Cloud Native Network Function Requirements

Version 1.3

Jan 2022

# **Introduction**

Red Hat is building a Telco platform to serve network needs across Core, Edge, Lite and Far Edge. In this journey, the platform will be the entry path for 5G Core cloud-native NF’s and bring forth the introduction of a CNCF-based stack for Cloud-Native Network Functions (CNFs). Red Hat is building a Kubernetes-based CaaS (Container as a Service) Platform with PaaS (Platform as a Service) services to support 5G Services-based Architecture.

# **Scope**

This document and the current platform configuration is 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.

# **Refactoring**

NFs should break their software down into the smallest set of microservices as possible. Running monolithic applications inside of 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 (pebbles). All cloud-native network functions (CNFs) should 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 should be within the same name space.

There is a quote that describes this best from [Lewis and Fowler](https://martinfowler.com/articles/microservices.html): "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. "

### **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 so all containers in the same Pod have access to the same mount namespace and can mount the same volumes.

# **CNF Developer Guidelines**

### **Preface**

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

### Goal

This section is mainly for the developers of CNFs, who need to build high-performance Network Functions in a containerized environment. The guidance provided in this guide 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.

### Non-Goal

This is not a guide on how to build CNF’s functionality.

### 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 should not be requesting escalated privilege to perform this function.
3. Avoid using any host system resource if possible.
4. Leveraging read only root filesystem when possible.

### Avoid Accessing Resources on Host

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

### Avoid Mounting host directories as volumes

It is not necessary to mount host /sys/ or host /dev/ directory as a volume in a pod in order to use a network device such as SR-IOV VF. The moving of a network interface into the pod network namespace is done automatically by CNI. Mounting the whole /sys/ or /dev/ directory in the container will overwrite the network device descriptor inside the container which causes '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 DPDK applications with SR-IOV VF, device specs (in 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 specs under '/dev/vfio/' in the container.

### 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.

### 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]](#kix.m02gjmmhqh0e) 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 that 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.

Refer to [[1]](#kix.h926go207d3h) for more details on how to define and use the SCC.

## **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"

### 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.

### NET\_ADMIN

NET\_ADMIN capability is required to perform various network related administrative operations inside container 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 doesn't include:

* add/delete 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 should already be supported on the host by various CNIs in Openshift.

### (Avoid) SYS\_ADMIN

This capability is very powerful and overloaded. It allows the application to perform a range of system administration operations to the host. So you should avoid requiring this capability in your application.

### SYS\_NICE

In the case that a CNF is running on a node using the real-time kernel, SYS\_NICE will be used to allow real-time application to switch to SCHED\_FIFO.

### SYS\_PTRACE

This 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 SIGHUP from a process in a Container to another process in another Container. See [[9]](#kix.x9sn5ltpet8c) for more details

### Operations that shall be executed by OpenShift

The application should 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 shall be configured by modifying the manifests generated by openshift-installer before deploying the cluster. Refer to [[4]](#kix.1mwd3pp1xi0x) for more information.
  + For the additional networks managed by the Cluster Network Operator, it can be configured through the NetworkAttachmentDefinition resources generated by the Cluster Network Operator. Refer to [[5]](#kix.hutwtlst36x3) for more information.
  + For the SRIOV interfaces managed by the Sriov Network Operator. Refer to [[6]](#kix.prsfggqin4x0) for more information.
* Link state modification
  + All the links will be set to up before attaching it to a pod.
* IP/MAC address assignment
  + For all the networks, the IP/MAC address will be assigned to the interface during pod creation.
  + Multus also allows users to override the IP/MAC address. Refer to [[7]](#kix.7snexvyvu78o) for more information.
* Manipulate pod’s route table
  + By default, the default route of the pod will point to the cluster network, with or without the additional networks. Multus also allows users to override the default route of the pod. Refer to [[7]](#kix.7snexvyvu78o) for more information.
  + Non-default routes can be added to pod’s routing table by various IPAM CNI plugins during pod creation
* SRIOV VF setting
  + Besides the functions aforementioned, the SRIOV Network Operator supports configuring the following parameters for SRIOV VFs. Refer to [[8]](#kix.jowj8zhw9jik) for more information.
    - 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 it please refer to [[2]](#kix.ekb08grpyt9o) [[3]](#kix.v13xkav9etjm).
  + If your application works as a multicast source and you want to utilize the additional interfaces to carry the multicast traffic, then you don’t need the NET\_ADMIN capability. But you need to follow the instruction in [[3]](#kix.v13xkav9etjm) to set the correct multicast route in your pod’s routing table.

### Operations that can NOT be executed by OpenShift

All the CNI plugins will only be 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 would be required. There are some other functionalities that are not currently supported by any of the OpenShift components which would 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, tso, etc.
* 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

### 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 can find out what capabilities an application requires in order to run in a container. It also shows the syscalls which were invoked.

Another tool is ‘capable’ which is part of the BCC tools. It can be installed on RHEL8 with “dnf install bcc”.

