Cloud Native Network Function Requirements

Version 1.5

*Note: this doc a cleaned version (Added TOC formatting) of:* [VCP CNF Requirements Doc - v1.5 Draft clean](https://docs.google.com/document/d/1bjEIaAc8ZMbLXWbvtYphFv4Gqb-1bfhA_4WGrf_kV2M/edit)

[**1 OpenShift Platform 4**](#_vey3r3sptwir)

[1.1 Helm v3 4](#_wu3kvg85krs0)

[1.2 Kubernetes 5](#_jlh7c9efr18i)

[1.3 CNI-OVN 5](#_opif1kcpamgx)

[1.4 Container Storage (CSI) 5](#_gml22owov43r)

[1.5 Block Storage 6](#_otqmxeohgigf)

[1.6 Ephemeral Storage 6](#_8i21o33nt3xb)

[1.7 Local Storage 6](#_pkzwp0ao5eai)

[1.8 Container Runtime 7](#_yje7le1f43r9)

[1.9 CPU Manager/Pinning 7](#_a6aho9c3m6cz)

[1.10 Host OS 7](#_fz0a8r8hra3m)

[1.11 Universal Base Image 8](#_3s1se71w8dg9)

[1.11.1 Base Images 8](#_27xi6ou5ydjp)

[1.11.1.1 Runtime Languages 8](#_izswqdf7flea)

[1.11.1.2 Complementary packages 8](#_3zgd212kk1dc)

[1.12 Pod Security 9](#_l24vw9gmezp5)

[1.13 CI/CD Framework 9](#_mphdzxqybhn9)

[1.14 Kubernetes API Versions 9](#_sgd4veql3wpj)

[1.15 OVN-Kubernetes CNI 9](#_58fhts13ttya)

[1.16 User Plane Functions 9](#_hbch9e6sofrd)

[1.16.1 Performance Addon Operator (PAO) 9](#_vhmth02n3tjf)

[1.16.2 Hugepages 10](#_5kkbv98gjvqt)

[1.16.3 CPU Isolation 10](#_dk0813uyff3n)

[1.16.4 Topology Manager and NUMA Awareness 10](#_8p8g0yvgukcw)

[1.16.5 IPv4 & IPv6 11](#_ijasp68msmle)

[1.16.6 VRFs (aka routing instances) 11](#_4n6tmgy63xf3)

[1.16.7 Ports reserved by OpenShift 11](#_ebd5rw8hn6rn)

[**2 CNF Developer Guide 12**](#_z2ydgyzfvn4g)

[2.1 Preface 12](#_44jtqeefynql)

[2.2 Goal 12](#_bwl41sjvxo8m)

[2.3 Non-Goal 12](#_2u1hri2k7zma)

[2.3.1 Refactoring 12](#_muwv6ajv1ra)

[2.3.2 CNF Security 13](#_gwnqhddpfiw3)

[2.3.2.1 Avoid Accessing Resource on Host 13](#_yhfv2bw4lwpc)

[2.3.2.2 Avoid Mounting host directories as volumes 13](#_to21iinjh9lh)

[2.3.2.3 Avoid the host’s network namespace 14](#_i4b10yj5d6r9)

[2.3.3 Linux Capabilities 14](#_hn5tvvncg5fq)

[2.3.3.1 DEFAULT Capabilities 15](#_7ws2z2wumc63)

[2.3.3.2 IPC\_LOCK 15](#_ol89s4ntg9kb)

[2.3.3.3 NET\_ADMIN 16](#_jviyd1puhldf)

[2.3.3.4 (Avoid) SYS\_ADMIN 16](#_yv1fbcv6e172)

[2.3.3.5 SYS\_NICE 16](#_ll4z1c5vnsy5)

[2.3.3.6 SYS\_PTRACE 17](#_bfzz0j2gl00d)

[2.3.4 Operations that shall be executed by OpenShift 17](#_galfiug36oyi)

[2.3.4.1 MTU setting 17](#_e515t9ikr0rm)

[2.3.4.2 Link state modification 17](#_t4yrwyms2div)

[2.3.4.3 IP/MAC address assignment 17](#_qo62xi7mesb9)

[2.3.4.4 Manipulate pod’s route table 17](#_871hrhll0gfn)

[2.3.4.5 SR/IOV VF setting 18](#_nlhzhm27nptv)

[2.3.4.6 Multicast 18](#_889b3yakpagg)

[2.3.5 Operations that can NOT be executed by OpenShift 18](#_5yvvwn2cec6t)

[2.3.6 Analyzing Your Application 19](#_cevpfmx3rv8r)

[2.3.6.1 Example 19](#_arjorek1cjdn)

[2.3.7 CNF network security 21](#_egvcb8m3sla0)

[2.3.8 Secrets management 21](#_g7ky8ze1eoyg)

[2.3.9 SCC Permissions for an Application 21](#_pcols7brgg1v)

[2.3.9.1 User-Plane CNFs 21](#_zdwsn1qasn9i)

[2.4 CNF Expectations and Permissions 25](#_s4bf2b6rnbz7)

[2.4.1 Cloud Native Design Best Practices 25](#_q5tajq68t67b)

[2.4.2 High Level CNF Expectations 26](#_ij20acf0llgj)

[2.4.3 Pod Permissions 27](#_utdf4nugggwm)

[2.4.4 Logging 27](#_ch8xx8treaqh)

