



Cloud-native Network Function (CNF) Requirements

Version 1.3

January 2022

Table of Contents

1. Cloud-native Network Function Requirements	1
1.1. Introduction	1
1.2. Scope	1
1.3. Refactoring	1
1.4. Pods	2
2. CNF Developer Guidelines.....	3
2.1. Preface	3
2.2. Goals & Non-goals	3
2.3. Principle of Least Privilege.....	3
2.4. Avoid Accessing Resources on Host	4
2.5. Avoid Mounting host directories as volumes	4
2.6. Avoid the host's network namespace	4
2.7. Capabilities	4
2.7.1. IPC_LOCK.....	6
2.7.2. NET_ADMIN.....	6
2.7.3. (Avoid) SYS_ADMIN	7
2.7.4. SYS_NICE	7
2.7.5. SYS_PTRACE	7
2.8. Operations that can be executed by OpenShift.....	7
2.9. Operations that cannot be executed by OpenShift	9
2.10. Analyzing Your Application.....	10
2.11. Capabilities Example	10
3. CNF Best Practice	14
3.1. Control Plane and Management CNFs	14
4. Cloud-native CNFs - SCC Implementation	15
4.1. CNFs that do not require advanced networking features (Category 1)	15
4.2. CNFs that require advanced networking features (Category 2)	16
4.3. User-Plane CNFs (Category 3).....	18
5. CNF Expectations and Permissions	22
5.1. Cloud Native Design Best Practices.....	22
5.2. High Level CNF Expectations	23
5.3. Platform Restrictions	24
6. OpenShift Platform.....	26
7. Software Core/Edge	27
7.1. Helm v3	27

7.2. Kubernetes	27
7.3. CNI – OVN	27
7.4. Container storage (CSI)	28
7.5. Block storage	28
7.6. Object storage	28
7.7. Container Runtime	29
7.8. CPU Manager / Pinning	29
7.9. Host OS	29
8. PaaS Core/Edge	31
8.1. Certificate Management	31
8.2. Distributed Tracing	31
8.3. Pod Security	31
8.4. Load Balancer/Service Proxy	31
8.5. CI/CD Framework	31
8.6. Kubernetes API Versions	31
9. Pod Permissions	32
10. OpenShift Best Practices	33
10.1. Logging	33
10.2. Monitoring	34
10.3. CPU Allocation	35
10.3.1. NUMA Configuration	35
10.4. Memory Allocation	35
10.5. Affinity / Anti-affinity	36
10.6. Taints and Tolerations	36
10.7. Requests / Limits	37
10.8. Pods	37
10.8.1. No naked pods	37
10.8.2. Image tagging	37
10.8.3. One process per container	38
10.8.4. init containers	38
10.9. Security / RBAC	38
10.10. Multus	39
10.10.1. Multus SR-IOV / MACVLAN	39
10.10.2. SR-IOV Interface Settings	40
10.10.3. Attaching the VF to a pod	43
10.10.4. Discovering SR-IOV devices properties from the application	44
10.10.5. NUMA Awareness	45
10.11. Upgrades	46

10.11.1. Handling platform upgrades	46
10.12. OpenShift Virtualization / kubevirt	46
10.12.1. Openshift Virtualization and VMs (CNV) best practices	46
10.12.2. VM image Import Recommendations (CDI)	47
11. Operator Best Practices	49
12. Container Best Practices	50
12.1. Pod Exit Status	50
12.2. Graceful Termination	50
12.3. Pod Resource Profiles	51
12.4. Storage: emptyDir	51
12.5. Liveness and Readiness Probes	51
12.6. Use imagePullPolicy: IfNotPresent	52
12.7. Automount Services for Pods	52
12.8. Disruption budgets	53
13. Networking Overview	54
13.1. OVN-kubernetes CNI	54
14. User Plane Functions	55
14.1. Performance Addon Operator	55
14.2. Hugepages	55
14.3. CPU Isolation	56
14.4. NUMA Awareness	59
15. Application Service Exposure to External Networks	61
16. Service Mesh for Inter/Intra NF	63
16.1. Service Mesh Introduction	63
16.2. Service Mesh Tapping	65
16.3. Service Mesh Requirements for CNF	66
17. Application Deployment	68
18. Standards	69
18.1. Container Labeling Standards	69
18.2. Image Standards	69
18.2.1. Universal Base Image information	70
19. Security	72
19.1. Elevated privilege container capabilities	72
19.2. CPI-810	72
19.3. Image Security	72
19.4. CNF Network Security	72
19.5. Secrets Management	72
20. Contributors	74

21. Document History	75
22. Document Approvals	76

Chapter 1. Cloud-native Network Function Requirements

1.1. Introduction

Red Hat is building a Telco platform to serve network needs across Core, Edge, Lite and Far Edge. In this journey, the platform is the entry path for 5G Core Cloud-native Network Functions (CNFs) and brings forth the introduction of a CNCF-based stack for CNFs.

In addition, Red Hat is also building a Kubernetes-based Container as a Service (CaaS) Platform with Platform as a Service (PaaS) services to support 5G Services-based architecture.

1.2. Scope

This document, and the current platform configuration, are currently limited in scope to Wireless network elements.

This document covers the requirements for tenants to run their application on Red Hat's OpenShift Network Functions Virtualization Infrastructure (NFVI) platform.

1.3. Refactoring

Network Functions (NFs) break their software down into the smallest sets of microservices possible. Running monolithic applications inside a container is not the operating model recommended by Red Hat.

It is hard to move a 1,000 lb boulder. However, it is easy when that boulder is broken down into many pieces. All CNFs break apart each piece of the functions/services/processes into separate containers. These containers can still be within OpenShift pods, and all of the functions that perform a single task must be within the same namespace.

There is a quote that describes this best from [Lewis and Fowler](#): "the microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery."

1.4. Pods

Pods are the smallest deployable units of computing that can be created and managed in Kubernetes.

A pod can contain one or more running containers at a time. Containers running in the same pod have access to several of the same Linux namespaces. For example, each application has access to the same network namespace, meaning that one running container can communicate with another running container over `127.0.0.1:<port>`.

The same is true for storage volumes. All containers in the same pod have access to the same mount namespace, and can mount the same volumes.

Chapter 2. CNF Developer Guidelines

2.1. Preface

Cloud-native Network Functions (CNFs) are containerized instances of classic physical or Virtual Network Functions (VNFs) which have been decomposed into microservices supporting elasticity, lifecycle management, security, logging, and other capabilities in a Cloud-Native format.

2.2. Goals & Non-goals

This section is mainly for the developers of CNFs, who need to build high-performance Network Functions (NFs) in a containerized environment. The guidance provided in these guidelines should assist partners when developing their CNFs so that they can be deployed on the OpenShift Container Platform (OCP) in a secure, efficient and supportable way.

These guidelines do not detail how to build CNF's functionality.

2.3. Principle of Least Privilege

In OpenShift Container Platform (OCP), it is possible to run privileged containers that have all of the root capabilities on a host machine, allowing the ability to access resources which are not accessible in ordinary containers. This, however, increases the security risk to the whole cluster. Containers should only request those privileges they need to run their legitimate functions. No containers will be allowed to run with full privileges without an exception.

The general guidelines are:

1. Only ask for the necessary privileges and access control settings for your application.
2. If the function required by your CNF can be fulfilled by OCP components, your application must not be requesting escalated privileges to perform this function.
3. If possible, avoid using any host system resource.
4. Leveraging read only root filesystem when possible.

2.4. Avoid Accessing Resources on Host

It is not recommended for an application to access the following resources on the host.

2.5. Avoid Mounting host directories as volumes

It is not necessary to mount host `/sys/` or host `/dev/`` directory as a volume in a pod to use a network device, such as Single Root I/O Virtualization (SR-IOV) VF. The moving of a network interface into the pod network namespace is done automatically by the Container Network Interface (CNI). Mounting the whole `/sys/` or `/dev/` directory in the container overwrites the network device descriptor inside the container, which causes the `device not found` or `no such file or directory` error.

Network interface statistics can be queried inside the container using the same `/sys` path as was done when running directly on the host. When running network interfaces in containers, relevant `/sys/` statistics interfaces are available inside the container, such as `/sys/class/net/net1/statistics/`, `/proc/net/tcp` and `/proc/net/tcp6`.

For running Data Plane Development Kit (DPDK) applications with SR-IOV VF, device specifications (in the case of `vfio-pci`) are automatically attached to the container via the Device Plugin. There is no need to mount the `/dev/` directory as a volume in the container, as the application can find device specifications under `/dev/vfio/` in the container.

2.6. Avoid the host's network namespace

Application pods must avoid using `hostNetwork`. Applications may not use the host network, including `nodePort` for network communication. Any networking needs, beyond the functions provided by the pod network and ingress/egress proxy, must be serviced via a multus-connected interface.

2.7. Capabilities

Linux Capabilities allow you to break apart the power of root into smaller groups of privileges. The Linux `capabilities(7)` man page provides a detailed description of how capabilities management is performed in Linux. In brief the Linux kernel associates various capability sets with threads and files. The thread's Effective capability set determines the current privileges of a thread. When a thread executes a binary program the kernel updates the various thread capability sets according to a set of rules that take into account the UID of the thread before and after the

exec system call and the file capabilities of the program being executed. Refer to the blog series in [10] for more details about Linux capabilities and some examples.

Users may choose to specify the required permissions for their running application in the Security Context of the pod specification. In OCP, administrators can use the Security Context Constraint (SCC) admission controller plugin to control the permissions allowed for pods deployed to the cluster. If the pod requests permissions that are not allowed by the SCCs available to that pod, the pod will not be admitted to the cluster.

The following runtime and SCC attributes control the capabilities that will be granted to a new container:

- The capabilities granted to the CRI-O engine. The default capabilities are listed here: <https://github.com/cri-o/cri-o/blob/master/internal/config/capabilities/capabilities.go>

NOTE

As of version 1.18, CRI-O no longer runs with NET_RAW or SYS_CHROOT by default. <https://cri-o.github.io/cri-o/v1.18.0.html>

- The values in the SCC for allowedCapabilities, defaultAddCapabilities and requiredDropCapabilities
- allowPrivilegeEscalation: controls whether a container can acquire extra privileges through setuid binaries or the file capabilities of binaries

The capabilities associated with a new container are determined as follows:

- If the container has the UID 0 (root) its Effective capability set is determined according to the capability attributes requested by the pod or container security context and allowed by the SCC assigned to the pod. In this case, the SCC provides a way to limit the capabilities of a root container.
- If the container has a UID non 0 (non root), the new container has an empty Effective capability set (see <https://github.com/kubernetes/kubernetes/issues/56374>). In this case the SCC assigned to the pod controls only the capabilities the container may acquire through the file capabilities of binaries it will execute.

Considering the general recommendation to avoid running root containers, capabilities required by non-root containers are controlled by the pod or container security context and the SCC capability attributes but can only be acquired by properly setting the file capabilities of the container binaries.

For more information on how to define and use the SCC, see [Managing Security Context Constraints](#).

DEFAULT Capabilities

The default capabilities that are allowed via the restricted SCC are as follows:

<https://github.com/cri-o/cri-o/blob/master/internal/config/capabilities/capabilities.go>

```
"CHOWN", +
"DAC_OVERRIDE", +
"FSETID", +
"FOwner", +
"SETPCAP", +
"NET_BIND_SERVICE"
```

2.7.1. IPC_LOCK

IPC_LOCK capability is required if any of these functions are used in an application:

```
* mlock()
* mlockall()
* shmctl()
* mmap().
```

Even though ‘mlock’ is not necessary on systems where page swap is disabled (for example, on OpenShift), it may still be required as it is a function that is built into DPDK libraries, and DPDK-based applications may indirectly call it by calling other functions.

2.7.2. NET_ADMIN

NET_ADMIN capability is required to perform various network-related administrative operations inside containers, such as:

- MTU setting
- Link state modification
- MAC/IP address assignment
- IP address flushing
- Route insertion/deletion/replacement

- Control network driver and hardware settings via ‘ethtool’

This does not include:

- Adding or deleting a virtual interface inside a container. For example, adding a VLAN interface
- Setting VF device properties

All the administrative operations (except `ethtool`) mentioned above that require the `NET_ADMIN` capability are already supported on the host by various CNIs in OpenShift.