### 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  
[...]

1. Use capable command

$ /usr/share/bcc/tools/capable

1. 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

1. 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

[...]

1. 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)

## **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 what capabilities it shall request.

### **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 |

# **Cloud-Native CNFs - SCC Implementation**

There are potentially three SCC profiles that may be utilized by OpenShift Container platform. All Apps are expected to fit in Category 1 (see description below) employing the default SCC profile. Apps matching Category 2 or Category 3 below require exception approval.

## **CNFs that do not require advanced networking features (Category 1)**

This kind of CNFs shall:

1. Uses the default CNI (OVN) network interface.
2. 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\_RAW

fsGroup:

type: MustRunAs

readOnlyRootFilesystem: false

runAsUser:

type: MustRunAsRange

seLinuxContext:

type: MustRunAs

supplementalGroups:

type: RunAsAny

volumes:

- configMap

- downwardAPI

- emptyDir

- persistentVolumeClaim

- projected

- secret

## **CNFs that require advanced networking features (Category 2)**

The CNFs with following characteristics may fall into this category:

1. Manipulate the low-level protocol flags, such as the 802.1p priority, VLAN tag, DSCP value, etc.
2. Manipulate the interface IP addresses or the routing table or the firewall rules on-the-fly.
3. Process Ethernet packets

This kind of CNF may:

1. Use Macvlan interface to sending and receiving Ethernet packets
2. Request CAP\_NET\_RAW for creating raw sockets
3. Request CAP\_NET\_ADMIN for
   1. Modify the interface IP address on-the-fly
   2. Manipulating the routing table on-the-fly
   3. Manipulating firewall rules on-the-fly.
   4. Setting 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

## **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.

This kind of CNF shall:

1. Use SR-IOV interfaces.
2. Fully or partially bypassing the kernel networking stack with userspace networking technologies, like DPDK, F-stack, VPP, OpenFastPath, etc. 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 shall be 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.

Here is an example pod manifest of a DPDK application, more can be found at [[12]](https://docs.google.com/document/d/1QwS5Plr9uMgu9hCKNQdiXpWbC9u7ErpBwr761aJ5-ec/edit?ts=5ef200f2" \l "bookmark=id.kql6vaad1cf0).

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/mlxnics: "1"

memory: "1Gi"

cpu: "4"

hugepages-1Gi: "4Gi"

requests:

openshift.io/mlxnics: "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

[1] <https://docs.openshift.com/container-platform/4.9/authentication/managing-security-context-constraints.html>

[2]<https://docs.openshift.com/container-platform/4.9/networking/openshift_sdn/enabling-multicast.html>

[3] [https://access.redhat.com/documentation/en-us/openshift\_container\_platform/4.9/html-single/networking/index#using-sriov-multicast](https://access.redhat.com/documentation/en-us/openshift_container_platform/4.9/html-single/networking/index" \l "using-sriov-multicast)

[4] <https://docs.openshift.com/container-platform/4.9/installing/installing_bare_metal/installing-bare-metal-network-customizations.html>

[5] <https://docs.openshift.com/container-platform/4.9/networking/multiple_networks/understanding-multiple-networks.html>

[6] <https://docs.openshift.com/container-platform/4.9/networking/hardware_networks/configuring-sriov-device.html>

[7] <https://docs.openshift.com/container-platform/4.9/networking/multiple_networks/attaching-pod.html>

[8]

<https://docs.openshift.com/container-platform/4.9/networking/hardware_networks/configuring-sriov-net-attach.html>

[9] <https://kubernetes.io/docs/tasks/configure-pod-container/share-process-namespace/>

[10] <https://linuxera.org/capabilities-seccomp-kubernetes/>

# **CNF Expectations and Permissions**

### **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 w/Messaging Interface

A container should address 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 should catch SIGTERM or SIGKILL from the platform and shut 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; i.e. customized images for different environments should typically not be built. Instead, an external means for storing and retrieving configurations that vary across environments for the container should be used.

Additionally, the container image should NOT dynamically install additional packages at runtime.

Process Disposability

Containers should be 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 should be used. For information such as user sessions, use of an external, low-latency, key-value store such as redis should be used.

Process disposability also requires that the application should 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’. I.e. When one service in your application depends on another service, it should 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 should 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.

### **High Level CNF Expectations**

* CNFs shall be built to be cloud-native
* Containers never run as root (uid=0). Applications that require elevated privileges will require an exception with HQ Planning
* Containers run with the minimal set of permissions required. Any pods that require elevated privileges require a security review providing an analysis of the special permissions required, and an exception should be provided. Avoid Privileged Pods. If Privileged Pods are required, the CNF developer should work with the planning department, Security Risk Management and Redhat 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 & 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 (Pods, Services, etc.) 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) shall result in a fully-functional CNF ready to serve traffic, without requiring any post-instantiation configuration of system parameters
* CNFs shall implement service resilience at the application layer and not rely on individual compute availability/stability
* CNFs shall decouple application configuration from Pods, to allow dynamic configuration updates
* CNFs shall support elasticity with dynamic scale up/down using kubernetes-native constructs such as ReplicaSets, etc.
* CNFs shall support canary upgrades employing the platform Service Mesh
* CNFs shall 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) shall be employed at a minimum.