[2.4.5 Monitoring 28](#_fmh1r3hgmxoc)

[2.4.6 CPU allocation 28](#_hvufl1mlncnp)

[2.4.7 Memory Allocation 28](#_7cvzsnxr7bgc)

[2.4.8 Pods 29](#_sz46badk1jxr)

[2.4.8.1 Pod Interaction/Configuration 29](#_rt85g9vi8pvf)

[2.4.8.2 Pod Exit Status 29](#_kezgn7ctrr8i)

[2.4.8.3 Graceful Termination 30](#_jm65r62l6udh)

[2.4.8.4 Pod Resource Profiles 30](#_3mnn8lq0e5o6)

[2.4.8.5 Storage: emptyDir 31](#_wdvru0n9fa5n)

[2.4.8.6 Liveness Readiness and Startup Probes 31](#_wh7a6cal947k)

[2.4.8.7 Affinity/Anti-affinity 32](#_coeuxpjw53xz)

[2.4.8.8 Upgrade expectations 33](#_yujuthqkzi3j)

[2.4.8.9 Taints and Tolerations 33](#_mh55hihj0smi)

[2.4.8.10 Requests/Limits 34](#_ktricqckyz6b)

[2.4.8.11 Use imagePullPolicy: IfNotPresent 34](#_dt5mbzrg36u1)

[2.4.8.12 Automount Services for Pods 34](#_1uoabvbdnww)

[2.4.8.13 Disruption budgets 35](#_oo0cbix8w9tx)

[2.4.8.14 No naked pods 35](#_quug73bz2pc0)

[2.4.8.15 Image tagging 36](#_6q8pxjcfkhmm)

[2.4.8.16 “One process per container” 36](#_b1mbpiic8c5v)

[2.4.8.17 init containers 36](#_xnebx5zcs5ls)

[2.4.9 Security/RBAC 37](#_90j5gq7dvnl0)

[2.4.10 Custom Role to access application CRDs 37](#_2g9casqdp7g8)

[2.4.11 MULTUS 37](#_rme8awy5zdu1)

[2.4.12 MULTUS SR-IOV / MACVLAN 38](#_2ji11fp09ihj)

[2.4.13 SR-IOV Interface settings 39](#_5rvuy7tqx9f1)

[2.4.14 Attaching the VF to a pod 41](#_y0099m4iqfh4)

[2.4.15 Discovering SR-IOV devices properties from the application 42](#_9lbjhsju5tmo)

[2.4.16 NUMA awareness 42](#_4tgsfuo5zqiq)

[2.4.17 Platform Upgrade 42](#_7v7n2jw98azb)

[2.4.18 OpenShift Virtualization/kubevirt 44](#_j0iqnrjfce3k)

[2.4.18.1 VM Image Import Recommendations (CDI) 45](#_s8wd0946ldap)

[2.4.18.2 Working with large VM disk images 46](#_ottrc7ei2hxt)

[2.4.19 Operator Best Practices 46](#_uyicx1vnfkl7)

[2.4.19.1 CNF Operator requirements 46](#_yuvccxjvhbcj)

[2.4.19.2 CNF Operator CRD Example 47](#_agcj580l7mr)

[2.4.20 Requirements for CNF 47](#_ozo9kfpokucl)

[2.4.20.1 Image Standards 48](#_lhdf3bnrnm2e)

[2.4.20.2 Universal Base Image information 49](#_pa7tq04l6qqs)

[2.4.21 Application DNS Configuration Requirements 50](#_p7eg5stjffgq)

# 1 OpenShift Platform

Certified Operator Guide - Guidelines on how to build an Operator that meets the Red Hat certification criteria

Partner Guide for Container and Operator Certification - Step-by-step instructions for partners on how to certify their images and operators

OpenShift Operator Badge Guide - Containers and Operators tests

## 1.1 Helm v3

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

https://docs.openshift.com/container-platform/4.7/cli\_reference/helm\_cli/getting-started-with-he lm-on-openshift-container-platform.html#installing-a-helm-chart-on-an-openshift-cluster\_getting started-with-helm-on-openshift

Please note that the version of OpenShift will change over time and this link only points to the current version that was in planning at the time this document was last updated.

CNF Requirement: If an application is using Helm to deploy it must use Helm v3 due to security issues with Helm v2

## 1.2 Kubernetes

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

The Kubernetes API is the mechanism with which applications and people 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.

## 1.3 CNI-OVN

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

## 1.4 Container Storage (CSI)

Pod Volumes are supported via local storage and the CSI for persistent volumes. Local storage is truly ephemeral storage, it is local only to the physical node that a pod is running on and 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 via the CSI backed by NetApp Trident.

When using storage with Kubernetes, one can leverage storage classes. Please refer to the Block Storage section 3.3.5 for a description of the available storage classes. 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/persisten t-storage-csi.html

## 1.5 Block Storage

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

There are 2 types of storage connectivity and 2 levels of storage in each. All block storage is located on a NetApp appliance.

The two types of storage connectivity are:

NFS - This is the default storage type

iSCSI - This type of storage should ONLY be used if needed by database type applications

More info on Block Volume storage support here:

https://docs.openshift.com/container-platform/4.4/storage/understanding-persistent-storage.htm l#block-volume-support\_understanding-persistent-storage

## 1.6 Ephemeral Storage

Pods and containers can require ephemeral or transient local storage for their operation. The lifetime of this ephemeral storage does not extend beyond the life of the individual pod, and this ephemeral storage cannot be shared across pods.

