Type: Workload HWaddr:9e:52:29:9e:97:9b
            Vrf:0 Flags:L2Vof QOS:-1 Ref:9
            RX queue packets:29087 errors:0
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0
            Fabric Interface: 0000:ca:19.1 Status: UP Driver: net_iavf
            Vlan Mode: Access Vlan Id: 1250 OVlan Id: 1250
            RX packets:29082 bytes:6766212 errors:5
            TX packets:0 bytes:0 errors:0
            Drops:29896
```

- **DPDK VF fabric interfaces**

DPDK VF fabric interfaces, which are associated with the physical network interface card (NIC) on the host server, accept traffic from multiple VLANs. On the vRouter CLI when you issue the `vif --list` command, the DPDK VF fabric interface looks like this:

```
vif0/1    PCI: 0000:31:01.0 (Speed 10000, Duplex 1)
          Type:Physical HWaddr:d6:22:c5:42:de:c3
          Vrf:65535 Flags:L2Vof QOS:-1 Ref:12
          RX queue packets:11813 errors:1
          RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0 1 0
          Fabric Interface: 0000:31:01.0 Status: UP Driver: net_iavf
          Vlan Mode: Trunk Vlan: 1001-1100
          RX packets:0 bytes:0 errors:49962
          TX packets:18188356 bytes:2037400554 errors:0
          Drops:49963
```

- **Active or standby bond interfaces**

Bond interfaces accept traffic from multiple VLANs. A bond interface runs in the active or standby mode (mode 0).

On the vRouter CLI when you issue the `vif --list` command, the bond interface looks like this:

```
vif0/2    PCI: 0000:00:00.0 (Speed 10000, Duplex 1)
          Type:Physical HWaddr:32:f8:ad:8c:d3:bc
          Vrf:65535 Flags:L2Vof QOS:-1 Ref:8
          RX queue packets:1882 errors:0
          RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0 0 0
          Fabric Interface: eth_bond_bond0 Status: UP Driver: net_bonding
          Slave Interface(0): 0000:81:01.0 Status: UP Driver: net_iavf
          Slave Interface(1): 0000:81:03.0 Status: UP Driver: net_iavf
          Vlan Mode: Trunk Vlan: 751-755
          RX packets:8108366000 bytes:486501960000 errors:4234
          TX packets:65083776 bytes:4949969408 errors:0
          Drops:8108370394
```

- **Pod interfaces using virtio and the DPDK data plane**

Virtio interfaces accept traffic from multiple VLANs and are associated with pod interfaces that use virtio on the DPDK data plane.



On the vRouter CLI when you issue the `vif --list` command, the virtio with DPDK data plane interface looks like this:

```
vif0/3    PMD: vhost242ip-93883f16-9ebb-4acf-b
          Type:Virtual HWaddr:00:16:3e:7e:84:a3
          Vrf:65535 Flags:L2 QOS:-1 Ref:13
          RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0
          Vlan Mode: Trunk Vlan: 1001-1003
          RX packets:0 bytes:0 errors:0
          TX packets:10604432 bytes:1314930908 errors:0
          Drops:0
          TX port packets:0 errors:10604432
```

- **Pod interfaces using virtual Ethernet (veth) pairs and the DPDK data plane**

Pod interfaces that use **veth** pairs and the DPDK data plane are access interfaces rather than trunk interfaces. This type of a pod interface allows traffic from only one VLAN to pass.

On the vRouter CLI when you issue the `vif --list` command, the veth pair with DPDK data plane interface looks like this:

```
vif0/4    Ethernet: jvknet1-88c44c3
          Type:Virtual HWaddr:02:00:00:3a:8f:73
          Vrf:0 Flags:L2Vof QOS:-1 Ref:10
          RX queue packets:524 errors:0
          RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0
          Vlan Mode: Access Vlan Id: 3001 0Vlan Id: 3001
          RX packets:9 bytes:802 errors:515
          TX packets:0 bytes:0 errors:0
          Drops: 525
```

- **VLAN sub-interfaces**

Starting in Juniper Cloud-Native Router Release 22.4, the cloud-native router supports . VLAN sub-interfaces are like logical interfaces on a physical switch or router. When we run the cloud-native

router in L2 mode, each sub-interface must be associated with a specific VLAN. On the JCNR-vRouter, a VLAN sub-interface look like this:

```
vif0/5      Virtual: vhostnet1-71cd7db1-1a5e-49.3003 Vlan(o/i)(,S): 3003/3003 Parent:vif0/4
Type:Virtual(Vlan) HWaddr:00:99:99:99:33:09
Vrf:0 Flags:L2 QOS:-1 Ref:3
RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0
RX packets:0 bytes:0 errors:0
TX packets:0 bytes:0 errors:0
Drops:0
```

- Physical Function (PF) workload interfaces
- PF fabric interfaces

**NOTE:** vRouter does not support the vhost0 interface when run in L2 mode.

The vRouter-agent detects L2 mode in **values.yaml** during deployment, so does not wait for the vhost0 interface to come up before completing installation. The vRouter-agent does not send a vhost interface add message so the vRouter doesn't create the vhost0 interface.

In L3 mode, the vhost0 interface is present and functional.

Pods are the Kubernetes element that contains the interfaces used in cloud-native router. You control interface creation by manipulating the value portion of the *key:value* pairs in YAML configuration files. The cloud-native router uses a pod-specific file and a network attachment device (NAD)-specific file for pod and interface creation. During pod creation, Kubernetes consults the pod and NAD configuration files and creates the needed interfaces from the values contained within the NAD configuration file.

You can see example NAD and pod YAML files in the ["L2 - Add User Pod with Kernel Access to a Cloud-Native Router Instance" on page 73](#) and ["L2 - Add User Pod with virtio Trunk Ports to a Cloud-Native Router Instance" on page 83](#) examples.

## Security Groups

A security group is a construct for holding security rules. When you create a pod in a virtual network, the cloud-native router associates a security group with the Virtual Management Interface (VMI). The VMI is the interface connecting the Pod and the vRouter container. Each rule in the security group is applied to either ingress or egress traffic. Ingress traffic is the traffic coming from the Pod over the VMI. Egress traffic is the traffic going from the VMI to the Pod.

With the Cloud-Native Router, you configure networking policy, including security groups, locally using gRPC messages from the cloud-native router controller. You can configure security groups using API calls, NETCONF, or the cloud-native router controller CLI by using the **edit routing-options flow security-group *security group name* rule *rule name*** command hierarchy.

## L2 API to Force Bond Link Switchover

When you use bond interfaces on cascaded nodes in L2 mode, you can make an API call to force traffic to switch from the active interface to the standby interface.

## MPLS Support in Juniper Cloud-Native Router

The Juniper Cloud-Native Router contains support for MPLS routing protocols. You use the JCNR-controller, or cRPD, to configure MPLS. The cRPD then sends the configuration to the vRouter-agent, using gRPC. The vRouter-agent then converts the configuration to network policies that it implements in the vRouter. The cloud-native router supports the following MPLS-based routing protocols:

- **L3 MPLS VPN (MPLS)**—L3 MPLS VPNs are also known as BGP/MPLS VPNs because BGP is used to distribute VPN routing information across the provider's backbone, and MPLS is used to forward VPN traffic across the backbone to remote VPN sites. The cloud-native router can participate as a sending, receiving, or transit router using the MPLS protocol
- **Segment Routing-MPLS (SR-MPLS)**—Segment routing is a control-plane architecture that enables an ingress router to steer a packet through a specific set of nodes and links in the network without relying on the intermediate nodes in the network to determine the actual path it should take. SR-MPLS employs segment routing in MPLS. The cloud-native router can participate as a sending, receiving or transit router in SR-MPLS networks.
- **MPLS over UDP (MPLSoUDP)**—MPLSoUDP is an overlay technology that encapsulates MPLS packets within UDP packets to traverse through some networks that do not support native MPLS or SR-MPLS. The cloud-native router can participate as a sending, receiving, or transit router using MPLSoUDP.

# 3

PART

## Juniper Cloud-Native Router (JCNR) - Examples

---

L2 - Add User Pod with Kernel Access to a Cloud-Native Router Instance | 73

L2 - Add User Pod with virtio Trunk Ports to a Cloud-Native Router Instance |  
83

L3 - Add User Pod to a Cloud-Native Router Instance | 93

---

## L2 - Add User Pod with Kernel Access to a Cloud-Native Router Instance

### SUMMARY

Read this topic to learn how to add a user pod with a kernel access interface to an instance of the cloud-native router.

### IN THIS SECTION

- [Overview | 73](#)
- [Before You Begin | 74](#)
- [Detailed Steps | 75](#)

## Overview

### IN THIS SECTION

- [High-Level Steps | 73](#)

To add a user pod to the cloud-native router, your high-level tasks are:

- Ensure that a network attachment definition (NAD) exists.
- Apply a pod YAML file to your cloud-native router cluster.

Throughout this example, we use the `kubectl` command with various options. You must run this command on the host-server CLI.

### High-Level Steps

In this example, we assume that this is the first user pod that you are adding to your newly installed cloud-native router. Therefore, we provide the steps to create a NAD on the cluster and then add the new user pod.

Below we provide a list of the individual steps we perform in this example. Each step in the list is a link to the detailed description of the step.

1. [View the vRouter interface list](#)
2. [Examine the example NAD YAML file](#)
3. [Apply the NAD to the cluster](#)
4. [Verify the NAD](#)
5. [Examine the example Pod YAML file](#)
6. [Apply the Pod to the cluster](#)
7. [Verify the Pod](#)
8. [View the updated vRouter interface list](#)

## Before You Begin

### IN THIS SECTION

- [Access the vRouter-Agent CLI | 74](#)

## Access the vRouter-Agent CLI

You perform the first and last steps of this example procedure on the CLI of the vRouter agent. We recommend that you open two SSH (terminal) sessions to the host server. You can use one session to run the CLI commands on the vRouter agent and the other session to run the `kubectl` commands that deploy the NAD and the pod on the cluster.

**NOTE:** To make it easy to copy and paste commands from here to your system, we do not include paths or shell prompts from the host server in the command listings.

Access the CLI of the `contrail-vrouter-agent` container in the `contrail-vrouter-masters` pod.

In one terminal, enter this command:

```
kubectl get pods -n contrail
```

The output should be a single line that looks like this:

```
NAME READY STATUS RESTARTS AGE
contrail-vrouter-masters-97v8z 3/3 Running 0 6h10m
```

This output gives you the name and specific instance hash of the vRouter pod, `contrail-vrouter-masters-97v8z`. We use this name in the next command to access the vRouter CLI. The name of your vRouter pod will have a different hash at the end. Use the pod name from your system in place of `<contrail-vrouter-masters-hash>` in the following command.

Enter the following command:

```
kubectl exec -n contrail -it <contrail-vrouter-masters-hash> -- bash
```

You should see the following two-line output:

```
Defaulted container "contrail-vrouter-agent" out of: contrail-vrouter-agent, contrail-vrouter-agent-dpdk, contrail-vrouter-telemetry-exporter, contrail-init (init), contrail-vrouter-kernel-init-dpdk (init)
root@jcnr1:/#
```

Note that the shell prompt has changed from what it was when you entered the command. On the system we used to create this example, the prompt changed from `[root@jcnr1 ~]#` to `root@jcnr1:/#`. This change in prompt indicates that you have successfully connected to the CLI of the vRouter agent.

You can now see the following detailed steps to complete the example.

## Detailed Steps

### 1. View the vRouter-agent interface list.

In the terminal session connected to the vRouter-agent CLI, enter the following command:

```
vif --list
```

The output looks like this:

```
Vrouter Operation Mode: PureL2
Vrouter Interface Table
```

Flags: P=Policy, X=Cross Connect, S=Service Chain, Mr=Receive Mirror  
 Mt=Transmit Mirror, Tc=Transmit Checksum Offload, L3=Layer 3, L2=Layer 2  
 D=DHCP, Vp=Vhost Physical, Pr=Promiscuous, Vnt=Native Vlan Tagged  
 Mnp=No MAC Proxy, Dpdk=DPDK PMD Interface, Rfl=Receive Filtering Offload,  
 Mon=Interface is Monitored  
 Uuf=Unknown Unicast Flood, Vof=VLAN insert/strip offload, Df=Drop New Flows, L=MAC  
 Learning Enabled  
 Proxy=MAC Requests Proxied Always, Er=Etree Root, Mn=Mirror without Vlan Tag, HbsL=HBS  
 Left Intf  
 HbsR=HBS Right Intf, Ig=Igmp Trap Enabled, Ml=MAC-IP Learning Enabled, Me=Multicast  
 Enabled

```
vif0/0      Socket: unix
             Type:Agent HWaddr:00:00:5e:00:01:00
             Vrf:65535 Flags:L2 QOS:-1 Ref:3
             RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0 0
             RX packets:0 bytes:0 errors:0
             TX packets:2127928 bytes:510246290 errors:0
             Drops:0

vif0/1      PCI: 0000:00:00.0 (Speed 10000, Duplex 1)
             Type:Physical HWaddr:3a:b2:ee:fe:a7:62
             Vrf:65535 Flags:L2Vof QOS:-1 Ref:12
             RX queue packets:172174354904 errors:20998087137
             RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 7293345594 6559356797 7145384746
             Fabric Interface: eth_bond_bond0 Status: UP Driver: net_bonding
             Slave Interface(0): 0000:18:09.1 Status: UP Driver: net_iavf
             Slave Interface(1): 0000:18:05.1 Status: UP Driver: net_iavf
             Vlan Mode: Trunk Vlan: 100 200 300 500 3001-3004 3201-3250
             RX packets:172172714121 bytes:33745848248728 errors:1642900
             TX packets:2272839360 bytes:4536582775436 errors:0
             Drops:80189427216

vif0/2      PCI: 0000:18:01.1 (Speed 1000, Duplex 1)
             Type:Physical HWaddr:a6:c1:0b:12:8c:44
             Vrf:65535 Flags:L2Vof QOS:-1 Ref:12
             RX queue packets:108 errors:0
             RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0 0
             Fabric Interface: 0000:18:01.1 Status: DOWN Driver: net_iavf
             Vlan Mode: Trunk Vlan: 100 200 300 500 3001-3004 3201-3250
             RX packets:0 bytes:0 errors:108
             TX packets:61278711540 bytes:15781059334468 errors:0
```