### **Platform Restrictions**

* CNFs may not deploy Nodeports
* CNFs may not use host networking
* Namespace creation will be performed by the platform team and should 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

# **Core/Edge**

# **OpenShift Platform**

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

[Certified Operator Guide](https://redhat-connect.gitbook.io/certified-operator-guide/) - Guidelines on how to build an Operator that meets the Red Hat certification criteria

[Partner Guide for Container and Operator Certification](https://redhat-connect.gitbook.io/partner-guide-for-red-hat-openshift-and-container/) - Step-by-step instructions for partners on how to certify their images and operators

[OpenShift Operator Badge Guide](https://redhat-connect.gitbook.io/openshift-badges/badges/cloud-native-network-functions-cnf/overview) - Containers and Operators tests

# **Stack Overview 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 section. 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.

### **CaaS**

## **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.

[https://docs.openshift.com/container-platform/4.7/cli\_reference/helm\_cli/getting-started-with-helm-on-openshift-container-platform.html#installing-a-helm-chart-on-an-openshift-cluster\_getting-started-with-helm-on-openshift](https://docs.openshift.com/container-platform/4.7/cli_reference/helm_cli/getting-started-with-helm-on-openshift-container-platform.html" \l "installing-a-helm-chart-on-an-openshift-cluster_getting-started-with-helm-on-openshift)

## **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 with which applications and people and applications interact with the cluster. There are several ways to do this, via the kubectl or oc CLI tools, via web based UIs or interacting directly with 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, bypassing the need to exit the cluster and come back in.

## 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.

## 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 will be lost in the event 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, one 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 should clear persistent storage by deleting their PVs when removing their application from a cluster.

Red Hat Persistent Storage Documentation

<https://docs.openshift.com/container-platform/4.4/storage/container_storage_interface/persistent-storage-csi.html>

## 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.

More info on Block Volume storage support here:

[https://docs.openshift.com/container-platform/4.4/storage/understanding-persistent-storage.html#block-volume-support\_understanding-persistent-storage](https://docs.openshift.com/container-platform/4.3/storage/understanding-persistent-storage.html" \l "block-volume-support_understanding-persistent-storage)

## 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.

## 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 should be OCI compliant. Images are recommended to be built using Red Hat's open Universal Base Image. See section 9.1.8 for additional information UBI and support.

Red Hat Documentation on CRI-O and Read Only Containers

<https://blog.openshift.com/add-a-layer-of-security-to-openshift-kubernetes-with-cri-o-in-read-only-mode/>

<https://github.com/cri-o/cri-o/blob/master/docs/crio.8.md>

This environment is maintained through the open source tools:

* runc
* skopeo
* buildah
* podman
* crio

## **CPU Manager/Pinning**

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

## **Host OS**

The 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 is 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 MCO (Machine Config Operator).

The controlled immutability 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 on RHCOS please refer to the following link.

[Red Hat Enterprise Linux CoreOS | Architecture | OpenShift Container Platform 4.7](https://docs.openshift.com/container-platform/4.7/architecture/architecture-rhcos.html)

## **Universal Base Image**

[https://developers.redhat.com/products/rhel/ubi/#assembly-field-sections-18455](https://developers.redhat.com/products/rhel/ubi/" \l "assembly-field-sections-18455)

The Red Hat 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 to non-Red Hat platforms -- no subscription is required. Since it’s built on Red Hat Enterprise Linux, UBI has the same industry leading reliability, security and performance benefits.

Base Images

A set of three base images (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 will 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.  
It is possible to utilize other base images to build containers that can be run on the OpenShift platform. See the link below for a view of the ease of support for containers utilizing various base images and differing levels of certification and supportability.

<https://redhat-connect.gitbook.io/partner-guide-for-red-hat-openshift-and-container>

# **PaaS Core/Edge**

### **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.

### **Distributed Tracing**

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

### **Pod Security**

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

### **Load Balancer/Service Proxy**

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

### **CI/CD Framework**

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

### **Kubernetes API versions**

OpenShift platform supports the full Kubernetes APIs as well as additional API calls that are OpenShift specific. Documentation can be found at the below link:

<https://docs.openshift.com/container-platform/4.7/rest_api/index.html>

# **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. SCC documentation from Red Hat can be found here : <https://docs.openshift.com/container-platform/4.7/authentication/managing-security-context-constraints.html>

Pods will execute on worker nodes, by default being 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 from the namespace annotation.
* 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.

# **OpenShift Best Practices**

### **Logging**

Log aggregation and analysis

The OpenShift platform will support logging from containers and forwarding those logs separately from the platform logging to a centralized logging repository. Logs may be forwarded based on Tenant Namespace identifier.