CNF Requirement: PODs must **NOT** place persistent data into ephemeral storage

## 1.7 Local Storage

Local storage is available on worker nodes for ephemeral storage ONLY.

CNF Requirement: PODs must **NOT** place persistent volumes on local storage

## 1.8 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 later sections for additional information on 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-onl y-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

## 1.9 CPU Manager/Pinning

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

## 1.10 Host OS

Red Hat Enterprise Linux CoreOS (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

## 1.11 Universal Base Image

https://developers.redhat.com/products/rhel/ubi/#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.

### 1.11.1 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.

#### 1.11.1.1 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.

#### 1.11.1.2 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 VNFs on as they will leverage the fully supported Red Hat ecosystem. In addition, once a VNF 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

## 1.12 Pod Security

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

## 1.13 CI/CD Framework

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

## 1.14 Kubernetes API Versions

Documentation can be found at the below links:

https://docs.openshift.com/container-platform/4.7/rest\_api/index.html

https://docs.openshift.com/container-platform/4.10/rest\_api/index.html

https://kubernetes.io/docs/reference/

CNF Requirement: All CNFs must verify that they are compliant with the correct release of REST API for Kubernetes and OpenShift. Please refer to the online documentation for deprecated APIs.

## 1.15 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.

https://docs.openshift.com/container-platform/4.7/networking/ovn\_kubernetes\_network\_provid er/about-ovn-kubernetes.html

## 1.16 User Plane Functions

### 1.16.1 Performance Addon Operator (PAO)

Red Hat created the Performance Addon Operator for low latency nodes. The emergence of Edge computing in the area of Telco / 5G plays a key role in reducing latency and congestion problems and improving application performance. Many of the deployed applications in the Telco space require low latency that can only tolerate zero packet loss. OpenShift Container Platform provides a Performance Addon Operator 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.

### 1.16.2 Hugepages

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

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-addo n-operator-for-low-latency-nodes.html#cnf-configuring-huge-pages\_cnf-master

### 1.16.3 CPU Isolation

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

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

CNF Requirement: In order to use Isolated CPUs, specific annotations must be defined in the pod spec

### 1.16.4 Topology Manager and 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 following OpenShift Documentation.

### 1.16.5 IPv4 & IPv6

Applications should discover services via DNS by doing an AAAA and A query. If an application gets a AAAA response the application should prefer using the IPv6 address in the AAAA response for application sockets.

In OpenShift 4.7, you can declare ipFamilyPolicy: PreferDualStack which will present an IPv4 and IPv6 address in the service.

CNF Recommendation: IPv4 **should** only be used inside a POD when absolutely necessary.

CNF Recommendation: Services **should** be created as IPv6 only services wherever possible. If an application requires dual stack it should create a dual stack service.

From K8S documentation:

Configure IPv4/IPv6 dual-stack

To configure IPv4/IPv6 dual-stack, set dual-stack cluster network assignments:

kube-apiserver:

--service-cluster-ip-range=<IPv4 CIDR>,<IPv6 CIDR>

### 1.16.6 VRFs (aka routing instances)

VRFs provide a way to have separate routing tables on the device enabling multiple L3 routing domains concurrently. This allows for traffic in different VRF to be treated independently of each other.

### 1.16.7 Ports reserved by OpenShift

The following ports are reserved by OpenShift and should NOT be used by any application. These ports are blocked by iptables on the nodes and traffic will not pass. Port list:

22623

22624

CNF Requirement: The following ports are reserved by OpenShift and **must NOT be used** by any application: 22623, 22624

# 2 CNF Developer Guide

This section discusses recommendations and requirements for CNF application builders.

## 2.1 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.

## 2.2 Goal

This document is mainly for the developers of CNFs, who need to build high-performance Network Functions in a containerized environment. We have created a guide that any partner can take and follow when developing their CNFs so that they can be deployed on the OpenShift Container Platform (OCP) in a secure, efficient and supportable way.

## 2.3 Non-Goal

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

### 2.3.1 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 to be in.

It is hard to move a 1000LB boulder. However, it is easy when that boulder is broken down into many pieces (pebbles). All containerized network functions (CNFs) should break apart each piece of the functions/services/processes into separate containers. These containers will still be within kubernetes pods and all of the functions that perform a single task should be within the same namespace.

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

### 2.3.2 CNF Security

In 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.

5.

CNF Requirement: Only ask for the necessary privileges and access control settings for your application

CNF Requirement: 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

CNF Requirement: **Avoid** using any host system resource

CNF Requirement: **Do not** mount host directories for device access

CNF Requirement: **Do not** use host network namespace

CNF Requirement: CNFs may **not** modify the platform in any way

#### 2.3.2.1 Avoid Accessing Resource on Host

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

#### 2.3.2.2 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.

#### 2.3.2.3 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.

CNF Requirement: Applications may **not** use NodePorts or the hostNetwork

### 2.3.3 Linux 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 thread before and after the exec system call and the file capabilities of the program being executed. Refer to the blog series in [10] for more details about Linux capabilities and some examples. For Red Hat specific review of capabilities please refer to the Linux Capabilities in OpenShift blog. An additional reference is Docker Run Reference.

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] for more details on how to define and use the SCC.