2.7.3. (Avoid) `SYS_ADMIN`

(Avoid) `SYS_ADMIN` capability is very powerful and overloaded. It allows the application to perform a range of system administration operations to the host. You should avoid requiring this capability in your application.

2.7.4. `SYS_NICE`

`SYS_NICE` capability is required when a CNF is running on a node using the real-time kernel, as it allows the real-time application to switch to `SCHED_FIFO`.

2.7.5. `SYS_PTRACE`

`SYS_PTRACE` capability is required when using Process Namespace Sharing. This is used when processes from one container need to be exposed to another Container.

For example, to send signals, like signal hang up (`SIGHUP`), from a process in one container to another process in another container. For more information, see [Share Process Namespace between Containers in a Pod](#).

2.8. Operations that can be executed by OpenShift

The application does not require `NET_ADMIN` capability to perform the following administrative operations:

- MTU setting
 - For the cluster network, also known as the OVN or OpenShift-SDN network, the MTU is

configured by modifying the manifests generated by OpenShift-installer before deploying the cluster. For more information, see [Installing a cluster on bare metal with network customizations](#).

- For the additional networks managed by the Cluster Network Operator, it can be configured through the NetworkAttachmentDefinition resources generated by the Cluster Network Operator. For more information, see [Understanding multiple networks](#).
- For the SR-IOV interfaces managed by the SR-IOV Network Operator. For more information, see [Configuring an SR-IOV network device](#).
- Link state modification
 - All the links are set to up before attaching it to a pod.
- IP/MAC address assignment
 - For all networks, the IP/MAC address is assigned to the interface during pod creation.
 - Multus also allows users to override the IP/MAC address. For more information, see [Specifying Pod-specific addressing and routing options](#).
- Manipulate pod's route table
 - By default, the default route of the pod points to the cluster network, with or without the additional networks. Multus also allows users to override the default route of the pod. For more information, see [Specifying Pod-specific addressing and routing options](#).
 - Non-default routes can be added to pod's routing table by various IPAM CNI plugins during pod creation
- SR-IOV VF setting
 - Besides the functions aforementioned, the SR-IOV Network Operator supports the configuration of the following parameters for SR-IOV VFs. For more information, see [Configuring an SR-IOV network attachment](#).
 - vlan
 - linkState
 - maxTxRate
 - minTxRate
 - vlanQoS
 - spoofChk

- trust
- Multicast
 - In OCP, multicast is supported for both the default interface (OVN or OpenShift-SDN) and the additional interfaces (macvlan, SR-IOV...). However, multicast is disabled by default. To enable multicast, see [Using multicast](#) and [Using high performance multicast](#).
 - If your application works as a multicast source, and you want to utilize the additional interfaces to carry the multicast traffic, then you do not need the NET_ADMIN capability. However, you must follow the instructions in [Using high performance multicast](#) to set the correct multicast route in your pod's routing table.

2.9. Operations that cannot be executed by OpenShift

All the CNI plugins are only invoked during pod creation and deletion. If your CNF wants to perform any operations mentioned in the above chapter at runtime, the NET_ADMIN capability is required.

There are some other functionalities that are not currently supported by any of the OpenShift components, which also require NET_ADMIN capability:

- Link state modification at runtime
- IP/MAC modification at runtime
- Manipulate pod's route table or firewall rules at runtime
- SRIOV VF setting at runtime
- Netlink configuration
 - For example, 'ethtool' can be used to configure things like rxvlan, txvlan, gso, and tso.
- Multicast
 - If your application works as a receiving member of IGMP groups, you need to specify the NET_ADMIN capability in the pod manifest. So that the app is allowed to assign multicast addresses to the pod interface and join an IGMP group.
- Set SO_PRIORITY to a socket to manipulate the 802.1p priority in ethernet frames
- Set IP_TOS to a socket to manipulate the DSCP value of IP packets

2.10. Analyzing Your Application

To find out which capabilities the application needs, Red Hat developed a SystemTap script (`container_check.stp`). With this tool, the CNF developer finds out what capabilities an application requires to run in a container. It also shows the syscalls which are invoked.

Another tool is `capable` which is part of the BCC tools. You can install it on RHEL8 with `dnf install bcc`.

2.11. Capabilities Example

Here is an example of how to find out the capabilities that an application needs. ‘testpmd’ is a DPDK based layer-2 forwarding application. It needs the `CAP_IPC_LOCK` to allocate the hugepage memory.

1.. Use `container_check.stp`. We can see `CAP_IPC_LOCK` and `CAP_SYS_RAWIO` are requested by `testpmd` and the relevant syscalls.

```

$ /usr/share/systemtap/examples/profiling/container_check.stp -c
'testpmd -l 1-2 -w 0000:00:09.0 -- -a --portmask=0x8 --nb-cores=1'
[...]
capabilities used by executables
    executable:      prob capability

    testpmd:         cap_ipc_lock
    testpmd:         cap_sys_rawio

capabilities used by syscalls
    executable,      syscall (      capability ) :      count
    testpmd,         mlockall (      cap_ipc_lock ) :          1
    testpmd,         mmap (      cap_ipc_lock ) :          710
    testpmd,         open (      cap_sys_rawio ) :           1
    testpmd,         iopl (      cap_sys_rawio ) :           1

forbidden syscalls
    executable,      syscall:          count

failed syscalls
    executable,      syscall =      errno:      count
eal-intr-thread,    epoll_wait =      EINTR:      1
lcore-slave-2,      read =              :      1
rte_mp_handle,      recvmsg =           :      1
    stapio,          =      EINTR:      1
    stapio,          execve =      ENOENT:      3
    stapio,          rt_sigsuspend =      :      1
testpmd,            flock =      EAGAIN:      5
testpmd,            stat =      ENOENT:      10
testpmd,            mkdir =      EEXIST:      2
testpmd,            readlink =      ENOENT:      3
testpmd,            access =      ENOENT:      1141
testpmd,            openat =      ENOENT:      1
testpmd,            open =      ENOENT:      13
[...]
```

2.. Use capable command

```
$ /usr/share/bcc/tools/capable
```

3.. Start the testpmd application from another terminal, and send some test traffic to it.

```
$ testpmd -l 18-19 -w 0000:01:00.0 -- -a --portmask=0x1 --nb-cores=1
```

4.. Check the output of the `capable` command. As we can see `CAP_IPC_LOCK` was requested for running `testpmd`.

```
[...]
00:41:58 0 3591 3591 testpmd 14 CAP_IPC_LOCK 1
00:41:58 0 3591 3591 testpmd 14 CAP_IPC_LOCK 1
00:41:58 0 3591 3591 testpmd 14 CAP_IPC_LOCK 1
00:41:58 0 3591 3591 testpmd 14 CAP_IPC_LOCK 1
00:41:58 0 3591 3591 testpmd 14 CAP_IPC_LOCK 1
00:41:58 0 3591 3591 testpmd 14 CAP_IPC_LOCK 1
00:41:58 0 3591 3591 testpmd 14 CAP_IPC_LOCK 1
00:41:58 0 3591 3591 testpmd 14 CAP_IPC_LOCK 1
00:41:58 0 3591 3591 testpmd 14 CAP_IPC_LOCK 1
00:41:58 0 3591 3591 testpmd 14 CAP_IPC_LOCK 1
00:41:58 0 3591 3591 testpmd 14 CAP_IPC_LOCK 1
00:41:58 0 3591 3591 testpmd 14 CAP_IPC_LOCK 1
00:41:58 0 3591 3591 testpmd 14 CAP_IPC_LOCK 1
00:41:58 0 3591 3591 testpmd 14 CAP_IPC_LOCK 1
[...]
```

5.. Also, we can try to run `testpmd` without the `CAP_IPC_LOCK` with `capsh`. Now we can see that the hugepage memory cannot be allocated.

```

$ capsh --drop=cap_ipc_lock -- -c testpmd -l 18-19 -w 0000:01:00.0 --
-a --portmask=0x1 --nb-cores=1
EAL: Detected 24 lcore(s)
EAL: Detected 2 NUMA nodes
EAL: Multi-process socket /var/run/dpdk/rte/mp_socket
EAL: No free hugepages reported in hugepages-1048576kB
EAL: Probing VFIO support...
EAL: VFIO support initialized
EAL: PCI device 0000:01:00.0 on NUMA socket 0
EAL: probe driver: 8086:10fb net_ixgbe
EAL: using IOMMU type 1 (Type 1)
EAL: Ignore mapping IO port bar(2)
EAL: PCI device 0000:01:00.1 on NUMA socket 0
EAL: probe driver: 8086:10fb net_ixgbe
EAL: PCI device 0000:07:00.0 on NUMA socket 0
EAL: probe driver: 8086:1521 net_e1000_igb
EAL: PCI device 0000:07:00.1 on NUMA socket 0
EAL: probe driver: 8086:1521 net_e1000_igb
EAL: cannot set up DMA remapping, error 12 (Cannot allocate memory)
testpmd: mlockall() failed with error "Cannot allocate memory"
testpmd: create a new mbuf pool <mbuf_pool_socket_0>: n=331456,
size=2176, socket=0
testpmd: preferred mempool ops selected: ring_mp_mc
EAL: cannot set up DMA remapping, error 12 (Cannot allocate memory)
testpmd: create a new mbuf pool <mbuf_pool_socket_1>: n=331456,
size=2176, socket=1
testpmd: preferred mempool ops selected: ring_mp_mc
EAL: cannot set up DMA remapping, error 12 (Cannot allocate memory)
EAL: cannot set up DMA remapping, error 12 (Cannot allocate memory)

```

Chapter 3. CNF Best Practice

The design and implementation of CNFs may be varied. However, from the platform networking perspective, we can put them into the following categories. Here, we have some recommendations for each kind of application on the capabilities it requests.

3.1. Control Plane and Management CNFs

Vocabulary	
CNF	Cloud-native Network Function
CNI	Container Network Interface
DPDK	Data Plane Development Kit
DSCP	Differentiated Services Code Point
IP	Internet Protocol
MTU	Maximum Transmission Unit
OVN	Open Virtual Network
PF	Physical Function
PMD	Poll Mode Driver
QoS	Quality of Service
RHEL	Red Hat Enterprise Linux
SR-IOV	Single Root I/O Virtualization
VLAN	Virtual Local Area Network
VF	Virtual Function
VPP	Vector Packet Processor

Chapter 4. Cloud-native CNFs - SCC Implementation

There are three Security Context Constraint (SCC) profiles that may be utilized by OpenShift Container Platform. All Apps are expected to fit in Category 1 employing the default SCC profile. Apps matching Category 2 or Category 3 require exception approval. Further details are listed in the following sections.

4.1. CNFs that do not require advanced networking features (Category 1)

These kind of CNFs:

1. Use the default CNI (OVN) network interface.
2. Do not request 'NET_ADMIN' or 'NET_RAW' for advanced networking functions.