* Containers are expected to write logs to stdout
* Requires vendor to follow pod/container naming standards
* Logs will be 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 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 use 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 anything 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 the 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"

}

### **Monitoring**

Network Functions are expected to bring their own metrics collection functions (e.g. Prometheus) for their application specific metrics. This metrics collector will not be expected to nor be able to poll platform level metric data. Network Functions shall support exposing their Prometheus collection functions via PromQL interfaces to existing OSS systems.

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

### **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 will then impose the CPU “Limits” to ensure the Pod doesn’t consume more than the intended allocation. The limit can never be lower than the request.

NUMA Configuration

OpenShift provides a topology manager which leverages the CPU manager and Device manager to help associate processes to CPUs. Topology manager handles NUMA affinity. This feature is available as of OpenShift 4.6. For some examples on how to leverage the topology manager and creating workloads that work in real time, see the following links.

<https://docs.openshift.com/container-platform/4.7/scalability_and_performance/using-topology-manager.html>

[https://docs.openshift.com/container-platform/4.7/scalability\_and\_performance/cnf-performance-addon-operator-for-low-latency-nodes.html#performance-addon-operator-creating-workload-that-works-in-real-time\_cnf-master](https://docs.openshift.com/container-platform/4.7/scalability_and_performance/cnf-performance-addon-operator-for-low-latency-nodes.html" \l "performance-addon-operator-creating-workload-that-works-in-real-time_cnf-master)

### **Memory Allocation**

Regarding memory allocation, there are a couple of considerations. How much of the platform is OpenShift itself using, and 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 limits. In the case of where a memory request has been specified, OpenShift will not schedule the pod unless the amount of memory required to launch it is available. In the case of a limit being specified, OpenShift will 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 will then impose the memory “Limits” to ensure the Pod doesn’t consume more than the intended allocation. The limit can never be lower than the request.

Vendors must supply quotas [per project](https://docs.openshift.com/container-platform/4.4/applications/quotas/quotas-setting-per-project.html) so that nodes can be sized appropriately and clusters are able to support the needs of vendor applications.

### **Affinity/Anti-affinity**

In 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

[https://docs.openshift.com/container-platform/4.7/nodes/scheduling/nodes-scheduler-pod-affinity.html#nodes-scheduler-pod-affinity](https://docs.openshift.com/container-platform/4.7/nodes/scheduling/nodes-scheduler-pod-affinity.html" \l "nodes-scheduler-pod-affinity)

### **Taints and Tolerations**

Taints and tolerations allow the Node to control which Pods should (or should not) be 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 apply tolerations for NoExecute, PreferNoSchedule, and NoSchedule.

See <https://docs.openshift.com/container-platform/4.7/nodes/scheduling/nodes-scheduler-taints-tolerations.html> for more information.

### **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. See the following for more information.

<https://docs.openshift.com/container-platform/4.7/applications/quotas/quotas-setting-per-project.html>

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.

<https://docs.openshift.com/container-platform/4.7/nodes/clusters/nodes-cluster-overcommit.html>

### **Pods**

### **No naked pods**

Do not use naked Pods (that is, Pods not bound to a ReplicaSet or Deployment). Naked Pods will not be rescheduled in the event of a node failure.

### **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.

<https://docs.openshift.com/container-platform/4.7/openshift_images/managing_images/tagging-images.html>

### **“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 separate 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, sidecar containers, etc. which may justify running multiple containers in a single pod.

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

More information about pods can be found here.

<https://docs.openshift.com/container-platform/4.7/nodes/pods/nodes-pods-using.html>

### **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.

<https://docs.openshift.com/container-platform/4.7/nodes/containers/nodes-containers-init.html>

### **Security/RBAC**

Roles / RolesBinding - A Role represents a set of permissions within a particular namespace. E.g: 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. E.g: A given user has “cluster-admin” privileges on the cluster. The ClusterRoleBinding is used for granting the permissions defined in a ClusterRole to a user or group of users.

<https://docs.openshift.com/container-platform/4.7/authentication/using-rbac.html>

### **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 should be used for every interface that can be supported from a technical standpoint.

[https://docs.openshift.com/container-platform/4.7/networking/multiple-networks/understanding-multiple-networks.html](https://docs.openshift.com/container-platform/4.7/networking/multiple_networks/understanding-multiple-networks.html)

## **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 exception use cases. Multus interfaces will be defined by the platform operations team for the network functions which can then consume them. Multus interfaces will have to be requested via the planning tools ahead of time by the companies personnel. VLANs will be 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.

<https://docs.openshift.com/container-platform/4.7/networking/hardware_networks/about-sriov.html>

By configuring the SR-IOV Network CRs named NetworkAttachmentDefinitions are exposed by the SR-IOV Operator in the CNF namespace.

Different names will be assigned to different Network Attachment Definitions that are namespace specific. MACVLAN versus Multus interfaces will be named differently to distinguish the type of device assigned to them (created by configuring SR-IOV devices via the SriovNetworkNodePolicy CR).

From the CNF perspective, a defined set of network attachment definitions will be 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 will be 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.