#### 2.3.3.1 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"

Note: The capabilities: "SETGID", "SETUID" &"KILL", have been removed from the OpenShift default

#### 2.3.3.2 IPC\_LOCK

IPC\_LOCK capability is required if any of these functions are used in an application: ● mlock()

● mlockall()

● shmctl()

● mmap().

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

#### 2.3.3.3 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.

CNF Requirement: Only userplane applications or applications using SR-IOV or Multicast can request NET\_ADMIN capability

#### 2.3.3.4 (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.

CNF Requirement: Applications **MUST NOT** use the SYS\_ADMIN Linux capability

#### 2.3.3.5 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 DPDK application to switch to SCHED\_FIFO.

#### 2.3.3.6 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] for more details

For more information on these capabilities refer to

https://cloud.redhat.com/blog/linux-capabilities-in-openshift

### 2.3.4 Operations that shall be executed by OpenShift

The application should not require NET\_ADMIN capability to perform the following administrative operations:

#### 2.3.4.1 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] 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] for more information.

○ For the SR/IOV interfaces managed by the SRIOV Network Operator. Refer to [6] for more information.

#### 2.3.4.2 Link state modification

○ All the links will be set up before attaching it to a pod.

#### 2.3.4.3 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] for more information.

#### 2.3.4.4 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] for more information.

○ Non-default routes can be added to pod’s routing table by various IPAM CNI plugins during pod creation

#### 2.3.4.5 SR/IOV VF setting

○ Besides the functions aforementioned, the SRIOV Network Operator supports configuring the following parameters for SRIOV VFs. Refer to [8] for more information. ■ vlan

■ linkState

■ maxTxRate

■ minRxRate

■ vlanQoS

■ spoofChk

■ trust

#### 2.3.4.6 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] [3].

○ 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] to set the correct multicast route in your pod’s routing table.

### 2.3.5 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

● SR/IOV 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

### 2.3.6 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. Find more info at [10].

Another tool is ‘capable’ which is part of the BCC tools. It can be installed on RHEL8 with “dnf install bcc”. Find more info at [11].

#### 2.3.6.1 Example

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

1. Use container\_check.stp. We can see CAP\_IPC\_LOCK and CAP\_SYS\_RAWIO are requested by ‘testpmd’ and the relevant syscalls.

$ /usr/share/systemtap/examples/profiling/container\_check.stp -c 'testpmd -l 1-2 -w 0000:00:09.0 -- -a --portmask=0x8 --nb-cores=1'

[...]

capabilities used by executables

executable: prob capability

testpmd: cap\_ipc\_lock

testpmd: cap\_sys\_rawio

capabilities used by syscalls

executable, syscall ( capability ) : count testpmd, mlockall ( cap\_ipc\_lock ) : 1 testpmd, mmap ( cap\_ipc\_lock ) : 710 testpmd, open ( cap\_sys\_rawio ) : 1 testpmd, iopl ( cap\_sys\_rawio ) : 1

forbidden syscalls

executable, syscall: count

failed syscalls

executable, syscall = errno: count eal-intr-thread, epoll\_wait = EINTR: 1 lcore-slave-2, read = : 1 rte\_mp\_handle, recvmsg = : 1 stapio, = EINTR: 1 stapio, execve = ENOENT: 3 stapio, rt\_sigsuspend = : 1 testpmd, flock = EAGAIN: 5 testpmd, stat = ENOENT: 10 testpmd, mkdir = EEXIST: 2 testpmd, readlink = ENOENT: 3 testpmd, access = ENOENT: 1141 testpmd, openat = ENOENT: 1 testpmd, open = ENOENT: 13 [...]

2. Use capable command

$ /usr/share/bcc/tools/capable

3. Start the testpmd application from another terminal, and send some test traffic to it. $ testpmd -l 18-19 -w 0000:01:00.0 -- -a --portmask=0x1 --nb-cores=1

4. Check the output of the ‘capable’ command. As we can see CAP\_IPC\_LOCK was requested for running ‘testpmd’.

[...]

00:41:58 0 3591 3591 testpmd 14 CAP\_IPC\_LOCK 1 00:41:58 0 3591 3591 testpmd 14 CAP\_IPC\_LOCK 1 00:41:58 0 3591 3591 testpmd 14 CAP\_IPC\_LOCK 1 00:41:58 0 3591 3591 testpmd 14 CAP\_IPC\_LOCK 1 00:41:58 0 3591 3591 testpmd 14 CAP\_IPC\_LOCK 1 00:41:58 0 3591 3591 testpmd 14 CAP\_IPC\_LOCK 1 00:41:58 0 3591 3591 testpmd 14 CAP\_IPC\_LOCK 1 00:41:58 0 3591 3591 testpmd 14 CAP\_IPC\_LOCK 1 00:41:58 0 3591 3591 testpmd 14 CAP\_IPC\_LOCK 1 00:41:58 0 3591 3591 testpmd 14 CAP\_IPC\_LOCK 1 00:41:58 0 3591 3591 testpmd 14 CAP\_IPC\_LOCK 1 00:41:58 0 3591 3591 testpmd 14 CAP\_IPC\_LOCK 1 00:41:58 0 3591 3591 testpmd 14 CAP\_IPC\_LOCK 1 [...]

5. Also, we can try to run ‘testpmd’ without the CAP\_IPC\_LOCK with ‘capsh’. Now we can see that the hugepage memory cannot be allocated.