Recommended SCC definition (default):

```

kind: SecurityContextConstraints
apiVersion: security.openshift.io/v1
metadata:
  name: cnf-catalog-1
users: []
groups: []
priority: null
allowHostDirVolumePlugin: false
allowHostIPC: false
allowHostNetwork: false
allowHostPID: false
allowHostPorts: false
allowPrivilegeEscalation: true
allowPrivilegedContainer: false
allowedCapabilities: null
defaultAddCapabilities: null
requiredDropCapabilities:
  - KILL
  - MKNOD
  - SETUID
  - SETGID
  - NET_RAWD
fsGroup:
  type: MustRunAs
readOnlyRootFilesystem: false
runAsUser:
  type: MustRunAsRange
seLinuxContext:
  type: MustRunAs
supplementalGroups:
  type: RunAsAny
volumes:
  - configMap
  - downwardAPI
  - emptyDir
  - persistentVolumeClaim
  - projected
  - secret

```

4.2. CNFs that require advanced networking features (Category 2)

CNFs with the following characteristics may fall into this category:

1. Manipulate the low-level protocol flags, such as the 802.1p priority, the VLAN tag, and the DSCP value.

2. Manipulate the interface IP addresses, the routing table, or the nftables on-the-fly.
3. Process Ethernet packets.

These kind of CNFs:

1. Use Macvlan interface to send and receive Ethernet packets
2. Request CAP_NET_RAW for creating raw sockets
3. Request CAP_NET_ADMIN for:
 - a. Modifying the interface IP address on-the-fly
 - b. Manipulating the routing table on-the-fly
 - c. Manipulating the iptables rules on-the-fly
 - d. Setting the packet DSCP value

Recommended SCC definition:

```

kind: SecurityContextConstraints
apiVersion: security.openshift.io/v1
metadata:
  name: cnf-catalog-2
users: []
groups: []
priority: null
allowHostDirVolumePlugin: false
allowHostIPC: false
allowHostNetwork: false
allowHostPID: false
allowHostPorts: false
allowPrivilegeEscalation: true
allowPrivilegedContainer: false
allowedCapabilities: [NET_ADMIN, NET_RAW]
defaultAddCapabilities: null
requiredDropCapabilities:
  - KILL
  - MKNOD
  - SETUID
  - SETGID
fsGroup:
  type: MustRunAs
readOnlyRootFilesystem: false
runAsUser:
  type: MustRunAsRange
seLinuxContext:
  type: MustRunAs
supplementalGroups:
  type: RunAsAny
volumes:
  - configMap
  - downwardAPI
  - emptyDir
  - persistentVolumeClaim
  - projected
  - secret

```

4.3. User-Plane CNFs (Category 3)

A CNF which handles user-plane traffic or latency-sensitive payloads at line rate falls into this category, such as load balancing, routing, deep packet inspection, and so on. Some of these CNFs may also need to process the packets at a lower level.

These kind of CNFs:

1. Use SR-IOV interfaces.
2. Fully or partially bypass the kernel networking stack with userspace networking technologies, like DPDK, F-stack, VPP, and OpenFastPath. A userspace networking stack can not only improve the performance but also reduce the need for the ‘CAP_NET_ADMIN’ and ‘CAP_NET_RAW’.

NOTE

For those using Mellanox devices, those capabilities are requested if the application needs to configure the device(CAP_NET_ADMIN) and/or allocate raw ethernet queue through kernel drive(CAP_NET_RAW)

As ‘CAP_IPC_LOCK’ is mandatory for allocating HugePage memory, this capability is granted to the DPDK-based applications. Additionally, if the workload is latency-sensitive, and needs the determinacy provided by the real-time kernel, the ‘CAP_SYS_NICE’ would also be required.

The following is an example pod manifest of a DPDK application. For more information, see [Using virtual functions \(VFs\) with DPDK and RDMA modes](#).

```

apiVersion: v1
kind: Pod
metadata:
  name: dpdk-app
  namespace: <target_namespace>
  annotations:
    k8s.v1.cni.cncf.io/networks: dpdk-network
spec:
  containers:
  - name: testpmd
    image: <DPDK_image>
    securityContext:
      capabilities:
        add: ["IPC_LOCK"]
    volumeMounts:
    - mountPath: /dev/hugepages
      name: hugepage
  resources:
    limits:
      openshift.io/mlxnlcs: "1"
      memory: "1Gi"
      cpu: "4"
      hugepages-1Gi: "4Gi"
    requests:
      ++openshift.io/mlxnlcs: "1"
      memory: "1Gi"
      cpu: "4"
      hugepages-1Gi: "4Gi"
    command: ["sleep", "infinity"]
  volumes:
  - name: hugepage
emptyDir:
  medium: HugePages

```

Recommended SCC definition:

```
kind: SecurityContextConstraints
apiVersion: security.openshift.io/v1
metadata:
  name: cnf-catalog-3
users: []
groups: []
priority: null
allowHostDirVolumePlugin: false
allowHostIPC: false
allowHostNetwork: false
allowHostPID: false
allowHostPorts: false
allowPrivilegeEscalation: true
allowPrivilegedContainer: false
allowedCapabilities: [IPC_LOCK, NET_ADMIN, NET_RAW]
defaultAddCapabilities: null
requiredDropCapabilities:
  - KILL
  - MKNOD
  - SETUID
  - SETGID
fsGroup:
  type: MustRunAs
readOnlyRootFilesystem: false
runAsUser:
  type: MustRunAsRange
seLinuxContext:
  type: MustRunAs
supplementalGroups:
  type: RunAsAny
volumes:
  - configMap
  - downwardAPI
  - emptyDir
  - persistentVolumeClaim
  - projected
  - secret
```

Chapter 5. CNF Expectations and Permissions

5.1. Cloud Native Design Best Practices

Cloud-native applications are developed as loosely-coupled, well-behaved manageable microservices running in containers managed by a container orchestration engine, such as kubernetes.

The following sections highlight some key principles of cloud-native application design.

Single Purpose with Messaging Interface

A container addresses a single purpose with a well-defined (typically RESTful API) messaging interface. The motivation here is that such a container image is more reusable and more replaceable/upgradeable.

High Observability

A container must provide APIs for the platform to observe the container health and act accordingly. These APIs include health checks (liveness and readiness), logging to stderr and stdout for log aggregation (by tools such as Logstash or Filebeat), and integrate with tracing and metrics-gathering libraries (such as Prometheus or Metricbeat).

Lifecycle Conformance

A container must receive important events from the platform and conform/react to these events properly. For example, a container catches SIGTERM or SIGKILL from the platform and shuts down as quickly as possible. Other typically important events from the platform are PostStart to initialize before servicing requests and PreStop to release resources cleanly before shutting down.

Image Immutability

Container images are meant to be immutable, that is, customized images for different environments are typically not built. Instead, an external means for storing and retrieving configurations that vary across environments for the container is used.

Additionally, the container image does not dynamically install additional packages at runtime.

Process Disposability

Containers are as ephemeral as possible, and ready to be replaced by another container instance at any point in time. There are many reasons to replace a container, such as failing a health check, scaling down the application, migrating the containers to a different host, platform

resource starvation, or another issue.

This means that containerized applications must keep their state externalized or distributed and redundant. To store files or block level data, persistent volume claims are used. For information such as user sessions, use of an external, low-latency, key-value store such as redis is used.

Process disposability also requires that the application be quick in starting up and shutting down, and even be ready for a sudden, complete hardware failure.

Another helpful practice in implementing this principle is to create small containers. Containers in cloud-native environments may be automatically scheduled and started on different hosts. Having smaller containers leads to quicker start-up times because before being restarted, containers need to be physically copied to the host system.

A corollary of this practice is to ‘retry instead of crashing’. This is when one service in your application depends on another service, it does not crash when the other service is unreachable. For example, your API service is starting up and detects the database is unreachable. Instead of failing and refusing to start, you design it to retry the connection. While the database connection is down, the API can respond with a 503 status code, telling the clients that the service is currently unavailable. This practice may already be followed by applications, but if you are working in a containerized environment where instances are disposable, then the need for it becomes more obvious.

Also related to this, by default containers are launched with shared images using COW filesystems which only exist as long as the container exists. Mounting Persistent Volume Claims enables a container to have persistent physical storage. Clearly defining the abstraction for what storage is persisted promotes the idea that instances are disposable.

5.2. High Level CNF Expectations

- CNFs should be built to be cloud-native
- Containers never run as root (uid=0). Applications that require elevated privileges require an exception with HQ Planning
- Containers run with the minimal set of permissions required. Any pods that require elevated privileges, also require a security review that provides an analysis of the special permissions required. Any exceptions should be provided. Avoid Privileged Pods. If Privileged Pods are required, the CNF developer should work with the planning department, security risk management, and Red Hat to determine acceptability of privilege and/or modifications to

Pods such that elevated privilege is not required.

- Use the main CNI for all traffic
- CNFs should leverage service mesh provided by the platform for internal and external communication
- CNFs should leverage platform service mesh for mTLS with other applications
- All images/helm charts must be packaged by the vendor and hosted on the image registry
- Naming and Labelling standards for all kubernetes objects (for example, Pods and Services) should be provided
- CNFs should employ N+k redundancy models
- CNFs must define their pod affinity/anti-affinity rules
- Instantiation of CNF (via Helm chart or Operators or otherwise) should result in a fully-functional CNF ready to serve traffic, without requiring any post-instantiation configuration of system parameters
- CNFs should implement service resilience at the application layer and not rely on individual compute availability/stability
- CNFs should decouple application configuration from Pods, to allow dynamic configuration updates
- CNFs should support elasticity with dynamic scale up/down using kubernetes-native constructs, such as ReplicaSets.
- CNFs should support canary upgrades employing the platform Service Mesh
- CNFs should self-recover from common failures like pod failure, host failure, and network failure. Kubernetes-native mechanisms, such as health-checks (Liveness, Readiness and Startup Probes) should be employed at a minimum.

5.3. Platform Restrictions

- CNFs may not deploy Nodeports
- CNFs may not use host networking
- Namespace creation is performed by the platform team and may not be created by the CNFs deployment method (Helm / Operator)
- CNFs may not perform Role creation

- CNFs may not perform Rolebinding creation
- CNFs may not have Cluster Roles
- CNFs are not authorized to bring their own CNI
- CNFs may not deploy Daemonsets
- CNFs may not modify the platform in any way

Chapter 6. OpenShift Platform

By default, OpenShift networking utilizes the host "baremetal" network for ingress/egress of the cluster.

[Certified Operator Build Guide](#) - Guidelines on how to build an Operator that meets the Red Hat certification criteria.

[Partner Guide for OpenShift and Container Certification](#) - Step-by-step instructions for partners on how to certify their images and operators.

[OpenShift Operator Badge Guide](#) - Containers and Operators tests.

Chapter 7. Software Core/Edge

The deployment consists of the CaaS (Container as a Service) and PaaS (Platform as a Service) functions. Details of the Stack components are covered in the subsequent sections. The Stack components are subject to change over the next 12 months as further evaluation and testing is conducted in the HQP Lab with select 5G CNFs.

7.1. Helm v3

Helm v3 is a serverless mechanism for defining templates that describe a complete kubernetes application. This allows a template to be built for an application and site/deployment-specific values to be provided as input to the template when being pushed to a cluster, such that different configurations can be made in different locations. It is roughly analogous to HEAT templates in the OpenStack environment.

For more information, see [Getting started with helm on OpenShift](#).

7.2. Kubernetes

Kubernetes is an open source container orchestration suite of software that is API- driven with a datastore keeping state of the deployments resident on the cluster.

The Kubernetes API is the mechanism by which applications and people and applications interact with the cluster. There are several ways to do this, including via the kubectl or oc CLI tools, via web based UIs or interacting directly with the API using tools such as curl, or the SDKs can be used to build your own tools.

When interacting with the API, this can be done in at least one of two ways. If the application, or person is external to the cluster, the APIs can be accessed externally. If the application or person is on the cluster, or inside the cluster, one can access the cluster by hitting the Kubernetes Service Resource directly, thereby bypassing the need to exit the cluster and come back in.

7.3. CNI – OVN

OVN is the default pod network CNI plugin for OpenShift and is supported directly by Red Hat. OVN is Red Hat's CNI for pods. It is a Geneve-based overlay that requires L3 reachability between the host nodes. This L3 reachability can be over L2 or a pre-existing overlay network. OpenShift's OVN forwarding is based on flow rules and implemented with nftables on the host

OS CNI POD.

7.4. Container storage (CSI)

Pod Volumes are supported, via local storage and the CSI, for persistent volumes. Local storage is truly ephemeral storage, it is local only to the physical node that a pod is running on, and is lost in the event that a pod is killed and recreated. If a pod requires persistent storage the CSI can be used via kubernetes native primitives, persistentVolume (PV) and persistentVolumeClaim (PVC), to get persistent storage, such as an NFS share. For example, via the CSI backed by NetApp Trident.

When using storage with Kubernetes, you can leverage storage classes. These allow you to classify different storage by capabilities. For example, storage backed by fast SSD may be assigned to a different class than that backed by rotational disk. Volumes can then be requested based on the parameters of the storage they wish to use.

Network Functions clear persistent storage by deleting their PVs when removing their application from a cluster.

For more information, see [Persistent Storage](#).

7.5. Block storage

OpenShift Container Platform can provision raw block volumes. These volumes do not have a file system, and can provide performance benefits for applications that either write to the disk directly or implement their own storage service.

For more information, see [Block volume storage support](#).