### SR-IOV Interfaces 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 (see [https://docs.openshift.com/container-platform/4.7/networking/hardware\_networks/add-pod.html#runtime-config-ethernet\_configuring-sr-iov](https://docs.openshift.com/container-platform/4.7/networking/hardware_networks/add-pod.html" \l "runtime-config-ethernet_configuring-sr-iov)).

Example sriovnetworknodepolicy: NOTE: this is enabled by the cluster administrator.

apiVersion: sriovnetwork.openshift.io/v1

kind: SriovNetworkNodePolicy

metadata:

name: nnp-w1ens3f0grp2

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: w1ens3f0grp2

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

Example 1: Empty IPAM

apiVersion: sriovnetwork.openshift.io/v1

kind: SriovNetwork

metadata:

name: sriovnet

namespace: openshift-sriov-network-operator

spec:

capabilities: '{ "mac": true }'

ipam: '{}'

networkNamespace: <CNF-NAMESPACE>

resourceName: w1ens3f0grp2

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":"FD97:0EF5:45A5:4000:00D0:0403:0000:0001/64","range\_start":"FD97:0EF5:45A5:4000:00D0:0403:0000:0001","range\_end":"FD97:0EF5:45A5:4000:00D0:0403:0000:0020","routes":[{"dst":"fd97:0ef5:45a5::/48","gw":"FD97:EF5:45A5:4000::1"}]}'**

**networkNamespace: <CNF-NAMESPACE>**

resourceName: w1ens3f0grp2

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":"10.120.26.1"}]}'

networkNamespace: <CNF-NAMESPACE>

resourceName: w1ens3f0grp2

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 both whereabouts and static IPAM are managed external to the cluster.

The above 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

### 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, as per the following example:

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"]

}

]

### 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, pci address.

*For 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 a file content inside the pod, at the /etc/podnetinfo/annotations path. A convenience library is available to easily consume those informations from the application (bindings in C and Go [https://docs.openshift.com/container-platform/4.7/networking/hardware\_networks/about-sriov.html#nw-sriov-app-netutil\_about-sriov](https://docs.openshift.com/container-platform/4.7/networking/hardware_networks/about-sriov.html" \l "nw-sriov-app-netutil_about-sriov)).

### 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).

See the Performance Add-On Operator section for NUMA awareness and more information about how HugePages are turned on.

### **Upgrades**

* + 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 <https://kubernetes.io/docs/reference/using-api/deprecation-policy/> shall 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 shall 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 should not be tied to a specific version of Kubernetes or any of its components

### **OpenShift Virtualization/kubevirt**

* + 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. In order 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 should be installed according to its documentation, and only documented supported features may be used unless an explicit exception has been granted.

See: <https://docs.openshift.com/container-platform/4.7/virt/about-virt.html>

In order 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 physical CPUs[1] and "isolate" QEMU threads, such as the emulator thread and the IO thread, on a separate physical CPU[2][3] so it will not affect the workloads CPU latency.

Similar to OpenStack, OpenShift Virtualization supports the device role tagging mechanism[4]. for the network interfaces (same format as it is in OSP). Users will be able to tag Network interfaces in the API and identify them in device metadata provided to the guest OS via the config drive.

[1] <https://docs.openshift.com/container-platform/4.7/virt/virtual_machines/advanced_vm_management/virt-dedicated-resources-vm.html>

[2] [https://kubevirt.io/user-guide/virtual\_machines/dedicated\_cpu\_resources/#requesting-dedicated-cpu-for-qemu-emulator](https://kubevirt.io/user-guide/virtual_machines/dedicated_cpu_resources/" \l "requesting-dedicated-cpu-for-qemu-emulator)

[3] [https://kubevirt.io/user-guide/#/creation/disks-and-volumes?id=iothreads-with-qemu-emulator-thread-and-dedicated-pinned-cpus](https://kubevirt.io/user-guide/" \l "/creation/disks-and-volumes?id=iothreads-with-qemu-emulator-thread-and-dedicated-pinned-cpus)

[4] [https://kubevirt.io/user-guide/virtual\_machines/startup\_scripts/#device-role-tagging](https://kubevirt.io/user-guide/virtual_machines/startup_scripts/" \l "device-role-tagging)

* + 1. 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 in order 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 should 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.

# **Operator Best Practices**

<https://github.com/operator-framework/community-operators/blob/master/docs/best-practices.md>

# **Container Best Practices**

### **Pod Exit Status**

The most basic requirement for the lifecycle management of Pods in OpenShift are the ability 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 different ways a pod can stop on Kubernetes. One way is that the pod can remain alive but non-functional. Another way is that the pod can crash and become non-functional. In the first 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. For the second case, when the application in the pod stops, it should exit with a code and write suitable log entries to help the administrator diagnose what the issue was that caused the problem.