```

Drops:108

vif0/3    PCI: 0000:18:0d.1 (Speed 10000, Duplex 1)
          Type:Physical HWaddr:7a:30:33:68:6c:70
          Vrf:65535 Flags:L2Vof QOS:-1 Ref:12
          RX queue  packets:91255 errors:626
          RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 120 397 109
          Fabric Interface: 0000:18:0d.1 Status: UP Driver: net_iavf
          Vlan Mode: Trunk Vlan: 100 200 300 500 3001-3004 3201-3250
          RX packets:2015 bytes:170734 errors:89240
          TX packets:61279338241 bytes:15781182125402 errors:0
          Drops:91703

vif0/4    PCI: 0000:86:00.0 (Speed 1000, Duplex 1)
          Type:Physical HWaddr:40:a6:b7:0d:7b:b8
          Vrf:65535 Flags:TcL2Vof QOS:-1 Ref:12
          RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0 0
          Fabric Interface: 0000:86:00.0 Status: DOWN Driver: net_i40e
          Vlan Mode: Trunk Vlan: 100 200 300 500 3001-3004 3201-3250
          RX packets:0 bytes:0 errors:0
          TX packets:61278779459 bytes:15781072646592 errors:0
          Drops:0

```

As you can see, the vRouter agent knows about five interfaces

- vif0/0
- vif0/1
- vif0/2
- vif0/3
- vif0/4

["Back to high-level steps" on page 73](#)

## 2. Examine the NAD YAML file.

In this step, we look at a commented NAD file in the YAML format. The comments start with a hash symbol (#) and are displayed in boldface. You do not need to change most of the values because this file contains a NAD example. The NAD specifies the parameters of a virtual device that enables the pod to connect to the network. You can use this example file on your cloud-native router only if you remove the comments from the file.

```
cat nad-kernel_access_bd3001.yaml
```

```
apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: nad-vswitch-bd3001 #Name of the NAD
spec:
  config: '{
    "cniVersion": "0.4.0",
    "name": "nad-vswitch-bd3001", #This is the name of the NAD as it appears in the K8s
cluster.
    "capabilities": {"ips": true},
    "plugins": [
      {
        "type": "jcnr", #Always define as jcnr
        "args": {
          "instanceName": "vswitch", #Prefix for the name of the NAD within the cluster
          "instanceType": "virtual-switch", #What type of NAD is this?
          "bridgeDomain": "bd3001", #The name of the bridge domain
          "bridgeVlanId": "3001", #Which VLAN ID is allowed on this bridge domain
          "dataplane": "dpdk", #Which dataplane to use. Options are dpdk and kernel
          "mtu": "9000",
          "interfaceType": "veth" #Options are veth or virtio. For this example veth is the
correct option
        },
        "ipam": {
          "type": "static", #IP address type. Leave as static in 22.2 release
          "capabilities": {"ips": true}, #Does this ipam definition support IP addresses?
          "addresses": [
            {
              "address": "2001:db8:3001::10.30.0.1/64", #IPv6 address of the IFL
              "gateway": "2001:db8:3001::10.30.0.254" #IPv6 gateway for the IFL
            },
            {
              "address": "10.30.0.1/24", #IPv4 address for the IFL
              "gateway": "10.30.0.254" #IPv4 Gateway for the IFL
            }
          ]
        }
      }
    ],
    "kubeConfig": "/etc/kubernetes/kubelet.conf"
  }
```

```
]
}'
```

When you apply the NAD YAML file to the cluster, the virtual device becomes visible in the Kubernetes cluster.

["Back to high-level steps" on page 73](#)

### 3. Apply the NAD to the cluster.

If you use the same file name for your version of the NAD file, you can run the following command on the host-server CLI:

```
kubectl apply -f nad-kernel_access_vlan_3001.yaml
```

The command output looks like this:

```
networkattachmentdefinition.k8s.cni.cncf.io/nad-vswitch-bd3001 created
```

["Back to high-level steps" on page 73](#)

### 4. Verify the NAD.

On the host-server CLI, issue the following command:

```
kubectl get network-attachment-definitions
```

The output from the command looks like this:

NAME	AGE
vswitch	1d
nad-vswitch-bd3001	3m47s

["Back to high-level steps" on page 73](#)

### 5. Examine the example Pod YAML file.

Similar to the NAD YAML file, the pod YAML file or the pod definition specifies the configuration of the user pod that you want to create. In this example, we create a pod that works with the pod1-vswitch-bd3001 NAD that we just applied. You can use the following file on your cloud-native router deployment .

```
cat pod-kernel-access-vlan-3001.yaml
```

```
apiVersion: v1
kind: Pod
```

```

metadata:
  name: odu-kernel-pod-bd3001
  annotations:
    k8s.v1.cni.cncf.io/networks: pod1-vswitch-bd3001
spec:
  containers:
    - name: odu-kernel-pod-bd3001
      image: svl-artifactory.juniper.net/junos-docker-local/warthog/pktgen19116:20210303
      imagePullPolicy: IfNotPresent
      command: ["/bin/bash", "-c", "sleep infinity"]
      securityContext:
        privileged: false
      env:
        - name: KUBERNETES_POD_UID
          valueFrom:
            fieldRef:
              fieldPath: metadata.uid
      volumeMounts:
        - name: dpdk
          mountPath: /dpdk
          subPathExpr: ${KUBERNETES_POD_UID}
  volumes:
    - name: dpdk
      hostPath:
        path: /var/run/jcnr/containers

```

["Back to high-level steps" on page 73](#)

## 6. Apply the pod to the cluster.

If you have used the preceding filename for the pod YAML file, then you can run the following command on the host-server CLI to apply the pod to the cluster:

```
kubectl apply -f pod-kernel-access-vlan-3001.yaml
```

The command output looks like this:

```
pod/odu-kernel-pod-bd3001 created created
```

["Back to high-level steps" on page 73](#)

## 7. Verify the pod.

On the host-server CLI, issue the following command to verify the pod creation:

```
kubect1 get pods odu-kernel-pod-bd3001
```

The command output looks like this:

NAME	READY	STATUS	RESTARTS	AGE
odu-kernel-pod-bd3001	1/1	Running	0	58s

["Back to high-level steps" on page 73](#)

#### 8. View the updated vRouter interface list.

On the vRouter-agent CLI, issue the following command:

```
vif --list
```

The command output looks like this:

```
Flags: P=Policy, X=Cross Connect, S=Service Chain, Mr=Receive Mirror
      Mt=Transmit Mirror, Tc=Transmit Checksum Offload, L3=Layer 3, L2=Layer 2
      D=DHCP, Vp=Vhost Physical, Pr=Promiscuous, Vnt=Native Vlan Tagged
      Mnp=No MAC Proxy, Dpdk=DPDK PMD Interface, Rfl=Receive Filtering Offload,
      Mon=Interface is Monitored
      Uuf=Unknown Unicast Flood, Vof=VLAN insert/strip offload, Df=Drop New Flows, L=MAC
      Learning Enabled
      Proxy=MAC Requests Proxied Always, Er=Etree Root, Mn=Mirror without Vlan Tag, HbsL=HBS
      Left Intf
      HbsR=HBS Right Intf, Ig=Igmp Trap Enabled, Ml=MAC-IP Learning Enabled, Me=Multicast
      Enabled

vif0/0      Socket: unix
            Type:Agent HWaddr:00:00:5e:00:01:00
            Vrf:65535 Flags:L2 QOS:-1 Ref:3
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            RX packets:0 bytes:0 errors:0
            TX packets:14 bytes:1672 errors:0
            Drops:0

vif0/1      PCI: 0000:00:00.0 (Speed 10000, Duplex 1)
```

```

Type:Physical HWaddr:3a:b2:ee:fe:a7:62
Vrf:65535 Flags:L2Vof QOS:-1 Ref:12
RX queue  packets:971 errors:0
RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0 0
Fabric Interface: eth_bond_bond0 Status: UP Driver: net_bonding
Slave Interface(0): 0000:18:05.1 Status: UP Driver: net_iavf
Slave Interface(1): 0000:18:09.1 Status: UP Driver: net_iavf
Vlan Mode: Trunk Vlan: 100 200 300 500 3001-3004 3201-3250
RX packets:18 bytes:1256 errors:965
TX packets:26 bytes:2204 errors:0
Drops:989

vif0/2 PCI: 0000:18:01.1 (Speed 1000, Duplex 1)
Type:Physical HWaddr:a6:c1:0b:12:8c:44
Vrf:65535 Flags:L2Vof QOS:-1 Ref:12
RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0 0
Fabric Interface: 0000:18:01.1 Status: DOWN Driver: net_iavf
Vlan Mode: Trunk Vlan: 100 200 300 500 3001-3004 3201-3250
RX packets:0 bytes:0 errors:0
TX packets:37 bytes:2862 errors:0
Drops:0

vif0/3 PCI: 0000:18:0d.1 (Speed 10000, Duplex 1)
Type:Physical HWaddr:7a:30:33:68:6c:70
Vrf:65535 Flags:L2Vof QOS:-1 Ref:12
RX queue  packets:331 errors:0
RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0 0
Fabric Interface: 0000:18:0d.1 Status: UP Driver: net_iavf
Vlan Mode: Trunk Vlan: 100 200 300 500 3001-3004 3201-3250
RX packets:7 bytes:590 errors:324
TX packets:37 bytes:2870 errors:0
Drops:331

vif0/4 PCI: 0000:86:00.0 (Speed 1000, Duplex 1)
Type:Physical HWaddr:40:a6:b7:0d:7b:b8
Vrf:65535 Flags:TcL2Vof QOS:-1 Ref:12
RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0 0
Fabric Interface: 0000:86:00.0 Status: DOWN Driver: net_i40e
Vlan Mode: Trunk Vlan: 100 200 300 500 3001-3004 3201-3250
RX packets:0 bytes:0 errors:0
TX packets:37 bytes:2862 errors:0
Drops:0

```

```
vif0/5      Ethernet: jvknet1-0ea0f72
            Type:Virtual HWaddr:02:00:00:b3:b9:a1
            Vrf:0  Flags:L2Vof QOS:-1 Ref:10
            RX queue  packets:23 errors:0
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0
            Vlan Mode: Access  Vlan Id: 3001  OVlan Id: 3001
            RX packets:19  bytes:1614 errors:4
            TX packets:0  bytes:0 errors:0
            Drops:24
```

The vRouter agent now knows about six interfaces. This is because the Pod created the sub-interface and the parent interface. You can see above that the virtual VLAN interface, vif0/4, shows the parent interface as vif0/3. The interface, vif0/3, is a virtual interface with a name that includes "net1" as defined in the NAD and pod YAML files.

["Back to high-level steps" on page 73](#)

# L2 - Add User Pod with virtio Trunk Ports to a Cloud-Native Router Instance

## SUMMARY

Read this topic to learn how to add a user pod with a VLAN sub-interface to an instance of the cloud-native router.

## IN THIS SECTION

- [Overview | 83](#)
- [Before You Begin | 84](#)
- [Detailed Steps | 86](#)

## Overview

### IN THIS SECTION

- [High-Level Steps | 84](#)

To add a user pod to the cloud-native router, your high-level tasks are:

- Apply a network attachment definition (NAD) to your cluster.
- Apply a pod YAML file to your cloud-native router cluster.

Throughout this example, we use the `kubectl` command with various options. You must run this command on the host-server CLI.

## High-Level Steps

In this example, we assume that you are adding the first user pod to your newly installed cloud-native router. Therefore, we provide the steps to create a new NAD and then add the new user pod.

Below is a list of the individual steps we take in this example. Each step in the list is a link to the detailed description of the step.

1. [View the vRouter interface list](#)
2. [Examine the example NAD YAML file](#)
3. [Apply the NAD to the cluster](#)
4. [Verify the NAD](#)
5. [Examine the example Pod YAML file](#)
6. [Apply the Pod to the cluster](#)
7. [Verify the Pod](#)
8. [View the updated vRouter interface list](#)

## Before You Begin

### IN THIS SECTION

- [Access the vRouter-Agent CLI | 85](#)



## Access the vRouter-Agent CLI

You perform the first and last steps of this example procedure on the vRouter-agent CLI. We recommend that you open two SSH (terminal) sessions to the host server. You can use one session to run the CLI commands on the vRouter agent and the other session to run the `kubectl` commands that deploy the NAD and the pod on the cluster.

**NOTE:** We have not included paths or shell prompts from the host server in the command listings. Therefore you can easily copy commands from here to your system.

Access the CLI of the **contrail-vrouter-agent** container in the **contrail-vrouter-masters** pod.

In one terminal enter the following command:

```
kubectl get pods -n contrail
```

You will see a single line of output that looks like:

```
NAME READY STATUS RESTARTS AGE
contrail-vrouter-masters-97v8z 3/3 Running 0 6h10m
```

This command gives us the name and specific instance hash of the vRouter Pod, `contrail-vrouter-masters-97v8z`. We use this name in the next command to access the vRouter CLI. The name of your vRouter pod will have a different hash at the end. Use the pod name from your system in place of *contrail-vrouter-masters-hash* in the command below.

Enter the following command:

```
kubectl exec -n contrail -it <contrail-vrouter-masters-hash> -- bash
```

You will see the following two-line output:

```
Defaulted container "contrail-vrouter-agent" out of: contrail-vrouter-agent, contrail-vrouter-agent-dpdk, contrail-vrouter-telemetry-exporter, contrail-init (init), contrail-vrouter-kernel-init-dpdk (init)
root@jcnr1:/#
```

Note that the shell prompt has changed from what it was when you entered the command. On the system we used to create this example, the prompt changed from `[root@jcnr1 ~]#` to `root@jcnr1:/#`. This change in prompt indicates that you have successfully connected to the vRouter-agent CLI.

You can now see the following detailed steps to complete the example.

## Detailed Steps

### 1. View the vRouter-agent interface list

In the terminal session connected to the vRouter-agent, enter the following command in the CLI:

```
vif --list
```

The output looks like this:

```
Vrouter Operation Mode: PureL2
Vrouter Interface Table

Flags: P=Policy, X=Cross Connect, S=Service Chain, Mr=Receive Mirror
      Mt=Transmit Mirror, Tc=Transmit Checksum Offload, L3=Layer 3, L2=Layer 2
      D=DHCP, Vp=Vhost Physical, Pr=Promiscuous, Vnt=Native Vlan Tagged
      Mnp=No MAC Proxy, Dpdk=DPDK PMD Interface, Rfl=Receive Filtering Offload,
      Mon=Interface is Monitored
      Uuf=Unknown Unicast Flood, Vof=VLAN insert/strip offload, Df=Drop New Flows, L=MAC
      Learning Enabled
      Proxy=MAC Requests Proxied Always, Er=Etree Root, Mn=Mirror without Vlan Tag, HbsL=HBS
      Left Intf
      HbsR=HBS Right Intf, Ig=Igmp Trap Enabled, Ml=MAC-IP Learning Enabled, Me=Multicast
      Enabled

vif0/0      Socket: unix
            Type:Agent HWaddr:00:00:5e:00:01:00
            Vrf:65535 Flags:L2 QOS:-1 Ref:3
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            RX packets:0 bytes:0 errors:0
            TX packets:2127928 bytes:510246290 errors:0
            Drops:0

vif0/1      PCI: 0000:00:00.0 (Speed 10000, Duplex 1)
```

```

Type:Physical HWaddr:3a:b2:ee:fe:a7:62
Vrf:65535 Flags:L2Vof QOS:-1 Ref:12
RX queue  packets:172174354904 errors:20998087137
RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 7293345594 6559356797 7145384746
Fabric Interface: eth_bond_bond0 Status: UP Driver: net_bonding
Slave Interface(0): 0000:18:09.1 Status: UP Driver: net_iavf
Slave Interface(1): 0000:18:05.1 Status: UP Driver: net_iavf
Vlan Mode: Trunk Vlan: 100 200 300 500 3001-3004 3201-3250
RX packets:172172714121 bytes:33745848248728 errors:1642900
TX packets:2272839360 bytes:4536582775436 errors:0
Drops:80189427216

vif0/2 PCI: 0000:18:01.1 (Speed 1000, Duplex 1)
Type:Physical HWaddr:a6:c1:0b:12:8c:44
Vrf:65535 Flags:L2Vof QOS:-1 Ref:12
RX queue  packets:108 errors:0
RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Fabric Interface: 0000:18:01.1 Status: DOWN Driver: net_iavf
Vlan Mode: Trunk Vlan: 100 200 300 500 3001-3004 3201-3250
RX packets:0 bytes:0 errors:108
TX packets:61278711540 bytes:15781059334468 errors:0
Drops:108

vif0/3 PCI: 0000:18:0d.1 (Speed 10000, Duplex 1)
Type:Physical HWaddr:7a:30:33:68:6c:70
Vrf:65535 Flags:L2Vof QOS:-1 Ref:12
RX queue  packets:91255 errors:626
RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 120 397 109
Fabric Interface: 0000:18:0d.1 Status: UP Driver: net_iavf
Vlan Mode: Trunk Vlan: 100 200 300 500 3001-3004 3201-3250
RX packets:2015 bytes:170734 errors:89240
TX packets:61279338241 bytes:15781182125402 errors:0
Drops:91703

vif0/4 PCI: 0000:86:00.0 (Speed 1000, Duplex 1)
Type:Physical HWaddr:40:a6:b7:0d:7b:b8
Vrf:65535 Flags:TcL2Vof QOS:-1 Ref:12
RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Fabric Interface: 0000:86:00.0 Status: DOWN Driver: net_i40e
Vlan Mode: Trunk Vlan: 100 200 300 500 3001-3004 3201-3250
RX packets:0 bytes:0 errors:0

```

```
TX packets:61278779459  bytes:15781072646592  errors:0
Drops:0
```

As you can see, the vRouter agent knows about five interfaces.

["Back to high-level steps" on page 84](#)

## 2. Examine the NAD YAML file.

In this step, we look at a commented NAD file in the YAML format. The comments start with a hash symbol (#) and are in boldface. You do not need to change most of the values because this file contains a NAD example. The NAD specifies the parameters of a virtual device that enables the pod to connect to the network. You can use this example file on your cloud-native router only if you remove the comments from the file.

```
cat nad-virtio-trunk1.yaml
```

```
apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: vswitch-trunk9 #This is the name of the NAD as it appears in the K8s cluster.
spec:
  config: '{
    "cniVersion":"0.4.0",
    "name": "vswitch-trunk9", # Name of the NAD
    "type": "jcnr", # Always define as jcnr
    "args": {
      "instanceName": "vswitch",
      "instanceType": "virtual-switch",
      "dataplane": "dpdk", #Leave as DPDK for this example. Options are DPDK or kernel
      "vlanIdList": "3001, 3002, 3003, 3210" #List of allowed VLANs. You can allow a range
like this: 100-200
    },
    "kubeConfig":"/etc/kubernetes/kubelet.conf"
  }'
```

When you apply the NAD YAML file to the cluster, the virtual device becomes visible in the Kubernetes cluster.

["Back to high-level steps" on page 84](#)

## 3. Apply the NAD to the cluster.

If you use the same file name for your version of the NAD file, you can run the following command on the host-server CLI:

```
kubectl apply -f nad-virtio-trunk1.yaml
```

The command output looks like this:

```
networkattachmentdefinition.k8s.cni.cncf.io/nad-virtio-trunk1.yaml created
```

["Back to high-level steps" on page 84](#)

#### 4. Verify the NAD.

On the host-server CLI, issue the following command:

```
kubectl get network-attachment-definitions
```

The command output looks like this:

NAME	AGE
vswitch	25h
vswitch-trunk1	54s

["Back to high-level steps" on page 84](#)

#### 5. Examine the example pod YAML file.

Similar to the NAD YAML file, the pod YAML file or the pod definition specifies the configuration of the user pod that you want to create. In this example, we create a pod that works with the `vswitch-trunk1` NAD that we just applied. You can use the following example file on your cloud-native router deployment.