7.6. Object storage

Object storage may be located at core locations. Access to object storage may be possible via S3 and Swift API, accessed via HAProxy Load Balancer over HTTPS protocol. Clients accessing object storage may route via the CNI eth0 network through the load balancer, and across the WAN to the object storage endpoints they are assigned during onboarding.

7.7. Container Runtime

OpenShift uses CRI-O as a CRI interface for Kubernetes. CRI-O manages runC for container image execution. CRI-O is an open-source container engine that provides a stable, performant platform for running OCI compatible runtimes. CRI-O is developed, tested and released in tandem with Kubernetes major and minor releases. Images are OCI compliant. Images are recommended to be built using Red Hat's open Universal Base Image. For more information, see Universal Base Image in the following sections.

For more information, see [CRI-O and Read-only containers](#) and [CRI-O](#).

This environment is maintained through the open source tools:

- runc
- skopeo
- buildah
- podman
- cri-o

7.8. CPU Manager / Pinning

The OpenShift Container Platform can use the Kubernetes CPU Manager to support CPU pinning for applications.

7.9. Host OS

OpenShift Container Platform will run Red Hat Enterprise Linux CoreOS (RHCOS) in a bare metal environment. There is no hypervisor layer between the containers and the host OS. RHCOS is the next generation container operating system. RCHOS is part of the OpenShift Container Platform and is used as the OS for the Control plane, and can be the default for worker nodes. RHCOS is based on RHEL, has some immutability, leverages the CRI-O runtime, contains container tools, and is updated through the Machine Config Operator (MCO).

The controlled immutability nature of RHCOS does not support installing RPMs or additional packages in the traditional way. Some 3rd party services or functionalities need to run as agents on nodes of the cluster.

For more information, see [Red Hat Enterprise Linux CoreOS | Architecture | OpenShift Container Platform 4.7](#).

Chapter 8. PaaS Core/Edge

8.1. Certificate Management

Certificate Management may be obtained through the platform via the network. A CSR may be generated to get a certificate signed via Citadel which should provide an ICA certificate. Platform certificate rotation on behalf of an application may happen.

8.2. Distributed Tracing

Distributed L7 tracing may be supported by the platform via a Service Mesh with Jaeger as the UI to the trace data.

8.3. Pod Security

SELinux is always enabled within the OpenShift Container Platform and is used to enforce syscalls that containers make. In addition, Kubernetes has another native function called pod security policies.

8.4. Load Balancer/Service Proxy

The default OpenShift load balancer, metalLB, can be used or any certified load balancer on Red Hat Openshift can be used. For example, F5 SPK or F5 BigIP. Applications must use the Load Balancer to get traffic into and out of the pod network.

8.5. CI/CD Framework

Applications should target a CI/CD approach for deployment and validation.

8.6. Kubernetes API Versions

The OpenShift Container Platform supports the full Kubernetes APIs, as well as additional API calls that are OpenShift specific.

For more information, see [Rest API](#).

Chapter 9. Pod Permissions

The default permissions of the platform should not permit pods to run as root. Pod restrictions are enforced by SCC within the OpenShift platform. For more information, see [Managing Security Context Constraints](#).

Pods execute on worker nodes by default, and admitted to the cluster with the "restricted" SCC. The "restricted" SCC:

- Ensures that no containers within the pod can run with the `allowPrivilegedContainer` flag set.
- Ensures that pods cannot mount host directory volumes.
- Requires that a pod run as a user in a pre-allocated range of UIDs from the namespace annotation.
- Requires that a pod run with a pre-allocated MCS label.
- Allows pods to use any supplemental group.

Any pods requiring elevated privileges must document the required capabilities driven by application syscalls and an exception process to validate the requirements must occur. Upon approval of an exception, a custom SCC can be created with the specific permissions needed, and made available to the CNF.

Chapter 10. OpenShift Best Practices

10.1. Logging

The OpenShift Container Platform supports logging from containers and forwards those logs separately from the platform logging to a centralized logging repository. Logs may be forwarded based on the Tenant Namespace identifier.

- Containers are expected to write logs to stdout
- Requires vendor to follow pod/container naming standards
- Logs are forwarded to a centralized storage location
- Logs can be parsed so that specific vendor logs can be sent back to the CNF, if required
- Requires vendor to provide svc/fqdn
- Logs may be sent back logs to the matching namespace using the below tag format:
 - `¬vendor-function-000.logs ¬` Logs for namespace 000
 - `¬vendor-function-001.logs ¬` Logs for namespace 001
 - pod in the tenant namespace for receiving these logs:
 - must write any logs to a traditional log file on PV (disk) handling log rotation itself (either by using a framework or the traditional logrotate pattern)
 - must not write any logs to default stdout/stderr container pipes to avoid getting back into the log stream (avoiding a feedback loop). In other words, that container must redirect stdout/stderr somewhere other than the default for that container

Log messages are aggregated as a JSON document after being normalized to add metadata. An example of a typical log message:

```

{
  "docker" : {
    "container_id" : "a2e6d10494f396a45e..."
  },
  "kubernetes" : {
    "container_name" : "rhel-logtest",
    "namespace_name" : "logflatx",
    "pod_name" : "rhel-logtest-987vr",
    "container_image" : "quay.io/openshift/ocp-logtest:latest",
    "container_image_id" : "docker.io/mffi....",
    "pod_id" : "67667d28-13fe-4c89-aa44-06936279c399",
    "host" : "ip-10-0-153-186.us-east-2.compute.internal",
    "labels" : {
      "run" : "rhel-logtest",
      "test" : "rhel-logtest"
    },
    "master_url" : "https://kubernetes.default.svc",
    "namespace_id" : "e8fb5826-94f7-48a6-ae92-354e4b779008"
  },
  "message" : "2020-03-03 11:44:51,996 - SVTLogger - INFO",
  "level" : "unknown",
  "hostname" : "ip-10-0-153-186.us-east-2.compute.internal",
  "pipeline_metadata" : {
    "collector" : {
      "ipaddr4" : "10.0.153.186",
      "inputname" : "fluent-plugin-systemd",
      "name" : "fluentd",
      "received_at" : "2020-03-03T11:44:52.189331+00:00",
      "version" : "1.7.4 1.6.0"
    }
  },
  "@timestamp" : "2020-03-03T11:44:51.996384+00:00"
}

```

10.2. Monitoring

Network Functions are expected to bring their own metrics collection functions, for example, Prometheus, for their application-specific metrics. This metrics collector is not expected to, nor be able to, poll platform-level metric data. Network Functions may support exposing their Prometheus collection functions via PromQL interfaces to existing OSS systems.

Control Plane (infrastructure) metrics are collected by the platform in a separate Prometheus instance.

10.3. CPU Allocation

It is important to note that when the OpenShift scheduler is placing pods, it first reviews the Pod CPU “Request” and schedules it if there is a node that meets the requirements. It then imposes the CPU “Limits” to ensure the pod does not consume more than the intended allocation. The limit can never be lower than the request.

10.3.1. NUMA Configuration

OpenShift provides a topology manager which leverages the CPU manager and Device manager to help associate processes to CPUs. The topology manager handles NUMA affinity. This feature is available as of OpenShift 4.6.

For more information, and some examples on how to leverage the topology manager, and create workloads that work in real time, see [Using Topology Manager](#) and [Creating a workload that works in real-time](#).

10.4. Memory Allocation

Regarding memory allocation, there are a couple of considerations.

1. How much of the platform is OpenShift itself using? and
2. How much is left over to allocate for the applications running on OpenShift?

Once it has been determined how much memory is left over for the applications, quotas can be applied which specify both the requested amount of memory and the limits. In the case of where a memory request has been specified, OpenShift does not schedule the pod unless the amount of memory required to launch it is available.

In the case of a limit being specified, OpenShift does not allocate more memory to the application than the limit provides. It is important to note that when the OpenShift scheduler is placing pods, it first reviews the Pod memory “Request” and schedules it if there is a node that meets the requirements. It then imposes the memory “Limits” to ensure the pod does not consume more than the intended allocation. The limit can never be lower than the request.

Vendors must supply quotas per project so that nodes can be sized appropriately and clusters are able to support the needs of vendor applications. For more information, see [Resource quotas per project](#)

10.5. Affinity / Anti-affinity

With OpenShift Container Platform, pod affinity and pod anti-affinity allow you to constrain which nodes your pod is eligible to be scheduled on based on the key/value labels on other pods. There are two types of affinity rules, required and preferred. Required rules must be met, whereas preferred rules are best effort.

These pod affinity / anti-affinity rules are set in the pod specification as matchExpressions to a labelSelector. See the following link for examples and more information. See the following example for more information here:

```
apiVersion: v1
kind: Pod
metadata:
  name: with-pod-affinity
spec:
  affinity:
    podAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        - labelSelector:
            matchExpressions:
              - key: security
                operator: In
                values:
                  - S1
          topologyKey: failure-domain.beta.kubernetes.io/zone
  containers:
    - name: with-pod-affinity
  image: quay.io/ocpqe/hello-pod
```

For more information, see [Nodes scheduler pod affinity](#).

10.6. Taints and Tolerations

Taints and tolerations allow the Node to control which Pods are (or are not) scheduled on them. A taint allows a node to refuse a pod to be scheduled, unless that pod has a matching toleration.

You apply taints to a node through the node specification (NodeSpec) and apply tolerations to a pod through the pod specification (PodSpec). A taint on a node instructs the node to repel all pods that do not tolerate the taint.

Taints and tolerations consist of a key, value, and effect. An operator allows you to leave one of

these parameters empty.

It is possible to utilize taints and tolerations to allow pods to be rescheduled and moved from nodes that are in need of maintenance. Pods may be forcibly ejected from nodes to perform necessary maintenance. Do not apply tolerations for NoExecute, PreferNoSchedule, and NoSchedule.

For more information, see [Controlling pod placement using node taints](#).

10.7. Requests / Limits

Requests and limits provide a way for a CNF developer to ensure they have adequate resources available to run the application. Requests can be made for storage, memory, CPU and so on. These requests and limits can be enforced by quotas. The production platform may utilize quotas as a way to enforce requests and limits.

For more information, see [Resource quotas per project](#).

It is possible to overcommit node resources in development environments. Keep in mind though, that a node can be overcommitted which can affect the strategy of request / limit implementation. For example, when you need guaranteed capacity, use quotas to enforce and in a development environment, you can overcommit where a trade-off of guaranteed performance for capacity is acceptable. Overcommitment can be done on a project, node or cluster level.

For more information, see [Configuring your cluster to place pods on overcommitted nodes](#).

10.8. Pods

10.8.1. No naked pods

Do not use naked pods, that is, pods not bound to a ReplicaSet or Deployment. Naked pods are not rescheduled in the event of a node failure.

10.8.2. Image tagging

An image tag is a label applied to a container image in a repository that distinguishes a specific image from other images. Image tags may be used to categorize images as latest, stable, development and by versions within those two categories. This allows the administrator to be specific when declaring which image to test, or which image to run in production.

For more information, see [Tagging images](#).

10.8.3. One process per container

OpenShift organizes workloads into pods. Pods are the smallest unit of a workload that Kubernetes understands. Within pods, one can have one or more containers. Containers are essentially composed of the runtime that is required to launch and run a process.

Each container should run only one process. Different processes should always be split between containers, and where possible also separated into different pods. This can help in a number of ways, such as troubleshooting, upgrades, and more efficient scaling.

However, OpenShift does support running multiple containers per pod. This can be useful if parts of the application need to share namespaces, like networking and storage resources. Additionally, there are other models like launching init containers and sidecar containers, which may justify running multiple containers in a single pod.

Applications that utilize service mesh have an additional container injected into their pods to proxy workload traffic.

For more information, see [Using pods](#).

10.8.4. init containers

Init containers can be used for running tools / commands / or any other action that needs to be done before the actual pod is started. For example, loading a database schema or constructing a config file from a definition passed in via configMap or secret.

For more information, see [Using Init Containers to perform tasks before a pod is deployed](#).

10.9. Security / RBAC

Roles / RolesBinding - A role represents a set of permissions within a particular namespace. For example, a given user can list pods/services within the namespace. The RoleBinding is used for granting the permissions defined in a role to a user or group of users.