Pods should use terminationMessagePolicy: FallbackToLogsOnError to summarize why they crashed and use stderr to report errors on crash

### **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 which means that the process in the container should start shutting down, closing connections and stopping all activity. If the pod doesn’t shut down within the default 30 seconds then the platform may send a SIGKILL signal which will stop the pod immediately. This method isn’t as clean and the default time between the SIGTERM and SIGKILL messages can be modified based on the requirements of the application.

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

### **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. In order for OpenShift to do this correctly, the application developer must create a resource profile for the application. This resource profile should contain requirements such as how much memory, cpu and storage that the application needs. At that point, the scheduler is aware of which nodes in the cluster that can satisfy the workload and place the application on one of those nodes (or distribute it), or the scheduler will place the pod that the application is in, in a pending state until resources come available.

All pods should 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.

### **Storage: emptyDir**

There are several options for volumes and reading and writing files in OpenShift. When the requirement is temporary storage and given the option to write files into directories in containers versus an external filesystems, choose the emptyDir option. This will provide the administrator with the same temporary filesystem - when the pod is stopped the dir is deleted forever. Also, 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 limits instead of writing to the container directories will also allow enabling readonlyRootFilesystem on the container or pod.

### **Liveness and Readiness Probes**

As part of the pod lifecycle, the OpenShift platform needs to know what state the pod is in at all times. This can be accomplished with different health checks. There are at least three states that are important to the platform: 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 can be running but not actually able to serve requests. This can be detected with liveness checks. The liveness check will 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 will result in the container being restarted.

There is also a consideration of pod 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.

### **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 the if 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.

### **Automount Services for Pods**

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

### **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 there is a pod disruption budget set then OpenShift will respect that and ensure that the required number of pods are available by bringing up pods on different nodes before draining the current node.

# **Networking Overview**

OpenShift is a multi-tenant environment. NFs will be deployed within a single namespace. Supporting applications like an OAM platform for multiple NFs from the same vendor should 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 space, they are only reachable via service VIPs on the load balancers. The POD network may be NATed as traffic egresses the load balancer. Traffic inbound will be destination NATed to Service/Pod IP addresses.

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

### **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.

### **User Plane Functions**

**Performance Addon Operator (PAO)**

Red Hat created the [Performance Addon Operator](https://docs.openshift.com/container-platform/4.7/scalability_and_performance/cnf-performance-addon-operator-for-low-latency-nodes.html) 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 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

**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:

hugepages:

defaultHugepagesSize: "2M"

pages:

- size: "2M"

count: 32000

node: 0

- size: "2M"

count: 32000

node: 1

To request hugepages, 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 further reading on OCP's support of huge pages, see the Configuring huge pages documentation below..

[https://docs.openshift.com/container-platform/4.7/scalability\_and\_performance/cnf-performance-addon-operator-for-low-latency-nodes.html#cnf-configuring-huge-pages\_cnf-master](https://docs.openshift.com/container-platform/4.7/scalability_and_performance/cnf-performance-addon-operator-for-low-latency-nodes.html" \l "cnf-configuring-huge-pages_cnf-master)

**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

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: 0-3,40-43

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 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

Example pod.spec:

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 s

requests:

cpu: 1

memory: 1Gi

hugepages-2Mi: 1000Mi

restartPolicy: Always

**runtimeClassName: performance-perf-profile-2m-worker**

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**

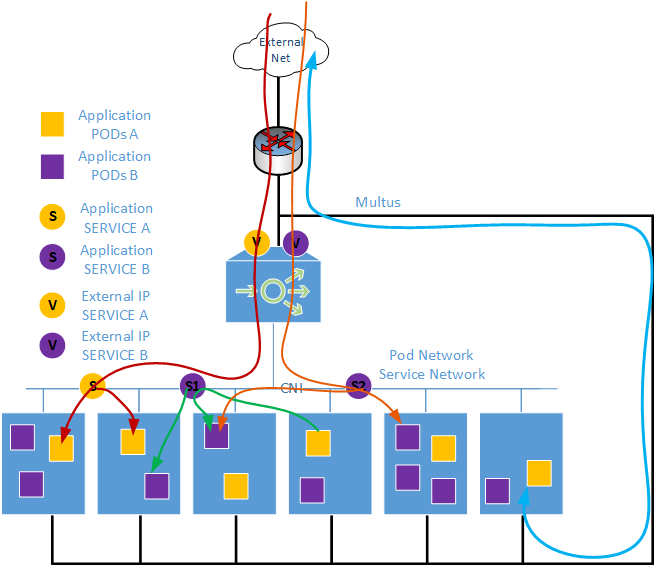
For each container in a pod with the restricted topology management policy, kubelet calls each Hint Provider to discover their resource availability. Using this information, the Topology Manager stores the preferred NUMA Node affinity for that container. If the affinity is not preferred, Topology Manager 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, the pod is admitted to the node. If a single NUMA Node affinity 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 on Topology manager, see the following [OpenShift Documentation](https://docs.openshift.com/container-platform/4.7/scalability_and_performance/using-topology-manager.html).

# **Application Service Exposure to External Networks**



The red 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 associated with their application specifically. 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 green 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 blue 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.