```
cat pod-virtio-trunk1.yaml
```

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-virtio-trunk1 #
  annotations:
    k8s.v1.cni.cncf.io/networks: vswitch-trunk1
spec:
  containers:
    - name: pod-virtio-trunk1
      image: svl-artifactory.juniper.net/junos-docker-local/warthog/pktgen19116:20210303
```

```

imagePullPolicy: IfNotPresent
securityContext:
  privileged: true
resources:
  requests:
    memory: 2Gi
  limits:
    hugepages-1Gi: 2Gi
env:
  - name: KUBERNETES_POD_UID
    valueFrom:
      fieldRef:
        fieldPath: metadata.uid
volumeMounts:
  - name: dpdk
    mountPath: /dpdk
    subPathExpr: $(KUBERNETES_POD_UID)
  - mountPath: /dev/hugepages
    name: hugepage
volumes:
  - name: dpdk
    hostPath:
      path: /var/run/jcni/containers
  - name: hugepage
    emptyDir:
      medium: HugePages

```

["Back to high-level steps" on page 84](#)

## 6. Apply the pod to the cluster.

If you have used the same filename shown above for the pod YAML file, then you can run the following command on the host-server CLI to apply the pod to the cluster:

```
kubectl apply -f pod-virtio-trunk1.yaml
```

The command output looks like this:

```
pod/pod-virtio-trunk1 created
```

["Back to high-level steps" on page 84](#)

## 7. Verify the pod.

On the host-server CLI, issue the following command to verify the pod creation:

```
kubect1 get pods pod-virtio-trunk1
```

The command output looks like this:

NAME	READY	STATUS	RESTARTS	AGE
pod-virtio-trunk1	1/1	Running	0	1m21s

["Back to high-level steps" on page 84](#)

## 8. View the updated vRouter interface list.

On the vRouter-agent CLI, issue the following command:

```
vif --list
```

The command output looks like this:

```
Vrouter Operation Mode: PureL2
Vrouter Interface Table

Flags: P=Policy, X=Cross Connect, S=Service Chain, Mr=Receive Mirror
      Mt=Transmit Mirror, Tc=Transmit Checksum Offload, L3=Layer 3, L2=Layer 2
      D=DHCP, Vp=Vhost Physical, Pr=Promiscuous, Vnt=Native Vlan Tagged
      Mnp=No MAC Proxy, Dpdk=DPDK PMD Interface, Rfl=Receive Filtering Offload,
      Mon=Interface is Monitored
      Uuf=Unknown Unicast Flood, Vof=VLAN insert/strip offload, Df=Drop New Flows, L=MAC
      Learning Enabled
      Proxy=MAC Requests Proxied Always, Er=Etree Root, Mn=Mirror without Vlan Tag, HbsL=HBS
      Left Intf
      HbsR=HBS Right Intf, Ig=Igmp Trap Enabled, Ml=MAC-IP Learning Enabled, Me=Multicast
      Enabled

vif0/0      Socket: unix
            Type:Agent HWaddr:00:00:5e:00:01:00
            Vrf:65535 Flags:L2 QOS:-1 Ref:3
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0
            RX packets:0 bytes:0 errors:0
            TX packets:205 bytes:74417 errors:0
```

```

Drops:0

vif0/1    PCI: 0000:00:00.0 (Speed 10000, Duplex 1)
          Type:Physical HWaddr:32:f8:ad:8c:d3:bc
          Vrf:65535 Flags:L2Vof QOS:-1 Ref:8
          RX queue  packets:3120 errors:0
          RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0
          Fabric Interface: eth_bond_bond0 Status: UP Driver: net_bonding
          Slave Interface(0): 0000:81:01.0 Status: UP Driver: net_iavf
          Vlan Mode: Trunk Vlan: 751-755
          RX packets:0 bytes:0 errors:7020
          TX packets:0 bytes:0 errors:0
          Drops:7020

vif0/2    PCI: 0000:81:09.0 (Speed 10000, Duplex 1)
          Type:Workload HWaddr:ca:ce:fc:d3:28:1e
          Vrf:0 Flags:L2Vof QOS:-1 Ref:7
          RX queue  packets:3120 errors:0
          RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0
          Fabric Interface: 0000:81:09.0 Status: UP Driver: net_iavf
          Vlan Mode: Access Vlan Id: 756 OVlan Id: 756
          RX packets:7020 bytes:1496820 errors:0
          TX packets:0 bytes:0 errors:0
          Drops:7215

vif0/3    PMD: vhostnet1-8ca7c251-481b-48
          Type:Virtual HWaddr:00:99:99:99:33:09
          Vrf:65535 Flags:L2 QOS:-1 Ref:10
          RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0
          RX packets:0 bytes:0 errors:0
          TX packets:0 bytes:0 errors:0
          Drops:0

vif0/4    Virtual: vhostnet1-8ca7c251-481b-48.3003 Vlan(o/i)(,S): 3003/3003 Parent:vif0/3
          Type:Virtual(Vlan) HWaddr:00:99:99:99:33:09
          Vrf:0 Flags:L2 QOS:-1 Ref:3
          RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0
          RX packets:0 bytes:0 errors:0
          TX packets:0 bytes:0 errors:0
          Drops:0

vif0/5    PMD: vhostnet1-35fee25e-7646-4281-ab
          Type:Virtual HWaddr:02:00:00:ac:88:fd

```



```
Vrf:65535 Flags:L2 QOS:-1 Ref:13
RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0 0
Vlan Mode: Trunk Vlan: 3001-3003 3210
RX packets:0 bytes:0 errors:0
TX packets:1 bytes:64 errors:0
Drops:0
TX port packets:0 errors:1
```

The vRouter agent now knows about six interfaces because the pod created the trunk interface. You can see in the preceding output that the vif0/5 interface shows the VLAN mode as Trunk and the correct list of VLANs that the vRouter allows to pass.

["Back to high-level steps" on page 84](#)

# L3 - Add User Pod to a Cloud-Native Router Instance

## SUMMARY

Read this topic to see an example of how to add an L3 user pod to a cloud-native router instance.

## IN THIS SECTION

- [Overview | 93](#)
- [Before You Begin | 94](#)
- [Detailed Steps | 96](#)

## Overview

## IN THIS SECTION

- [High-Level Steps | 94](#)

To add a user pod to the cloud-native router running in L3 mode, your high-level tasks are:

- Apply a network attachment definition (NAD) to your cluster.
- Apply a pod YAML file to your cloud-native router cluster.