ClusterRole / ClusterRoleBinding - A ClusterRole represents a set of permissions at the Cluster level. For example, a given user has 'cluster-admin' privileges admin on the cluster. The ClusterRoleBinding is used for granting the permissions defined in a ClusterRole to a user or

group of users.

For more information, see [Using RBAC to define and apply permissions](#).

10.10. Multus

Multus is a meta CNI that allows multiple CNIs that it delegates to. This allows pods to get additional interfaces beyond eth0 via additional CNIs. The solution may have additional CNIs for SR-IOV and MacVLAN interfaces. This would allow for direct routing of traffic to a pod without using the pod network via additional interfaces. This capability is being delivered for use in only corner case scenarios. It is not to be used in general for all applications.

Example use cases include, bandwidth requirements that necessitate SR-IOV, and protocols that are unable to be supported by the load balancer. The OVN-based pod network is used for every interface that can be supported from a technical standpoint.

For more information, see [Understanding multiple networks](#).

10.10.1. Multus SR-IOV / MACVLAN

SR-IOV is a specification that allows a PCIe device to appear to be multiple, separate physical PCIe devices. The Performance Addon component allows you to validate SR-IOV by running DPDK, SCTP, and device checking tests.

SR-IOV and MACVLAN interfaces are able to be requested for protocols that do not work with the default CNI, or for exceptions where a network function has not been able to move functionality onto the CNI. These are exceptional use cases. Multus interfaces are defined by the platform operations team for the network functions which can then consume them. Using the planning tools, multus interfaces have to be requested ahead of time by the company's personnel. VLANs are applied by the SR-IOV VF, thus the VLAN/network that the SR-IOV interface requires must be part of the request for the namespace.

For more information, see [About Single Root I/O Virtualization \(SR-IOV\) hardware networks](#)

By configuring the SR-IOV Network CRs named NetworkAttachmentDefinitions are exposed by the SR-IOV Operator in the CNF namespace. Different names are assigned to different Network Attachment Definitions that are namespace-specific. MACVLAN versus Multus interfaces are named differently to distinguish the type of device assigned to them. These are created by configuring SR-IOV devices using the SriovNetworkNodePolicy CR.

From the CNF perspective, a defined set of network attachment definitions are available in the assigned namespace to serve secondary networks for regular usage, or to serve for DPDK payloads.

The SR-IOV devices are configured by the cluster admin, and they are available in the namespace assigned to the CNF.

The command "oc -n <cnfnamespace> get network-attachment-definitions" will return the list of secondary networks available in the namespace.

10.10.2. SR-IOV Interface Settings

The following settings must be negotiated with the cluster administrator for each network type available in the namespace:

- The type of netdevice to be used for the VF (kernel or userspace)
- The vlan ID to be applied to a given set of VFs available in a namespace
- For kernel-space devices, the ip allocation is provided directly by the cluster ip assignment mechanism
- The option to configure the ip of a given SR-IOV interface at runtime. For more information, see [Runtime configuration for an Ethernet-based SR-IOV attachment](#).

Example sriovnetworknodepolicy:

NOTE	This is enabled by the cluster administrator.
-------------	---

```

apiVersion: sriovnetwork.openshift.io/v1
kind: SriovNetworkNodePolicy
metadata:
  name: nnp-wlens3f0grp2
  namespace: openshift-sriov-network-operator
spec:
  deviceType: vfio-pci
  isRdma: false
  linkType: eth
  mtu: 9000
  nicSelector:
    deviceID: 158b
    pfNames:
      - ens3f0#50-63
    vendor: "8086"
  nodeSelector:
    kubernetes.io/hostname: worker-3
  numVfs: 64
  priority: 99
  resourceName: wlens3f0grp2

```

Example 1: Empty IPAM

NOTE

The sriovnetwork CR creates the network-attach-definition within the target networkNamespace.

```

apiVersion: sriovnetwork.openshift.io/v1
kind: SriovNetwork
metadata:
  name: sriovnet
  namespace: openshift-sriov-network-operator
spec:
  capabilities: '{ "mac": true }'
  ipam: '{}'
  networkNamespace: <CNF-NAMESPACE>
  resourceName: wlens3f0grp2
  spoofChk: "off"
  trust: "on"
  vlan: 282

```

Example 2: Whereabouts IPAM

```

apiVersion: sriovnetwork.openshift.io/v1
kind: SriovNetwork
metadata:
  name: sriovnet
  namespace: openshift-sriov-network-operator
spec:
  capabilities: '{ "mac": true }'
  ipam:

'{"type": "whereabouts", "range": "FD97:0EF5:45A5:4000:00D0:0403:0000:00
01/64", "range_start": "FD97:0EF5:45A5:4000:00D0:0403:0000:0001", "range
_end": "FD97:0EF5:45A5:4000:00D0:0403:0000:0020", "routes": [{ "dst": "fd9
7:0ef5:45a5::/48", "gw": "FD97:EF5:45A5:4000::1" } ]}'
  networkNamespace: <CNF-NAMESPACE>
  resourceName: wlens3f0grp2
  spoofChk: "off"
  trust: "on"
  vlan: 282

```

Example 3: Static IPAM

```

apiVersion: sriovnetwork.openshift.io/v1
kind: SriovNetwork
metadata:
  name: sriovnet
  namespace: openshift-sriov-network-operator
spec:
  capabilities: '{ "mac": true }'
  ipam: '{ "type":
"static", "addresses": [{ "address": "10.120.26.5/25", "gateway": "10.120.2
6.1" } ] }'
  networkNamespace: <CNF-NAMESPACE>
  resourceName: wlens3f0grp2
  spoofChk: "off"
  trust: "on"
  vlan: 282

```

Example 4: Using Pod Annotations to attach

```

apiVersion: v1
kind: Pod
metadata:
  name: sample-pod
  annotations:
    k8s.v1.cni.cncf.io/networks: |-
    [
      {
        "name": "net1",
        "mac": "20:04:0f:f1:88:01",
        "ips": ["192.168.10.1/24", "2001::1/64"]
      }
    ]

```

The examples depict scenarios used within the Core solution to deliver secondary network interfaces, with and without IPAM, to a pod.

- Example 1 creates a network attachment definition that does not specify an IP address.
- Example 2 makes use of the static IPAM, and
- Example 3 makes use of the whereabouts CNI that provides a cluster wide DHCP option.

The actual addresses used for static IPAM and whereabouts are managed external to the cluster.

The SrioVnetwork CR will configure a network attachment definition within the CNF's namespace.

```
[c]$ oc get net-attach-def -n <CNF-NAMESPACE>
```

NAME	AGE
sriovnet	9d

Within the CNF namespace, the sriov resource is consumed via a pod annotation:

```

kind: Pod
metadata:
  annotations:
    k8s.v1.cni.cncf.io/networks: sriovnet

```

10.10.3. Attaching the VF to a pod

Once the right network attachment definition is found, applying the `k8s.v1.cni.cncf.io/networks` annotation with the name of the network attachment

definition to the pod will add the additional network interfaces in the pod namespace.

Example:

```
apiVersion: v1
kind: Pod
metadata:
  name: sample-pod
  annotations:
    k8s.v1.cni.cncf.io/networks: |-
    [
      {
        "name": "net1",
        "mac": "20:04:0f:f1:88:01",
        "ips": ["192.168.10.1/24", "2001::1/64"]
      }
    ]
```

10.10.4. Discovering SR-IOV devices properties from the application

All the properties of the interfaces are added to the pod's

`k8s.v1.cni.cncf.io/network-status` annotation. The annotation is json-formatted and for each network object contains information, such as ips (where available), mac address, and pci address.

Example:

```
k8s.v1.cni.cncf.io/network-status: |-
[ {
  "name": "",
  "interface": "eth0",
  "ips": [
    "10.132.3.148"
  ],
  "mac": "0a:58:0a:84:03:94",
  "default": true,
  "dns": {}
}, {
  "name": "cnfns/networkname",
  "interface": "net1",
  "ips": [
    "1.1.1.2"
  ],
  "mac": "ba:1d:e7:31:2a:e0",
  "dns": {},
  "device-info": {
    "type": "pci",
    "version": "1.0.0",
    "pci": {
      "pci-address": "0000:19:00.5"
    }
  }
}]
```

NOTE The ip information is not available if the driver specified is vf-io.

The same annotation is available as file content inside the pod, at the `/etc/podnetinfo/annotations` path. For convenience, a library is available to easily consume this information from the application (bindings in C and Go). For more information, see [DPDK library for use with container applications](#).

10.10.5. NUMA Awareness

If the pod is using a guaranteed QoS class, and the kubelet is configured with a suitable topology manager policy (restricted, single-numa node), then the VF assigned to the pod will belong to the same NUMA node as the other assigned resources (CPU and other NUMA aware devices).

For more information, see [What huge pages do and how they are consumed by applications](#).

10.11. Upgrades

10.11.1. Handling platform upgrades

- CNF vendors should expect that the platform may be upgraded to new versions on an ongoing basis employing CI/CD runtime deployment, without any advance notice to CNF vendors.
- During platform upgrades, the Kubernetes API deprecation policy defined in the [Kubernetes Deprecation Policy](#) must be followed.
- CNFs are expected to maintain service continuity during Platform Upgrades, and during CNF version upgrades.
- CNFs need to be prepared for nodes to reboot, or shut down without notice.
- CNFs configure pod disruption budget appropriately to maintain service continuity during platform upgrades.
- Applications **may not** deploy pod disruption budgets that prevent zero pod disruption.
- Applications must not be tied to a specific version of Kubernetes, or any of its components.

10.12. OpenShift Virtualization / kubevirt

10.12.1. Openshift Virtualization and VMs (CNV) best practices

The platform was designed as a pure container-based system, where all network functions are containerized. However, it has become apparent that some NFs have not completed re-architecting all components of their network functions to be fully containerized. To deal with this lag, VMs are orchestrated via Kubernetes for an interim period of time for applications that require low latency connectivity between containers and these VMs. When OpenShift Virtualization becomes generally-available for enterprise workloads, such throughput- and latency-insensitive workloads, may be added to the cluster. VNFs and other throughput- or latency-sensitive applications can be considered only after careful validation. Until then, it is recommended to keep these workloads on OSP VMs.

OpenShift virtualization must be installed according to its [About container-native virtualization documentation](#), and only documented supported features may be used unless an explicit exception has been granted. For more information, see [About OpenShift Virtualization](#).

To improve overall virtualization performance and reduce CPU latency, critical VNFs can take advantage of OpenShift Virtualization's high-performance features. These can provide the VNFs with dedicated CPU resources and "isolate" QEMU threads, such as the emulator and the IO threads, on a separate physical CPU so that it does not affect the workloads CPU latency. For more information, see:

- [Enabling dedicated resources for virtual machines](#),
- [Requesting dedicated cpu for QEMU emulator](#),
- [IO threads with QEMU emulator thread and dedicated pinned CPUs](#)

Similar to OpenStack, OpenShift Virtualization supports the device role tagging mechanism for the network interfaces (with the same format as it is in OSP). Users are able to tag Network interfaces in the API and identify them in device metadata provided to the guest OS via the config drive. For more information, see [Device Role Tagging](#).

10.12.2. VM image Import Recommendations (CDI)

OpenShift Virtualization VMs store their persistent disks on kubernetes Persistent Volumes (PVs). PVs are requested by VMs using kubernetes Persistent Volume Claims (PVCs). VMs may require a combination of blank and pre-populated disks to function. Blank disks can be initialized automatically by KubeVirt when an empty PV is initially encountered by a starting VM. Other disks must be populated prior to starting the VM. OpenShift Virtualization provides a component called the Containerized Data Importer (CDI) which automates the preparation of pre-populated persistent disks for VMs. CDI integrates with KubeVirt to synchronize VM creation and deletion with disk preparation by using a custom resource called a DataVolume. Using DataVolumes, data can be imported into a PV from various sources including container registries and HTTP servers.

The following recommendations must be followed when managing persistent disks for VMs:

- Blank disks: Create a PVC and associate it with the VM using a `persistentVolumeClaim` volume type in the `volumes` section of the `VirtualMachine` spec.
- Populated disks: In the `VirtualMachine` spec, add a `DataVolume` to the `dataVolumeTemplates` section and always use the `dataVolume` volume type in the `volumes` section.