$ capsh --drop=cap\_ipc\_lock -- -c testpmd -l 18-19 -w 0000:01:00.0 -- -a --portmask=0x1 --nb-cores=1

EAL: Detected 24 lcore(s)

EAL: Detected 2 NUMA nodes

EAL: Multi-process socket /var/run/dpdk/rte/mp\_socket

EAL: No free hugepages reported in hugepages-1048576kB

EAL: Probing VFIO support...

EAL: VFIO support initialized

EAL: PCI device 0000:01:00.0 on NUMA socket 0

EAL: probe driver: 8086:10fb net\_ixgbe

EAL: using IOMMU type 1 (Type 1)

EAL: Ignore mapping IO port bar(2)

EAL: PCI device 0000:01:00.1 on NUMA socket 0

EAL: probe driver: 8086:10fb net\_ixgbe

EAL: PCI device 0000:07:00.0 on NUMA socket 0

EAL: probe driver: 8086:1521 net\_e1000\_igb

EAL: PCI device 0000:07:00.1 on NUMA socket 0

EAL: probe driver: 8086:1521 net\_e1000\_igb

EAL: cannot set up DMA remapping, error 12 (Cannot allocate memory) testpmd: mlockall() failed with error "Cannot allocate memory" testpmd: create a new mbuf pool <mbuf\_pool\_socket\_0>: n=331456, size=2176, socket=0

testpmd: preferred mempool ops selected: ring\_mp\_mc

EAL: cannot set up DMA remapping, error 12 (Cannot allocate memory) testpmd: create a new mbuf pool <mbuf\_pool\_socket\_1>: n=331456, size=2176,

socket=1

testpmd: preferred mempool ops selected: ring\_mp\_mc

EAL: cannot set up DMA remapping, error 12 (Cannot allocate memory) EAL: cannot set up DMA remapping, error 12 (Cannot allocate memory)

### 2.3.7 CNF network security

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.

CNF Requirement: Applications must define network policies that permit only the minimum network access the application needs to function

### 2.3.8 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

### 2.3.9 SCC Permissions for an Application

Permissions to use a SCCs is done by adding a cluster role that has *uses* permissions for the SCC and then rolebindings for the users within a namespace to that role for users that need that SCC. Application admins can create their own role/rolebindings to assign permissions to a Service Account.

#### 2.3.9.1 User-Plane CNFs

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 may need to:

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 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].

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-2Mi: "4Gi"

requests:

openshift.io/mlxnics: "1"

memory: "1Gi"

cpu: "4"

hugepages-2Mi: "4Gi"

command: ["sleep", "infinity"]

volumes:

- name: hugepage

emptyDir:

medium: HugePages

Recommended SCC definition:

kind: SecurityContextConstraints

apiVersion: security.openshift.io/v1

metadata:

name: cnfname

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/4.7/authentication/managing-security-context-constrain ts.html

[2]

https://docs.openshift.com/container-platform/4.7/networking/openshift\_sdn/enabling-multicast.html

[3]

https://docs.openshift.com/container-platform/4.3/networking/multiple\_networks/configuring-sr-iov.ht ml#nw-configuring-high-performance-multicast-with-SRIOV\_configuring-sr-iov

[4]

https://docs.openshift.com/container-platform/4.7/networking/hardware\_networks/using-SRIOV-multica st.html

[5]

https://docs.openshift.com/container-platform/4.7/networking/multiple\_networks/understanding-multi ple-networks.html

[6]

https://docs.openshift.com/container-platform/4.7/networking/hardware\_networks/configuring-SRIOV-d evice.html

[7]

https://docs.openshift.com/container-platform/4.7/networking/multiple\_networks/attaching-pod.html# nw-MULTUS-advanced-annotations\_attaching-pod

[8]

https://docs.openshift.com/container-platform/4.7/networking/hardware\_networks/configuring-SRIOV-n et-attach.html

[9] https://kubernetes.io/docs/tasks/configure-pod-container/share-process-namespace/ [10] https://linuxera.org/capabilities-seccomp-kubernetes/

## 2.4 CNF Expectations and Permissions

### 2.4.1 Cloud Native Design Best Practices

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

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

**Single Purpose 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.

CNF Requirement: Application design should conform to cloud native design principles to the maximum extent possible.

### 2.4.2 High Level CNF Expectations

● CNFs shall be built to be cloud-native

● Containers MUST NOT run as root (uid=0). Applications that require elevated privileges will require an exception with HQ Planning

● Containers MUST run with the minimal set of permissions required. Avoid Privileged Pods.

● Use the main CNI for all traffic - MULTUS/SRIOV/MacVLAN are for corner cases only (extreme throughput requirements, protocols that are unable to be load balanced)

● CNFs should employ N+k redundancy models

● CNFs MUST define their pod affinity/anti-affinity rules.

● All secondary network interfaces employed by CNFs with the use of MULTUS MUST support Dual-Stack IPv4/IPv6.

● 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

● 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.

CNF Requirement: Containers must not run as root

CNF Requirement: All secondary interfaces (MULTUS) must support dual stack

CNF Requirement: CNFs shall not use node selectors nor taints/tolerations to assign pod location

### 2.4.3 Pod Permissions

By default, pods should not expect to be permitted 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-con straints.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 a process to validate the requirements must occur.