Throughout this example, we use the `kubectl` command with various options. You must run this command on the host-server CLI.

## High-Level Steps

In this example, we assume that you are adding the first user pod to your newly installed cloud-native router. Therefore, we provide the steps to create a new NAD and then add the new user pod.

Below is a list of the individual steps we take in this example. Each step in the list is a link to the detailed description of the step.

1. [View the vRouter interface list](#)
2. [Examine the example NAD YAML file](#)
3. [Apply the NAD to the cluster](#)
4. [Verify the NAD](#)
5. [Examine the example Pod YAML file](#)
6. [Apply the Pod to the cluster](#)
7. [Verify the Pod](#)
8. [View the updated vRouter interface list](#)

## Before You Begin

### IN THIS SECTION

- [Access the vRouter-Agent CLI | 95](#)

## Access the vRouter-Agent CLI

You perform the first and last steps of this example procedure on the vRouter-agent CLI. We recommend that you open two SSH (terminal) sessions to the host server. You can use one session to run the CLI commands on the vRouter agent and the other session to run the `kubectl` commands that deploy the NAD and the pod on the cluster.

**NOTE:** We have not included paths or shell prompts from the host server in the command listings. Therefore you can easily copy commands from here to your system.

Access the CLI of the **contrail-vrouter-agent** container in the **contrail-vrouter-masters** pod.

In one terminal enter the following command:

```
kubectl get pods -n contrail
```

You will see a single line of output that looks like:

```
NAME READY STATUS RESTARTS AGE
contrail-vrouter-masters-6av7b 3/3 Running 0 9h1m
```

This command gives us the name and specific instance hash of the vRouter Pod, `contrail-vrouter-masters-97v8z`. We use this name in the next command to access the vRouter CLI. The name of your vRouter pod will have a different hash at the end. Use the pod name from your system in place of *contrail-vrouter-masters-hash* in the command below.

Enter the following command:

```
kubectl exec -n contrail -it contrail-vrouter-masters-hash -- bash
```

You will see the following two-line output:

```
Defaulted container "contrail-vrouter-agent" out of: contrail-vrouter-agent, contrail-vrouter-agent-dpdk, contrail-vrouter-telemetry-exporter, contrail-init (init), contrail-vrouter-kernel-init-dpdk (init)
root@jcnr1:/#
```

Note that the shell prompt has changed from what it was when you entered the command. On the system we used to create this example, the prompt changed from `[root@jcnr1 ~]#` to `root@jcnr1:/#`. This change in prompt indicates that you have successfully connected to the vRouter-agent CLI.

You can now use the following detailed steps to complete the example.

## Detailed Steps

### 1. View the vRouter-agent interface list

In the terminal session connected to the vRouter-agent, enter the following command in the CLI:

```
vif --list
```

The output looks like this:

```
vif --list
Vrouter Interface Table

Flags: P=Policy, X=Cross Connect, S=Service Chain, Mr=Receive Mirror
      Mt=Transmit Mirror, Tc=Transmit Checksum Offload, L3=Layer 3, L2=Layer 2
      D=DHCP, Vp=Vhost Physical, Pr=Promiscuous, Vnt=Native Vlan Tagged
      Mnp=No MAC Proxy, Dpdk=DPDK PMD Interface, Rfl=Receive Filtering Offload,
      Mon=Interface is Monitored
      Uuf=Unknown Unicast Flood, Vof=VLAN insert/strip offload, Df=Drop New Flows, L=MAC
      Learning Enabled
      Proxy=MAC Requests Proxied Always, Er=Etree Root, Mn=Mirror without Vlan Tag, HbsL=HBS
      Left Intf
      HbsR=HBS Right Intf, Ig=Igmp Trap Enabled, Ml=MAC-IP Learning Enabled, Me=Multicast
      Enabled

vif0/0      PCI: 0000:00:00.0 (Speed 20000, Duplex 1) NH: 4
            Type:Physical HWaddr:b4:96:91:e3:ba:52 IPaddr:0.0.0.0
            Vrf:0 Mcast Vrf:65535 Flags:TcL3L2VpVofErMe QOS:-1 Ref:15
            RX port   packets:3936680 errors:0
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            Fabric Interface: eth_bond_bond0 Status: UP Driver: net_bonding
            Slave Interface(0): 0000:31:00.0 Status: UP Driver: net_ice
            Slave Interface(1): 0000:31:00.1 Status: UP Driver: net_ice
            RX packets:3936680 bytes:376494695 errors:0
```

```

TX packets:1346179  bytes:103926504  errors:0
Drops:0
TX port  packets:1346162  errors:17

vif0/1  PMD: vhost0 NH: 5
Type:Host HWaddr:b4:96:91:e3:ba:52 IPAddr:17.1.0.205
IP6addr:2001:db8::b696:91ff:fee3:ba52
Vrf:0 Mcast Vrf:65535 Flags:L3DEr QOS:-1 Ref:15
RX device packets:1136926  bytes:88046862  errors:0
RX queue  packets:1136926  errors:0
RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0 0
RX packets:1136926  bytes:88046862  errors:0
TX packets:4133648  bytes:384767351  errors:0
Drops:0
TX queue  packets:4133648  errors:0
TX device packets:4133648  bytes:384767351  errors:0

vif0/2  Socket: unix
Type:Agent HWaddr:00:00:5e:00:01:00 IPAddr:0.0.0.0
Vrf:65535 Mcast Vrf:65535 Flags:L3Er QOS:-1 Ref:3
RX port  packets:406221  errors:0
RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0 0
RX packets:406221  bytes:42026022  errors:0
TX packets:65953  bytes:6995184  errors:0
Drops:0

```

As you can see, the vRouteragent knows about three interfaces.

["Back to high-level steps" on page 94](#)

## 2. Examine the NAD YAML file.

In this step, we look at a commented NAD file in the YAML format. The comments start with a hash symbol (#) and are in boldface. You do not need to change most of the values because this file contains a NAD example. The NAD specifies the parameters of a virtual device that enables the pod to connect to the network. You can use this example file on your cloud-native router only if you remove the comments from the file.

```
cat L3_nad-net1.yaml
```

```

apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:

```

```

name: net1 #This is the name of the NAD as it appears in the K8s cluster.
spec:
  config: '{
    "cniVersion": "0.4.0",
    "name": "net1", # Name of the NAD
    "type": "jcnr", #Always define as type jcnr
    "args": {
      "vrfName": "net1", #Virtual routing and forwarding instance name
      "vrfTarget": "1:11" #Virtual routing and forwarding target
    },
    "kubeConfig": "/etc/kubernetes/kubelet.conf"
  }'
```

When you apply the NAD YAML file to the cluster, the virtual device becomes visible in the Kubernetes cluster.