Working with large VM disk images

In contrast to container images, VM disk images can be quite large (30GiB or more is common). It is important to consider the costs of transferring large amounts of data when planning

workflows involving the creation of VMs (especially when scaling up the number of VMs). The efficiency of an image import depends on the format of the file and also the transfer method used. The most efficient workflow, for two reasons, is to host a gzip-compressed raw image on a server and import via HTTP. Compression avoids transferring zeros present in the free space of the image, and CDI can stream the contents directly into the target PV without any intermediate conversion steps. In contrast, images imported from a container registry must be transferred, unarchived, and converted prior to being usable. These additional steps increase the amount of data transferred between a node and the remote storage.

Chapter 11. Operator Best Practices

To learn more details for OLM and SDK projects, including best practices, common recommendations, suggestions and conventions, see [Operator Best Practices](#).

Chapter 12. Container Best Practices

12.1. Pod Exit Status

The most basic requirements for the lifecycle management of pods in OpenShift are the abilities to start and stop correctly. When starting up, health probes like liveness and readiness checks can be put into place to ensure the application is functioning properly.

There are two different ways a pod can stop on Kubernetes. The first way is that the pod can remain alive but non-functional. In this case, if the administrator has implemented liveness and readiness checks, OpenShift can stop the pod, and either restart it on the same node or a different node in the cluster.

The second way is that the pod can crash and become non-functional. In this case, when the application in the pod stops, it exits with a code, and writes suitable log entries to help the administrator diagnose the issue that caused the problem.

Pods use `terminationMessagePolicy: FallbackToLogsOnError` to summarize why they crashed, and use `stderr` to report errors on the crash.

12.2. Graceful Termination

There are different reasons that a pod may need to shutdown on an OpenShift cluster. It might be that the node the pod is running on needs to be shut down for maintenance, or the administrator is doing a rolling update of an application to a new version which requires that the old versions are shutdown properly.

When pods are shut down by the platform they are sent a `SIGTERM` signal. This means that the process in the container starts shutting down, closing connections, and stopping all activity. If the pod does not shut down within the default 30 seconds, then the platform may send a `SIGKILL` signal which stops the pod immediately. This method is not as clean and the default time between the `SIGTERM` and `SIGKILL` messages can be modified based on the requirements of the application.

Pods exit with zero exit codes when they are gracefully terminated.

12.3. Pod Resource Profiles

OpenShift comes with a default scheduler that is responsible for being aware of the current available resources on the platform, and placing containers / applications on the platform appropriately. For OpenShift to do this correctly, the application developer must create a resource profile for the application. This resource profile contains requirements, such as how much memory, CPU, and storage that the application needs. At that point, the scheduler is aware of the nodes in the cluster that can satisfy the workload, and place the application on one of those nodes (or distribute it). Or the scheduler places the pod that the application is in in a pending state until resources come available.

All pods have a resource request that is the minimum amount of resources the pod is expected to use at steady state for both memory and CPU.

12.4. Storage: emptyDir

There are several options for volumes and reading and writing files in OpenShift. When the requirement is for temporary storage, and given the option to write files into directories in containers versus an external filesystems, choose the `emptyDir` option. This provides the administrator with the same temporary filesystem, so when the pod is stopped the dir is deleted forever. In addition, the `emptyDir` can be backed by whatever medium is backing the node, or it can be set to memory for faster reads and writes.

Using `emptyDir` with requested local storage, which limits instead of writing to the container directories, will also allow enabling `readOnlyRootFilesystem` on the container or pod.

12.5. Liveness and Readiness Probes

As part of the pod lifecycle, the OpenShift platform needs to know the status of the pod at all times. This can be accomplished with different health checks. There are at least three states that are important to the platform, these are, startup, running, shutdown. Applications can also be running, but not healthy, meaning, the pod is up and the application shows no errors, but it cannot serve any requests.

When an application starts up on OpenShift it may take a while for the application to become ready to accept connections from clients, or perform whatever duty it is intended for.

Two health checks that are required to monitor the status of the applications are liveness and

readiness. As mentioned above, the application may be running but not actually able to serve requests. This can be detected with liveness checks. The liveness checks send-specific requests to the application that, if satisfied, indicate that the pod is in a healthy state, and operating within the required parameters that the administrator has set. A failed liveness check results in the container being restarted.

There is also a state of a pod at startup. Here the pod may start and take a while for different reasons. Pods can be marked as ready if they pass the readiness check. The readiness check determines that the pod has started properly and is able to answer requests. There are circumstances where both checks are used to monitor the applications in the pods. A failed readiness check results in the container being taken out of the available service endpoints. An example of this being relevant is when the pod was under heavy load, failed the readiness check, gets taken out of the endpoint pool, processes requests, passes the readiness check, and is added back to the endpoint pool.

12.6. Use `imagePullPolicy: IfNotPresent`

If there is a situation where the container dies and needs to be restarted, the image pull policy becomes important. There are three image pull policies available: `Always`, `Never` and `ifNotPresent`. It is generally recommended to have a pull policy of `ifNotPresent`. This means that if a pod needs to restart for any reason, the kubelet will check on the node where the pod is starting, and reuse the already downloaded container image if it's available. OpenShift intentionally does not set `AlwaysPullImages` as turning on this admission plugin can introduce new kinds of cluster failure modes. Self-hosted infrastructure components are still pods. Enabling this feature can result in cases where a loss of contact to an image registry can cause redeployment of an infrastructure or application pod to fail. We use `PullIfNotPresent` so that a loss of image registry access does not prevent the pod from restarting.

It is noted that any container images, protected by registry authentication, have a condition whereby a user who is unable to download an image directly can still launch it by leveraging the host's cached image.

12.7. Automount Services for Pods

Set `automountServiceAccountToken: false` on all pods, unless the containers need to access the Kubernetes API.

12.8. Disruption budgets

When managing the platform, there are at least two types of disruptions that can occur. They are voluntary and involuntary.

When dealing with voluntary disruptions, a pod disruption budget can be set that determines how many replicas of the application must remain running at any given time. For example, consider the case where an administrator is shutting down a node for maintenance and the node has to be drained.

If a pod disruption budget is set, then OpenShift respects that and ensures that the required number of pods are available by bringing up pods on different nodes, before draining the current node.

Chapter 13. Networking Overview

OpenShift is a multi-tenant environment. NFs are deployed within a single namespace. Supporting applications, like an OAM platform for multiple NFs from the same vendor, must be run in an additional separate namespace.

Multus may be supported within the platform for additional NICs within containers. However, Multus should be used only for those cases that cannot be supported, for example, by a F5 load balancer.

The POD and Services networks may have an unrouted address spaces and are only reachable via service VIPs on the load balancers. The POD network may be NATed as traffic egresses the load balancer. Inbound traffic is destination-NATed to Service/Pod IP addresses.

Applications use Network Policies for firewalling the application. Network Policies must be written with a default deny, and only allow ports and protocols on an as needed basis for any pods and services.

13.1. OVN-kubernetes CNI

OVN is Red Hat's CNI for pod networking. It is a Geneve-based overlay that requires L3 reachability between the host nodes. This L3 reachability can be over L2 or a pre-existing overlay network. OpenShift's OVN forwarding is based on flow rules and implemented with nftables on the host OS CNI POD.

Chapter 14. User Plane Functions

14.1. Performance Addon Operator

Red Hat created the Performance Addon Operator for low latency nodes. The emergence of Edge computing in the area of Telco / 5G plays a key role in reducing latency and congestion problems, and improving application performance. Many of the deployed applications in the Telco space require low latency that can only tolerate zero packet loss. OpenShift Container Platform provides a Performance Addon Operator (PAO) to implement automatic tuning to achieve low latency performance for applications. The PAO is a meta operator that leverages MachineConfig, Topology Manager, CPU Manager, Tuned, and KubeletConfig to optimize the nodes.

The PAO enables:

- Hugepages
- Dynamic CPU isolation
- NUMA Awareness

For more information, see [Performance Addon Operator for low latency nodes](#).

14.2. Hugepages

In the Openshift Container Platform, nodes/hosts must pre-allocate huge pages.

All workers within a cluster may have 32,000, 2M hugepages per NUMA node enabled as follows:

```
hugepages:
  defaultHugepagesSize: "2M"
  pages:
    - size: "2M"
      count: 32000
      node: 0
    - size: "2M"
      count: 32000
      node: 1
```

To request huge pages, pods must supply the following within the pod.spec for each container:

```
resources:
  limits:
    hugepages-2Mi: 100Mi
    memory: "1Gi"
    cpu: "1"
  requests:
    hugepages-2Mi: 100Mi
    memory: "1Gi"
    cpu: "1"
```

For more information, see [Configuring huge pages](#).

14.3. CPU Isolation

The Performance Addon Operator manages host CPUs by dividing them into reserved CPUs for cluster and operating system housekeeping duties, and isolated CPUs for workloads. CPUs that are used for low latency workloads are set as isolated.

Device interrupts are load balanced between all isolated and reserved CPUs to avoid CPUs being overloaded, with the exception of CPUs where there is a guaranteed pod running. Guaranteed pod CPUs are prevented from processing device interrupts when the relevant annotations are set for the pod.

Worker nodes may have the following CPU profile applied, reserving 2 Cores per socket for housekeeping (kernel) and the rest for workloads.

```
spec:
  cpu:
    isolated: 4-39,44-79
    reserved: 0-3,40-43
```

- isolated - Has the lowest latency. Processes in this group have no interruptions and so can, for example, reach much higher DPDK zero packet loss bandwidth.
- reserved - The housekeeping CPUs. Threads in the reserved group tend to be very busy, so latency-sensitive applications should be run in the isolated group

Default worker node `performanceprofile` that may be enabled as follows:

```

apiVersion: performance.openshift.io/v2
kind: PerformanceProfile
metadata:
  name: perf-profile-2m-worker
spec:
  cpu:
    isolated: 4-39,44-79
    reserved: 0-3,40-43
  hugepages:
    defaultHugepagesSize: "2M"
    pages:
      - size: "2M"
        count: 32000
        node: 0
      - size: "2M"
        count: 32000
        node: 1
  numa:
    topologyPolicy: best-effort
  realTimeKernel:
    enabled: false
  nodeSelector:
    node-role.kubernetes.io/workerperf: ""

```

The resulting KubeletConfig: (partial config shown below)

```

{
  "kind": "KubeletConfiguration",
  ...
  "cpuManagerPolicy": "static",
  "cpuManagerReconcilePeriod": "5s",
  "topologyManagerPolicy": "best-effort",
  ...
},
  "reservedSystemCPUs": "0-3,40-43",
}

```

Additionally, the performanceprofile creates a “runTimeClass” that pods must specify within the pod.spec in order to fully achieve CPU isolation for the workload.

```

oc describe performanceprofile perf-profile-2m-worker
Name:          perf-profile-2m-worker
Namespace:
Labels:        <none>
Annotations:   <none>
API Version:   performance.openshift.io/v2
Kind:          PerformanceProfile
Spec:
  Cpu:
    Isolated:  4-39,44-79
    Reserved:  0-3,40-43Container Best Practices
  Hugepages:
    Default Hugepages Size:  2M
  Pages:
    Count:  32000
    Node:    0
    Size:    2M
    Count:  32000
    Node:    1
    Size:    2M
  Node Selector:
    node-role.kubernetes.io/workerperf:
  Numa:
    Topology Policy:  best-effort
  Real Time Kernel:
    Enabled:  false
Status:
  Runtime Class:      performance-perf-profile-2m-worker
  Tuned:              openshift-cluster-node-tuning-
operator/openshift-node-performance-perf-profile-2m-worker

```

For workloads requiring CPU isolation in (OCP 4.7.11) the `pod.spec` must have the following:

- For each container within the pod, resource requests and limits must be identical (Guaranteed Quality of Service)
- Request and Limits are in the form of whole CPUs
- The `runtimeClassName` must be specified
- Annotations disabling CPU and IRQ load-balancing

An example `pod.spec` is as follows:

```

metadata:
  annotations:
    cpu-load-balancing.crio.io: "disable"
    irq-load-balancing.crio.io: "disable"
  name: pao-example-podspec
spec:
  containers:
  - image: <PATH-TO-IMAGE>
    name: test
    resources:
      limits:
        cpu: 1
        memory: 1Gi
        hugepages-2Mi: 1000Mi
      requests:
        cpu: 1
        memory: 1Gi
        hugepages-2Mi: 1000Mi
    restartPolicy: Always
    runtimeClassName: performance-perf-profile-2m-worker

```

14.4. NUMA Awareness

Topology Manager collects hints from the CPU Manager, Device Manager, and other Hint Providers to align pod resources, such as CPU, SR-IOV VFs, and other device resources, for all Quality of Service (QoS) classes on the same non-uniform memory access (NUMA) node. This topology information and the configured Topology Manager policy determine whether a workload is accepted or rejected on a node.