### 2.4.4 Logging

Log aggregation and analysis

● Containers are expected to write logs to stdout. It is highly recommended that stdout/stderr leverage some standard logging format for output.

● Logs CAN be parsed to a limited extent so that specific vendor logs can be sent back to the CNF if required.

CNFs requiring log parsing must leverage some standard logging library or format for all stdout/stderr. Examples of standard logging libraries include; klog, rfc5424, and oslo.

### 2.4.5 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 able to poll

platform level metric data.

### 2.4.6 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-performanc e-addon-operator-for-low-latency-nodes.html#performance-addon-operator-creating-workload-t hat-works-in-real-time\_cnf-master

### 2.4.7 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.

CNF Requirement: Vendors must supply quotas per project/namespace

### 2.4.8 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 are in the same Pod have access to the same mount namespace and can mount the same volumes.

#### 2.4.8.1 Pod Interaction/Configuration

Pod configurations should be created in a kubernetes native manner, the most basic example of a kubernetes native manner of configuration deployment is the use of a configmap. Configmaps can be loaded into kubernetes and pods can consume the data in a configmap by using the data in the

configmap to populate container environment variables or can be consumed as volumes in a container and read by an application.

Interaction with a running pod should be done via oc exec or oc rsh commands. This allows API RBAC to the pods and command line interaction for debugging.

CNF Requirement**:** SSH daemons must NOT be used in Openshift for pod interaction**.**

#### 2.4.8.2 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

CNF Requirement: All pods shall have a liveness, readiness and startup probes defined

#### 2.4.8.3 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

CNF Requirement: All pods must respond to SIGTERM signal and shutdown gracefully with a zero exit code.

#### 2.4.8.4 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.

#### 2.4.8.5 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.

#### 2.4.8.6 Liveness Readiness and Startup 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.

Please see upstream documentation for more details on probes:

https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-pr obes/

**Important note on using probes**: If the CNF is doing CPU pinning and running a DPDK process do not use exec probes (executing a command within the container) as it may pile up and block the node eventually.

CNF Requirement: If a CNF is doing CPU pinning, exec probes may not be used.

#### 2.4.8.7 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-affin ity.html#nodes-scheduler-pod-affinity

CNF Requirement: Pods which need to be co-located on the same node need Affinity rules and pods which should not be co-located for resiliency purposes require Anti affinity rules

CNF Requirement: Pods that perform the same microservice and that could be disrupted if multiple members of the service are unavailable must implement affinity/anti-affinity to group or spread their pods across nodes to prevent disruption in the event of node failures/patches/upgrades

#### 2.4.8.8 Upgrade expectations

● 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 should not be tied to a specific version of Kubernetes or any of its components

Applications **MUST** specify a pod disruption budget appropriately to maintain service continuity during platform upgrades. The budget should be defined with a balance such that it allows operational flexibility for the cluster to drain nodes, but restrictive enough so that the service is not degraded over upgrades.

CNF Requirement: Pods that perform the same microservice and that could be disrupted if multiple members of the service are unavailable must implement pod disruption budgets to prevent disruption in the event of patches/upgrades

#### 2.4.8.9 Taints and Tolerations

Taints and tolerations allow the Node to control toleration 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.

See

https://docs.openshift.com/container-platform/4.7/nodes/scheduling/nodes-scheduler-taints-tole rations.html for more information.

#### 2.4.8.10 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. Quotas can be used 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-projec t.html

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.htm l

CNF Requirement: Pods must define requests and limits values for CPU and memory

#### 2.4.8.11 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.

#### 2.4.8.12 Automount Services for Pods

Pods which do not require API access should set the value of automountServiceAccountToken to false within the pod spec, for example:

**apiVersion: v1**

**kind: Pod**

**metadata:**

**name: my-pod**

**spec:**

**serviceAccountName: examplesvcacct**

**automountServiceAccountToken: false**

#### 2.4.8.13 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.

#### 2.4.8.14 No naked pods

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

CNF Requirement: Applications must not depend on any single pod being online for their application to function.

CNF Requirement: Pods must be deployed as part of a Deployment or StatefulSet. CNF Requirement: Pods may not be deployed in a DaemonSet.

#### 2.4.8.15 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 (for example: latest, stable, development) and by versions within the 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-image s.html

#### 2.4.8.16 “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.

More information about pods can be found here.

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

#### 2.4.8.17 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

### 2.4.9 Security/RBAC

Roles / RoleBindings - 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. Applications may create roles and rolebindings within their namespace, however the scope of a role will be limited to the same permissions that the creator has or less.

ClusterRole / ClusterRoleBinding - A ClusterRole represents a set of permissions at the Cluster level that can be used by multiple namespaces. The ClusterRoleBinding is used for granting the permissions defined in a ClusterRole to a user or group of users at a namespace level. Applications will not be permitted to install cluster roles or create cluster role bindings, this is an administrative activity and will be done by cluster administrators. CNFs should not use cluster roles; exceptions can be granted to allow this, however it is discouraged.

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

CNF Requirement: CNFs may not create ClusterRole / ClusterRoleBinding, cluster administrators shall create them.

### 2.4.10 Custom Role to access application CRDs

If an application requires installing/deploying CRDs (Custom Resource Definitions), the application must provide a role that allows necessary permissions to create CRs within the CRDs. The custom role to access CRDs must not create any permissions to access any other API resources than the CRDs.