# **Service Mesh for Inter/Intra NF**

### **Service Mesh Introduction**

Kubernetes clusters have 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 and any details of that communication (plaintext or encryption, what

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 thus,

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 from or to 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. 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 in this implementation is found, 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 communicate in any

special way with the sidecar directly. Generally, application containers should be 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

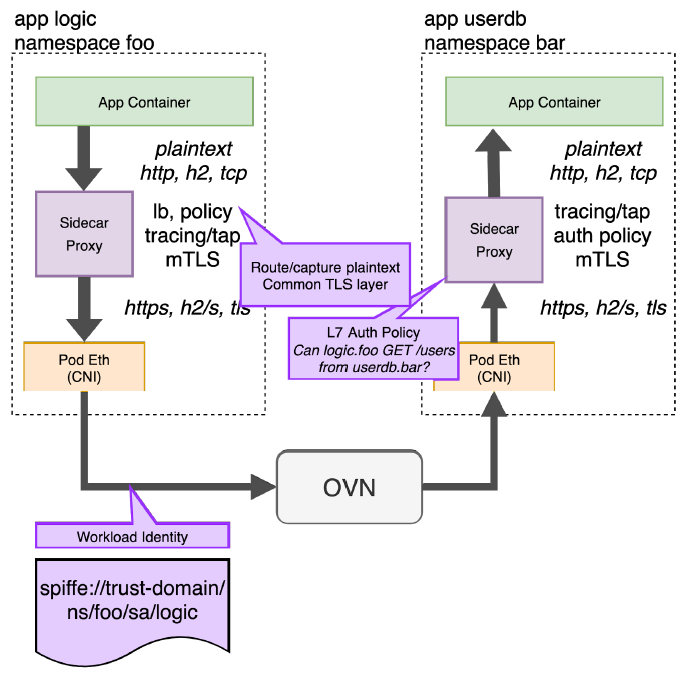
requirements so that the sidecar can correctly proxy application traffic and apply policy.

Service Mesh is also capable of doing CSR generation, signing and installation 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 via kubernetes secrets as a volume mount should an application require the keys for doing any TLS via Multus based interfaces.

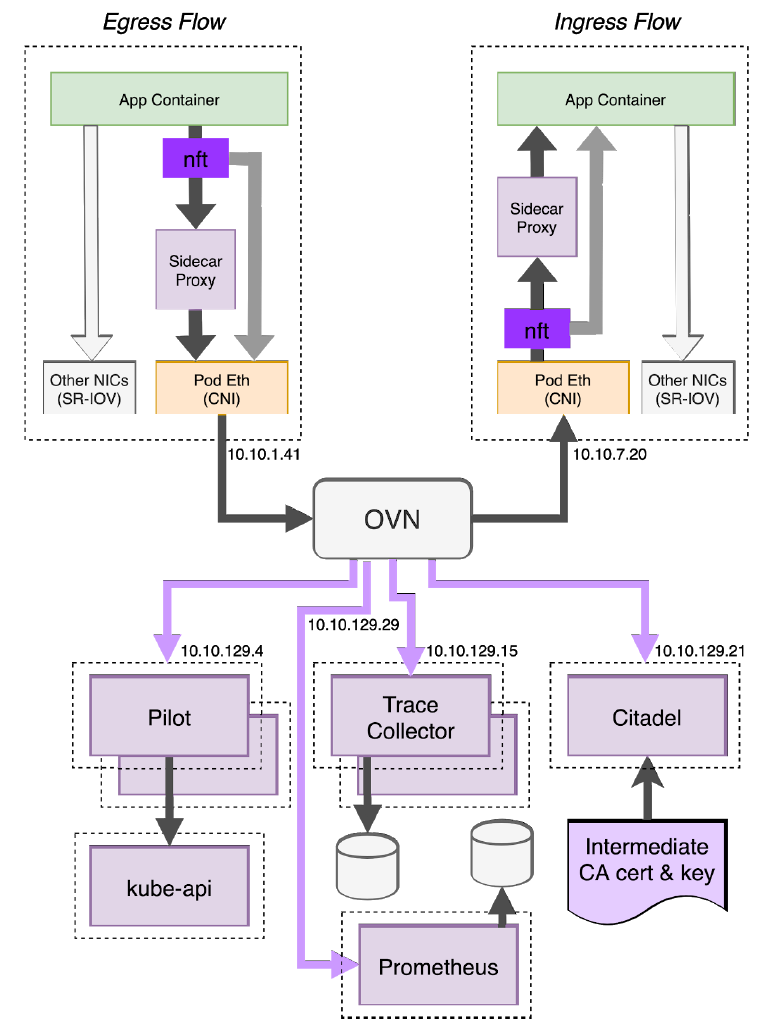
Additionally Service Mesh allows distributed tracing for HTTP based flows that can be analyzed via the Jaeger UI. This provides a consistent visibility mechanism across NFs for Service Based Interface (SBI) based communication.

The following diagram depicts Service Mesh at a high level.