NOTE

To align CPU resources with other requested resources in a Pod spec, the CPU Manager must be enabled with the static CPU Manager policy.

The following Topology manager policies are available and dependent on the requirements of the workload can be enabled. For high performance workloads making use of SR-IOV VFs, NUMA awareness follows the NUMA node to which the SR-IOV capable network adapter is connected.

Best-effort policy

For each container in a pod with the best-effort topology management policy, kubelet calls each Hint Provider to discover their resource availability. Using this information, the Topology Manager stores the preferred NUMA Node affinity for that container. If the affinity is not preferred, Topology Manager stores this and admits the pod to the node.

Restricted policy

For each container in a pod with the restricted topology management policy, kubelet calls each Hint Provider to discover their resource availability. Using this information, the Topology Manager stores the preferred NUMA Node affinity for that container. If the affinity is not preferred, Topology Manager rejects this pod from the node, resulting in a pod in a Terminated state with a pod admission failure.

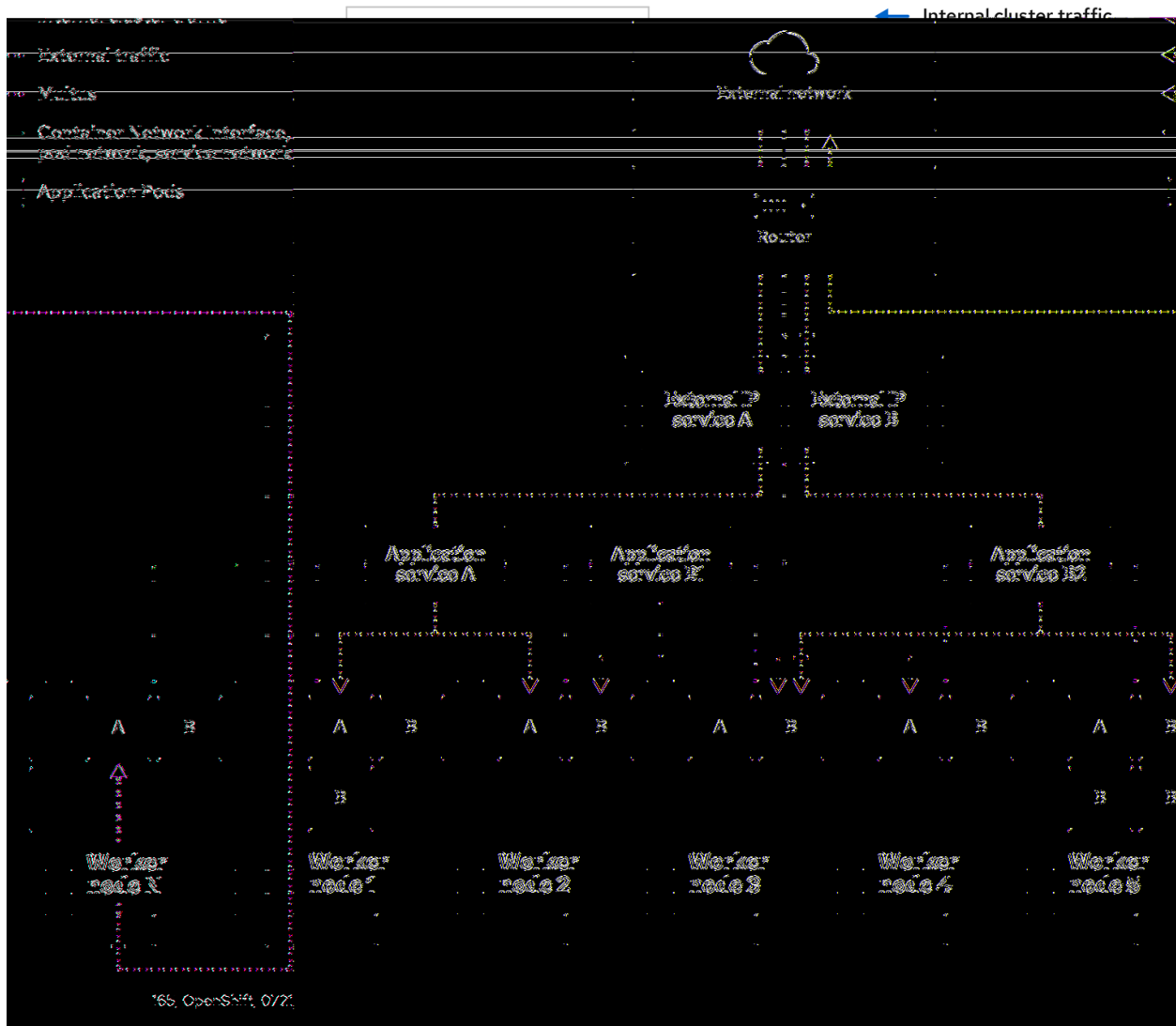
Single-numa-node policy

For each container in a pod with the single-numa-node topology management policy, kubelet calls each Hint Provider to discover their resource availability. Using this information, the Topology Manager determines if a single NUMA Node affinity is possible. If it is possible, the pod is admitted to the node. If it is not possible, the Topology Manager rejects the pod from the node. This results in a pod in a Terminated state with a pod admission failure.

For more information, see [Using Topology Manager](#).

Chapter 15. Application Service Exposure to External Networks

The following diagram depicts an overview of the exposure of application services to external networks.



The broken blue lines depict a pair of services that are exposed to the external networks. These services consist of a VIP on the load balancer or a L7 Ingress. These VIPs are backed by services at the Kubernetes layer that are then serviced by pods running in the platform. Each application can get different IP addresses that are specifically associated with their application. Additionally, if desired, multiple ports on the same VIP can listen for different functions and forward to different services, and thereby different pods within the application.

The second mechanism of service delivery (depicted by the solid blue line), is to make a service

available within the platform to either an application itself (internally to the application) or to other applications within the platform.

The third mechanism (depicted by the broken orange line), is the **LEAST PREFERRED** and requires a design exception, to expose a pod via Multus and additional interfaces beyond eth0 on that pod. It is also possible to have the load balancer act as a one-armed load balancer for this Multus IP.

Chapter 16. Service Mesh for Inter/Intra NF

16.1. Service Mesh Introduction

Kubernetes clusters contain a network with flat layer 3 IP connectivity between all pods, provided by a pluggable layer in Kubernetes called the Container Network Interface (CNI). In the basic deployment scenario, that means that any application container can communicate with any other application container. Any details of that communication (including plaintext or encryption, protocols, authentication, monitoring) are the responsibility of each application container.

Without intervention, each application is forced to implement all aspects of security requiring a high degree of effort and diligence. Much of this effort is replicated since each application must repeat it. As an example, even if all applications implement TLS properly, when a vulnerability is discovered in a TLS implementation, every individual application must have a separate vulnerability analysis executed, and patches made to software code and updated in the field.

A modern approach to securing traffic between Kubernetes pods (and accordingly, the containerized applications that run in them, such as CNFs and MEC apps), is a service mesh. The service mesh is a security-enhancing sidecar container that runs in each pod and proxies network traffic to or from the main application container. In its position as a proxy, the service mesh sidecar can provide security, resiliency, load balancing, and detailed measurement for the application container. More importantly, the sidecar provides a consistent implementation of each of these functions, regardless of the details of the application. For example, the sidecar has one implementation of TLS that is used for all sidecars across the entire Kubernetes cluster. If a vulnerability is found in this implementation, only one upgrade must be performed.

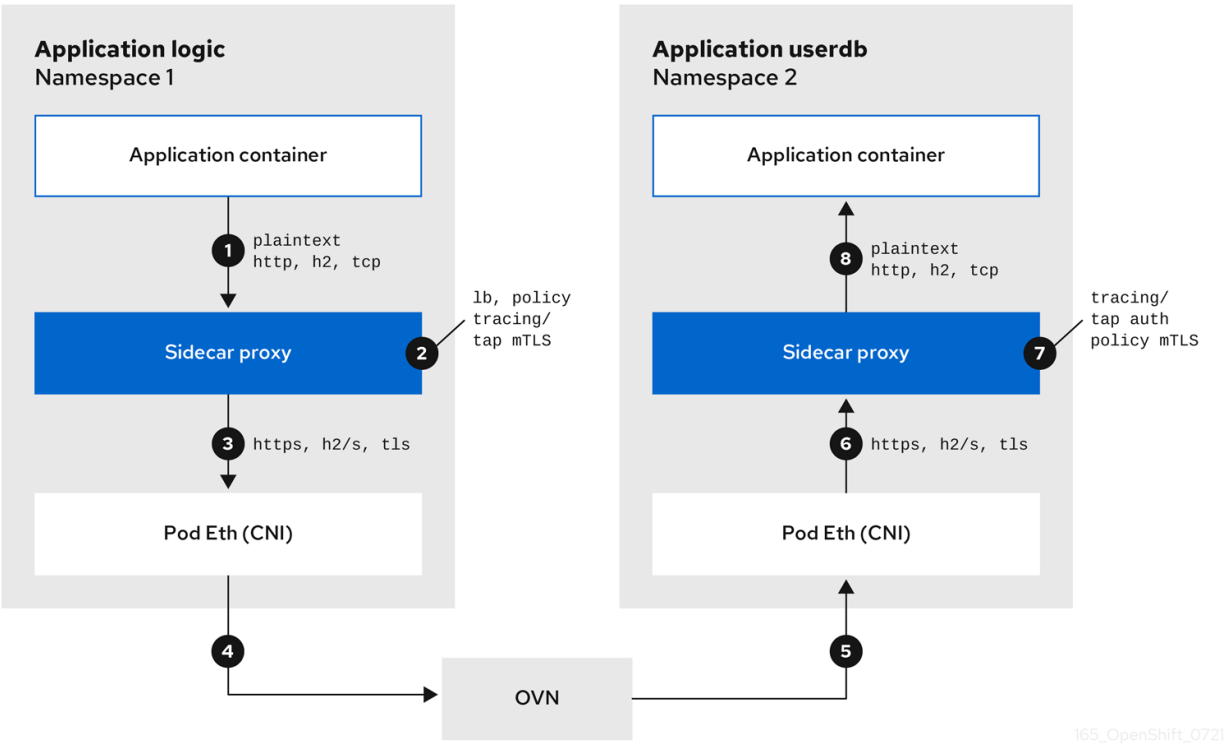
The sidecar is transparently injected into the Kubernetes pod without requiring any intervention by the main application container. Application containers do not need to directly communicate in any special way with the sidecar. Generally, application containers are unaware of the presence of the sidecar. They communicate "as normal" and the sidecar transparently proxies these communications, as long as they are allowed by policy. This document details the requirements so that the sidecar can correctly proxy application traffic and apply policy.