CNF Requirement: If an application creates CRDs it must supply a role to access those CRDs and no other API resources/permissions

### 2.4.11 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. Having additional CNIs for SR-IOV and MacVLAN interfaces 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.

CNF Requirement: Unless an application has a special traffic requirement that is not supported by SPK or ovn-kubernetes CNI the applications must use the pod network for traffic

https://docs.openshift.com/container-platform/4.7/networking/multiple-networks/understanding multiple-networks.html

### 2.4.12 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. 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.ht ml

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.

### 2.4.13 SR-IOV Interface settings

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

- The type of netdevice to be used for the VF (kernel or userspace)

- The vlan ID to be applied to a given set of VFs available in a namespace

- For kernel-space devices, the ip allocation is provided directly by the cluster ip assignment mechanism.

- The option to configure the ip of a given SR-IOV interface at runtime (see

https://docs.openshift.com/container-platform/4.7/networking/hardware\_networks/add-pod.ht ml#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\_star t":"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 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**

### 2.4.14 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"]

}

]

### 2.4.15 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": {}

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.ht](https://docs.openshift.com/container-platform/4.7/networking/hardware_networks/about-sriov.ht)

[ml#nw-SRIOV-app-netutil\_about-SRIOV](https://docs.openshift.com/container-platform/4.7/networking/hardware_networks/about-sriov.ht)).

### 2.4.16 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). Please

note that HugePages are currently not NUMA aware.

See the Performance Add-On Operator section for NUMA awareness and more information about

how HugePages are turned on.

### 2.4.17 Platform Upgrade

Openshift upgrades happen as follows:

Small example Cluster - let's name it kubey

kubey consists of:

master-0

master-1

master-2

worker-10

worker-11

worker-12

worker-13

loadbalancer-14

loadbalancer-15

In the above example cluster named kubey, there are three machine config pools: masters,

workers, loadbalancers. This is an example cluster configuration, there may be more machine

config pools based on functionality, e.g., 10 MCPs if needed.

When our cluster kubey is upgraded, the API server and etcD are updated first. So the master

config pool will be done first. Incrementally the cluster will go through and reboot master-0, 1,

2 to bring them to the new kubernetes version. After these are updated it will cycle to the next

two machine pools one at a time. Openshift will consult the maxunavilable nodes in the

machine config pool spec and reboot only as many as allowed by maxunavailable.

In a cluster as small as the above, maxUnavailable would be set to 1, so OpenShift would reboot

loadbalancer-14 and worker-10 simultaneously as they are different machineconfigpools.

Openshift will wait until worker-10 is ready before proceeding onwards to worker-11 and

continue. OpenShift will in parallel wait for loadbalancer-14 to become available again before

restarting loadbalancer-15.

In clusters larger than our kubey example, the maxUnavailable for the worker pool may be set

to a large number to reboot multiple nodes in parallel to speed up deployment of the new

version of OpenShift. This number will take into account the work loads on the cluster to make

sure sufficient resources are left to maintain application availability.

For an application to stay healthy during this process, if they are stateful at all, they should

specify a statefulset or replicatset, kubernetes by default will attempt to schedule the set

members across multiple nodes to give additional resiliency. In order to prevent kubernetes

from stealing too many nodes out from under an application, an application that has a

minimum number of pods that need to be running must specify a pod disruption budget. Pod

disruption budgets allow an application to tell kubernetes that it needs N number of pods of

said microservice alive at any given time. For example, a small stateful DB may need 2 out of

three pods available at any given time, so that application should set a pod disruption budget

with a minavailable set to a value of 2. This will allow the scheduler to know that it should not

take the second pod out of a set of 3 down at any given time during the series of node reboots.

### NOTE!!! Do NOT set your pod disruption budget to maxUnavailable <number of pods in

replica> or minUnavailable zero, operations will change your pod disruption budget to proceed

with an upgrade at the risk of your application.

A corollary to the pod disruption budget is a strong readiness and health check. A well

implemented readiness check is key for surviving these upgrades in that a pod should not

report itself ready to kubernetes until it is actually ready to take over the load from another pod

of the example set. An example of this being implemented poorly would be for a pod to report

itself ready but it is not in sync with the other DB pods in the example above. Kubernetes could

see that three of the pods are "ready" and destroy a second pod and cause disruption to the DB

leading to failure of the application served by said DB.

Reference on Pod Disruption budget:

<https://kubernetes.io/docs/tasks/run-application/configure-pdb/>

apiVersion: policy/v1beta1

kind: PodDisruptionBudget

metadata:

name: db-pod-disruption-budget

spec:

minAvailable: 2

selector:

matchLabels:

app: db

Reference on maxUnavailable:

<https://docs.openshift.com/container-platform/4.2/scalability_a>

[nd\_performance/recommended-host-practices.html#master-node-sizi](https://docs.openshift.com/container-platform/4.2/scalability_a)

[ng\_](https://docs.openshift.com/container-platform/4.2/scalability_a)

By default, only one machine is allowed to be unavailable when applying the kubelet-related

configuration to the available worker nodes. For a large cluster, it can take a long time for the

configuration change to be reflected. At any time, you can adjust the number of machines that

are updating to speed up the process.