["Back to high-level steps" on page 94](#)

### 3. Apply the NAD to the cluster.

If you use the same file name for your version of the NAD file, you can run the following command on the host-server CLI:

```
kubectl apply -f L3_nad-net1.yaml
```

The command output looks like this:

```
networkattachmentdefinition.k8s.cni.cncf.io/L3_nad-net1.yaml created
```

["Back to high-level steps" on page 94](#)

### 4. Verify the NAD.

On the host-server CLI, issue the following command:

```
kubectl get network-attachment-definitions
```

The command output looks like this:

NAME	AGE
vswitch	22h
net1	41s

["Back to high-level steps" on page 94](#)

## 5. Examine the example pod YAML file.

Similar to the NAD YAML file, the pod YAML file or the pod definition specifies the configuration of the user pod that you want to create. In this example, we create a pod that works with the `net1` NAD that we just applied. You can use the following example file on your cloud-native router deployment.

```
cat L3-pktgen-odu1.yaml
```

```
apiVersion: v1
kind: Pod
metadata:
  name: L3-pktgen-odu1
  annotations:
    k8s.v1.cni.cncf.io/networks: |
      [
        {
          "name": "net1",
          "interface": "net1",
          "cni-args": {
            "mac": "aa:bb:cc:dd:ee:51",
            "dataplane": "vrouter",
            "ipConfig": {
              "ipv4": {
                "address": "10.1.51.2/30",
                "gateway": "10.1.51.1",
                "routes": [
                  "10.1.51.0/30"
                ]
              },
              "ipv6": {
                "address": "2001:db8::10:1:51:2/126",
                "gateway": "2001:db8::10:1:51:1",
                "routes": [
                  "2001:db8::10:1:51:0/126"
                ]
              }
            }
          }
        }
      ]
spec:
  containers:
    - name: L3-pktgen-odu1
```

```

image: svl-artifactory.juniper.net/blr-data-plane/dpdk-app/dpdk:21.11
imagePullPolicy: IfNotPresent
command: ["/bin/bash", "-c", "sleep infinity"]
securityContext:
  privileged: false
env:
  - name: KUBERNETES_POD_UID
    valueFrom:
      fieldRef:
        fieldPath: metadata.uid
resources:
  requests:
    memory: 4Gi
  limits:
    hugepages-1Gi: 4Gi
name: hugepages
command: ["sleep"]
args: ["infinity"]
volumeMounts:
  - name: dpdk
    mountPath: /dpdk
    subPathExpr: ${KUBERNETES_POD_UID}
  - name: hugepages
    mountPath: /hugepages
volumes:
  - name: dpdk
    hostPath:
      path: /var/run/jcnr/containers
  - name: hugepages
    emptyDir:
      medium: HugePages

```

["Back to high-level steps" on page 94](#)

## 6. Apply the pod to the cluster.

If you have used the same filename shown above for the pod YAML file, then you can run the following command on the host-server CLI to apply the pod to the cluster:

```
kubectl apply -f L3-pktgen-odu1.yaml
```

The command output looks like this:



pod/L3-pktgen-odu1 created

["Back to high-level steps" on page 94](#)

## 7. Verify the pod.

On the host-server CLI, issue the following command to verify the pod creation:

```
kubect1 get pods L3-pktgen-odu1
```

The command output looks like this:

NAME	READY	STATUS	RESTARTS	AGE
L3-pktgen-odu1	1/1	Running	0	1m11s

["Back to high-level steps" on page 94](#)

## 8. View the updated vRouter interface list.

On the vRouter-agent CLI, issue the following command:

```
vif --list
```

The command output looks like this:

```
vif --list
Vrouter Interface Table

Flags: P=Policy, X=Cross Connect, S=Service Chain, Mr=Receive Mirror
      Mt=Transmit Mirror, Tc=Transmit Checksum Offload, L3=Layer 3, L2=Layer 2
      D=DHCP, Vp=Vhost Physical, Pr=Promiscuous, Vnt=Native Vlan Tagged
      Mnp=No MAC Proxy, Dpdk=DPDK PMD Interface, Rfl=Receive Filtering Offload,
      Mon=Interface is Monitored
      Uuf=Unknown Unicast Flood, Vof=VLAN insert/strip offload, Df=Drop New Flows, L=MAC
      Learning Enabled
      Proxy=MAC Requests Proxied Always, Er=Etree Root, Mn=Mirror without Vlan Tag, HbsL=HBS
      Left Intf
      HbsR=HBS Right Intf, Ig=Igmp Trap Enabled, Ml=MAC-IP Learning Enabled, Me=Multicast
      Enabled

vif0/0      PCI: 0000:00:00.0 (Speed 20000, Duplex 1) NH: 4
            Type:Physical HWaddr:b4:96:91:e3:ba:52 IPaddr:0.0.0.0
            Vrf:0 Mcast Vrf:65535 Flags:TcL3L2VpVofErMe QOS:-1 Ref:15
```

```

RX port  packets:3936680 errors:0
RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0
Fabric Interface: eth_bond_bond0 Status: UP Driver: net_bonding
Slave Interface(0): 0000:31:00.0 Status: UP Driver: net_ice
Slave Interface(1): 0000:31:00.1 Status: UP Driver: net_ice
RX packets:3936680 bytes:376494695 errors:0
TX packets:1346179 bytes:103926504 errors:0
Drops:0
TX port  packets:1346162 errors:17

vif0/1  PMD: vhost0 NH: 5
Type:Host HWAddr:b4:96:91:e3:ba:52 IPAddr:10.17.1.205
IP6addr:2001:db8::b696:91ff:fee3:ba52
Vrf:0 Mcast Vrf:65535 Flags:L3DEr QOS:-1 Ref:15
RX device packets:1136926 bytes:88046862 errors:0
RX queue  packets:1136926 errors:0
RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0
RX packets:1136926 bytes:88046862 errors:0
TX packets:4133648 bytes:384767351 errors:0
Drops:0
TX queue  packets:4133648 errors:0
TX device packets:4133648 bytes:384767351 errors:0

vif0/2  Socket: unix
Type:Agent HWAddr:00:00:5e:00:01:00 IPAddr:0.0.0.0
Vrf:65535 Mcast Vrf:65535 Flags:L3Er QOS:-1 Ref:3
RX port  packets:406221 errors:0
RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0
RX packets:406221 bytes:42026022 errors:0
TX packets:65953 bytes:6995184 errors:0
Drops:0

vif0/3  PMD: vhostnet1-0fba9461-e5e4-4701-b1 NH: 17
Type:Virtual HWAddr:aa:bb:cc:dd:ee:51 IPAddr:10.1.51.2
IP6addr:2001:db8::10:1:51:2
Vrf:2 Mcast Vrf:2 Flags:PL3DEr QOS:-1 Ref:16
RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0
RX packets:0 bytes:0 errors:0
TX packets:0 bytes:0 errors:0
Drops:0

```

The vRouter agent now knows about four interfaces because the pod created the vif0/3 interface. You can see in the preceding output that the vif0/3 interface shows IPv4 and IPv6 addresses

["Back to high-level steps" on page 94](#)