Service mesh is also capable of doing CSR generation, signing and installing into the namespace as part of the solution. This offloads all PKI efforts from the CNF, and also makes the certificates available for NFs within the application. The certificates are made available via Kubernetes secrets as a volume mount if an application requires the keys for doing any TLS via Multus

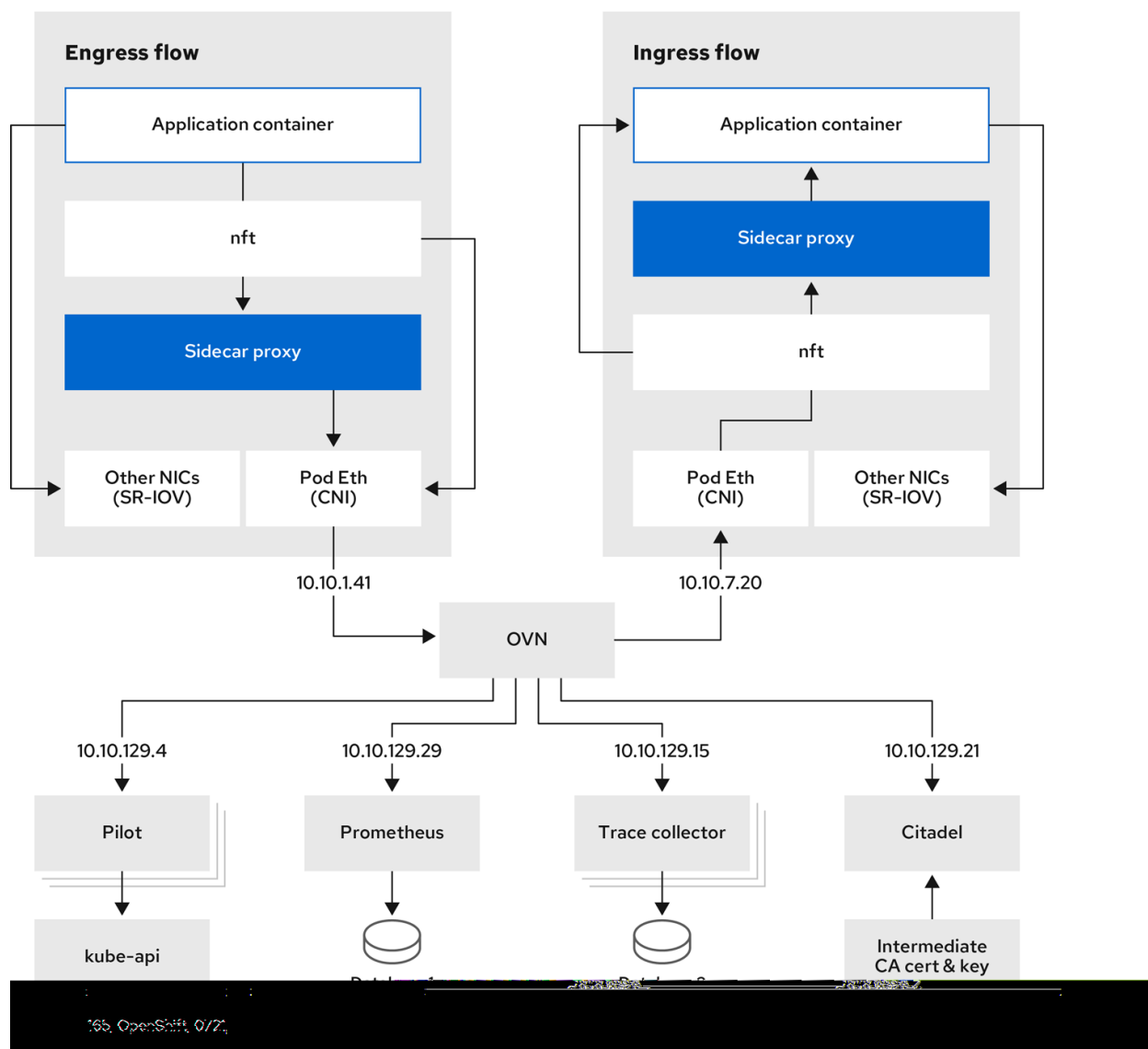
based interfaces.

Additionally, service mesh allows distributed tracing for HTTP based flows that can be analyzed via the Jaeger UI. This provides a consistent visibility mechanism across NFs for Service Based Interface (SBI) communication.

The following diagram depicts a high level overview of a service mesh implementation.



The following diagram depicts a detailed overview of a service mesh implementation.



It shows the full suite of components that are involved with service mesh delivery. Pilot is the Istio Control plane, the sidecar proxies are Envoy based proxies. Trace collector and Prometheus provide statistics and traces. Not depicted is Jaeger which allows viewing of traces.

16.2. Service Mesh Tapping

As service mesh's unique location related to the HTTP traffic for SBI interfaces, it is well positioned to provide a tapping solution. This solution feeds tools for doing traces on network traffic for the purposes of call traces, and evaluating overall network health via statistical analysis.

Service mesh is able to front end all SBI interfaces, and so provides the capacity for comprehensive and consistent visibility across all of the SBI interfaces within the 5G Core.

16.3. Service Mesh Requirements for CNF

1. The application **must** declare all listening ports as containerPorts in the pod specification it provides to Kubernetes.
 - a. The application **must not** listen on any other ports that are undeclared.
 - b. The service mesh **may** be configured to block connections to these ports.
 - c. These ports **must** be named in the pod specification with the protocol they implement.
 - i. The name field in the ContainerPort section must be of the form <protocol>[-<suffix>] where <protocol> is one of the below, and the optional <suffix> can be chosen by the application.
 - ii. Preferred prefixes: grpc , grpc-web , http , http2
 - iii. Fallback prefixes: tcp , udp
 - iv. Valid Example: http-webapi or grpc
2. The application **must** communicate with Kubernetes Services by their service IP instead of individually selecting pods in that service.
 - a. The service mesh selects the appropriate pod.
3. The application **must not** encrypt outbound traffic on the cluster network interface.
 - a. The service mesh applies policy, authenticates servers and encrypts outbound traffic before it leaves the application pod.
4. The application **must not** decrypt inbound traffic on the cluster network interface.
 - a. The service mesh decrypts, authenticates clients and applies policy before redirecting traffic to the application container.
5. The application **should not** manage certificates related to communication over the cluster network interface.
 - a. The service mesh manages, rotates and validates these certificates.
6. The application **must not** provide nftables or iptables rules.
7. The application **must not** use UID 1337 or tcp ports 15001 and 15020.
8. The application **must not** define Kubernetes Custom Resources in these namespaces:
 - a. *.istio.io
 - b. *.aspenmesh.io

9. The application **must not** define Kubernetes resources in the istio-system namespace.
10. The application **must** propagate tracing headers when making outgoing requests based on incoming requests.
 - a. Example: If an application receives a request with a trace header identifying that request with traceid 785a908c8d93b2d2 , and decides based on application logic that it must make a new request to another application pod to fulfill that request, it must annotate the new request with the same traceid 785a908c8d93b2d2.
 - b. The application **must** propagate all of these tracing headers if present: x-request-id, x-b3-traceid, x-b3-spanId, x-b3-parentspanid, x-b3-sampled, x-b3-flags, b3.
 - c. The application **must** propagate the tracing headers by copying any header value from the original request to the new request.
 - d. The application **should not** modify any of these header values unless it understands the format of the headers and wishes to enhance them (for example, implements OpenTracing)
 - e. If some or none of the headers are present, the application **should not** create them.
 - f. If an application makes a new request and it is not in service of exactly one incoming request, it MAY omit all tracing headers.
 - i. The application does not have to generate headers in this case. It could generate headers if it implements, for example, OpenTracing. In this case, the service mesh would use and propagate those IDs. This is optional.
 - ii. If there are no tracing headers, the service mesh generates a new trace.

Chapter 17. Application Deployment

One option for application deployment is via Helm v3. Helm provides a mechanism to deploy with site-specific templates that allow for repeatable deployment in multiple locations.

Make use of `values.yaml` to make replicable deployments in different locations with different parameters.

It is recommended that Images be built with Red Hat's Universal Base Image.

Chapter 18. Standards

18.1. Container Labeling Standards

Labels are used to organize and select subsets of objects. For example, labels enable a service to find and direct traffic to an appropriate pod. While pods can come and go, when labeled appropriately, the service detects new pods, or a lack of pods, and forwards or reduces the traffic accordingly.

When designing your label scheme, it may make sense to map applications as types, location, departments, roles, and other relevant details. The scheduler uses these attributes when colocating pods or spreading the pods out amongst the cluster. It is also possible to search for resources by label.

18.2. Image Standards

It is recommended that container images be built utilizing Red Hat's Universal Base Image (UBI) as they will have a solid security baseline, as well as support from Red Hat.

Vendors must satisfy three requirements related to maintaining proper workload isolation in a containerized environment. These requirements include:

1. Work with Red Hat's Restricted SCC¹
2. Work with Red Hat's Default SELinux Context¹
3. Evidence the container image is secure:²
 - a. Supported by dedicated, full time team providing releases of base image at least as quickly as:
 - i. Scheduled release every six weeks to pick up less critical fixes
 - ii. On-demand release for Critical or Important CVE within five days of CVE public release
 - b. Guarantees alignment with host OS packages, versions, and so on, that run tightly coupled to the container artifacts. Many CVEs and potential attacks result from mismatch of untested versions of utility functions
 - c. Ensures globally consistent time zone usage and resulting timestamps for global operators

- d. Enables Continuous Authorization to Operate (ATO). Authorize once, use many times
- e. Meets requirements of DOD, for example Air Force/DISA STIG
- f. Supports system wide crypto consistency (for example, must have same crypto implementation as our Red Hat host OS)
- g. Provides authentication of base layer via digital signature from originating vendor, and strong signature authority

¹ This is meant to forbid all changes to both primary config files (SCC, SEL) and the many related files referenced by these primary files. All security configuration files must be unchanged from the vendor's released version.

² The Red Hat UBI is able to meet these requirements, and enables images built with it to meet these requirements.

If a vendor cannot satisfy these requirements, they will have to submit a Security Exception to Network Operations for approval. Security Planning will not support approval of this exception unless the vendor commits to a roadmap for satisfying all three of these requirements within 6 months.

NOTE

This is not an exhaustive list of security requirements that vendors must satisfy. For example, it does not cover general security requirements, such as access control and logging.

18.2.1. Universal Base Image information

Universal Base Image (UBI) is designed to be a foundation for cloud-native and web applications use cases developed in containers. You can build a containerized application using UBI, push it to your choice of registry server, easily share it with others, and because it's freely redistributable, even deploy it on non-Red Hat platforms. And since it's built on Red Hat Enterprise Linux, UBI is a platform that is reliable, secure, and performant.

For more information, see [Red Hat Universal Base Images](#).

Base Images

A set of base images, Micro, Minimal, Standard, and Multi-service are provided to provide optimum starting points for a variety of use cases.

Runtime Languages

A set of language runtime images (PHP, Perl, Python, Ruby, Node.js) enable developers to start coding out of the gate with the confidence that a Red Hat built container image provides.

Complementary packages

A set of associated YUM repositories/channels include RPM packages and updates that allow users to add application dependencies and rebuild UBI container images anytime they want.

Red Hat UBI images are the preferred images to build CNFs on as they leverage the fully supported Red Hat ecosystem. In addition, once a CNF is standardized on a Red Hat UBI, the image can become Red Hat certified.

Red Hat UBI images are free to vendors so there is a low barrier of entry to getting started.

Chapter 19. Security

19.1. Elevated privilege container capabilities

Container images will not be granted the use of any non-default Linux capabilities. Such images will be blocked from running entirely or fail at runtime due to lack of privileges.

19.2. CPI-810

All CNFs should be compliant with CPI-810. If necessary, work with CPI-810 to determine CNFs compliance.

19.3. Image Security

Images should be scanned by CVE scanners while stored in the Internal Registry. Vulnerabilities found during scanning result in flags, and deployment of images with vulnerabilities require exceptions.

Images for use on the core solution must include digital signatures which validate that the image is from an authorized vendor. In addition, part or all of an authorized CNF delivered by the vendor, has a current component version, and has not been modified since signing. At a minimum, the signature must include information identifying the container base image included, as well as for the entire container contents. Accompanying software artifacts, such as Helm charts, and shell scripts must be similarly signed individually.

19.4. CNF Network Security

CNF Tenant security is the responsibility of the CNF team. Vendors must work through the onboarding process to create a security plan. CNFs must have the least permissions possible, and must implement Network Policies that drop all traffic by default. CNFs must also permit only the relevant ports and protocols to the narrowest ranges of addresses possible.

19.5. Secrets Management

Secrets objects in OpenShift provide a way to hold sensitive information, such as passwords, config files and credentials.

There are four types of secrets:

1. Service account
2. Basic auth
3. Ssh auth
4. TLS

Secrets can be added via deployment configurations or consumed by pods directly.

For more information, see [Understanding secrets](#).

Chapter 20. Contributors

Name	Title	Email	Area of Contribution

Chapter 21. Document History

Version	Date	Change	Version POC

Chapter 22. Document Approvals

Name	Title	Company	Date of Approval