Run:

$ oc edit machineconfigpool worker

Set maxUnavailable to the desired value.

spec:

maxUnavailable: <node\_count>

### 2.4.18 OpenShift Virtualization/kubevirt

OpenShift Virtualization and VMs (CNV) best practices

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_managem>

[ent/virt-dedicated-resources-vm.html](https://docs.openshift.com/container-platform/4.7/virt/virtual_machines/advanced_vm_managem)

[2]

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

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

[3]

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

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

[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)

#### 2.4.18.1 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.

#### 2.4.18.2 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.

### 2.4.19 Operator Best Practices

Operator best practices are currently managed here: Operator Best Practices and will

be incorporated into this document in the future.

OLM Packaged operators are a package that contains an index of all the images required to

install the operator, and the "cluster service version" which instructs openshift to create

resources as described in the cluster service version. The cluster service version is a list of the required resources that need to be created in the cluster, i.e. service accounts, crds, roles, etc

that are necessary for the operator and software that the operator installs to be successful

within the cluster.

The OLM Packaged operator will then run in openshift-operators namespace within the cluster.

Users can then utilize this operator by creating CRs within the CRDs that were created by the

operator OLM package, to deploy the software managed by the operator.

The platform administrator handles the OLM based operator installation for the users by

creating a custom catalog in the cluster that is targeted by the application. The users then

express via CRs that are consumed by the operator what they would like the operator to create

in the users namespace.

#### 2.4.19.1 CNF Operator requirements

* Operators should be certified against the openshift version of the cluster they will be

deployed on.

* + See Redhat Certification Documentation: Product Documentation for Red Hat Software Certification 8.56
  + Redhat SDK Bundle for certification: operator-sdk bundle validate
* Operators must be compatible with our version of openshift
* Operators must be in OLM bundle format (Operator Framework).
* Must be able to function without the use of openshift routes or ingress objects.
* All custom resources for operators require podspecs for both pod image override as well pod quotas.
* Operators must not use daemonsets
* The OLM operator CSV must support the “all namespaces” install method if the operator is upstream software. If the operator is a proprietary cnf operator it must support single namespaced installation. It is recommended for an operator to support all OLM install modes to ensure flexibility in our environment.
* The operator must default to watch all namespaces if the target namespace is left NULL or empty string as this is how the OLM global-operators operator group functions.
* All operator and operand images must be referenced using digest image tags "@sha256". Openshift "imagecontentsourcepolicy" objects (ICSP) only support mirror-by-digest at this time.
* For general third party upstream operators (example: mongodb), the OLM package is recommended to be located within the Red Hat registries below to support our image mirror policy
  + quay.io
  + registry.redhat.io
  + registry.connect.redhat.com
  + registry.access.redhat.com
* Operators that are proprietary to a cnf application must ensure that their CRD's are unique, and will not conflict with other operators in the cluster.
* If a cnf application requires a specific version of a third party non-proprietary operator for their app to function they will need to re-package the upstream third party operator and modify the api's so that it will not conflict with the globally installed operator version.
* Successful operator installation and runtime must be validated in pre-deployment lab environments before being allowed to be deployed to production.
* All required RBAC must be included in the OLM operator bundle so that it's managed by OLM.
* It is not recommended for a cnf application to share a proprietary operator with another cnf application if that application does not share the same version lifecycle. If a cnf application does share an operator the CRDs must be backwards compatible.

#### 2.4.19.2 CNF Operator CRD Example

### 2.4.20 Requirements for CNF

* The application MUST declare all listening ports as containerPorts in the Pod specification it provides to Kubernetes.
* The application MUST NOT listen on any other ports that are undeclared.
  + Listening ports MUST be named in the pod specification with the protocol they Implement.
    - 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.
    - Preferred prefixes: grpc , grpc-web , http , http2
    - Fallback prefixes: tcp , udp
    - Valid Example: http-webapi or grpc
* The application MUST communicate with Kubernetes Services by their service IP instead of selecting Pods in that service individually.
* The application MUST NOT encrypt outbound traffic on the cluster network interface.
* The application MUST NOT decrypt inbound traffic on the cluster network interface.
* The application SHOULD NOT manage certificates related to communication over the
* cluster network interface.
* The application MUST NOT provide nftables or iptables rules.
* The application MUST NOT define Kubernetes Custom Resources in \*.istio.io or \*.aspenmesh.io namespaces.
* The application MUST NOT define Kubernetes resources in the istio-system namespace.
* The application MUST propagate tracing headers when making outgoing requests based on incoming requests.
* 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.
* 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.
* The application MUST propagate the tracing headers by copying any header value from the original request to the new request.
* 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)
* If some or none of the headers are present, the application SHOULD NOT create them.
* If an application makes a new request and it is not in service of exactly one incoming request, it MAY omit all tracing headers.
  + Note: The application does not have to generate headers in this case. It could generate headers if it implements e.g. OpenTracing

#### 2.4.20.1 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.

#### 2.4.20.2 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 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 VNFs on as they will leverage the fully supported Red Hat ecosystem. In addition, once a VNF 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.

### 2.4.21 Application DNS Configuration Requirements

CNFs should use the service name only as a configuration parameter for attaching to a service within your namespace, the cluster will append namespace name and kubernetes service nomenclature on behalf of the application via search string in DNS. This allows a generic name for a service that works in all clusters no matter what the namespace name is and what the

cluster base FQDN is.

Kubernetes Upstream Reference for how pod/service names work in DNS:

<https://kubernetes.io/docs/concepts/services-networking/dns-pod-service/>