*Service Mesh High Level Overview*



*Service Mesh Detailed View*



This diagram depicts 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.

### **Service Mesh Tapping**

Because of Service Mesh's unique location related to the HTTP traffic for SBI interfaces, it is well positioned to provide a tapping solution to feed tools for doing traces on network traffic for the purposes of call traces and evaluating overall network health via statistical analysis. Because Service Mesh is able to front end all SBI interfaces, it provides the capacity for comprehensive and consistent visibility across all of the SBI interfaces within the 5G Core.

### **Service Mesh Requirements for CNF**

* + 1. The application MUST declare all listening ports as containerPorts in the Pod specification it provides to Kubernetes.
       1. The application MUST NOT listen on any other ports that are undeclared.
       2. The service mesh MAY be configured to block connections to these ports.
       3. These ports MUST be named in the pod specification with the protocol they implement.
          1. 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.
          2. Preferred prefixes: grpc , grpc-web , http , http2
          3. Fallback prefixes: tcp , udp
          4. Valid Example: http-webapi or grpc
    2. The application MUST communicate with Kubernetes Services by their service IP instead of selecting Pods in that service individually.
       1. The service mesh will select the appropriate pod.
    3. The application MUST NOT encrypt outbound traffic on the cluster network interface.
       1. The service mesh will apply policy, authenticate servers and encrypt outbound traffic before it leaves the application pod.
    4. The application MUST NOT decrypt inbound traffic on the cluster network interface.
       1. The service mesh will decrypt, authenticate clients and apply policy before redirecting traffic to the application container.
    5. The application SHOULD NOT manage certificates related to communication over the cluster network interface.
       1. The service mesh will manage, rotate and validate these certificates.
    6. The application MUST NOT provide nftables or iptables rules.
    7. The application MUST NOT use UID 1337 or tcp ports 15001, 15020.
    8. The application MUST NOT define Kubernetes Custom Resources in these namespaces:
       1. \*.istio.io
       2. \*.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.
        1. 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.
        2. 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
        3. The application MUST propagate the tracing headers by copying any header value from the original request to the new request.
        4. The application SHOULD NOT modify any of these header values unless it understands the format of the headers and wishes to enhance them (e.g. implements OpenTracing)
        5. If some or none of the headers are present, the application SHOULD NOT create them.
        6. If an application makes a new request and it is not in service of exactly one incoming request, it MAY omit all tracing headers.
           1. Note: The application does not have to generate headers in this case. It could generate headers if it implements e.g. OpenTracing, and the service mesh would use and propagate those IDs. This is optional.
           2. If there are no tracing headers, the service mesh will generate a new trace.

# **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.

# **Standards**

### **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 will detect new pods, or a lack of pods, and forward or reduce the traffic accordingly.

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

### **Image Standards**

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

Vendors must satisfy 3 requirements related to maintaining proper workload isolation in a containerized environment:

* Work with Red Hat’s Restricted SCC¹
* Work with Red Hat’s Default SELinux Context¹
* Evidence the container image is secure:²
* Supported by dedicated, full time team providing releases of base image at least as quickly as:
  + - Scheduled release every 6 weeks to pick up less critical fixes.
    - On-demand release for Critical or Important CVE within 5 days of CVE public release.
  + Guarantees alignment with host OS packages, versions, etc. that will run tightly coupled to the container artifacts. Many CVEs and potential attacks result from mismatch of untested versions of utility functions.
* Ensures globally consistent time zone usage and resulting timestamps for global operators
* Enables Continuous Authorization to Operate (ATO). Authorize once, use many times.
* Meets requirements of DOD, for example Air Force/DISA STIG
* Supports system wide crypto consistency (e.g. must have same crypto implementation as our Red Hat host OS)
* 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 3 of these requirements within 6 months.

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

### **Universal Base Image information**

[https://developers.redhat.com/products/rhel/ubi/#assembly-field-sections-18455](https://developers.redhat.com/products/rhel/ubi/" \l "assembly-field-sections-18455)

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.

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 will leverage the fully supported Red Hat ecosystem. In addition, once a CNF is standardized on a Red Hat UBI, the image can be become Red Hat certified.

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

# **Security**

### **Elevated privilege container capabilities**

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

### **CPI-810**

All CNFs should be compliant with CPI-810. Work to determine CNFs compliance with CPI-810.

### **Image Security**

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

Images for use on the core solution must include digital signatures which validate that the image is from an authorized vendor, 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.

### **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 CNFs must implement Network Policies that drop all traffic by default and permit only the relevant ports and protocols to the narrowest ranges of addresses possible.

### **Secrets management**

Secrets objects in OpenShift provide a way to hold sensitive information such as passwords, config files and credentials. There are 4 types of secrets; service account, basic auth, ssh auth and TLS. Secrets can be added via deployment configurations or consumed by pods directly. For more information on secrets and examples, see the following documentation.

<https://docs.openshift.com/container-platform/4.7/nodes/pods/nodes-pods-secrets.html>