Cloud Native Network Function Requirements

Version 1.2
October 2020

Table of Contents

1. Introduction

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

2. Scope

This document and the current platform configuration is currently limited in scope to Wireless network elements. This document covers the requirements for tenants to run their application on Red Hat's OpenShift network functions virtualization infrastructure (NFVI) platform. While Red Hat OpenStack was initially designed to support "Direct-Port" VNFs, this document's intent is to provide guidance as the Partner community evolves their software to support containerized cloud-native applications.

3. Refactoring

NFs should break their software down into the smallest set of microservices as possible. Running monolithic applications inside of a container is not the operating model recommended by Red Hat.

It is hard to move a 1000LB boulder. However, it is easy when that boulder is broken down into many pieces (pebbles). All cloud-native network functions (CNFs) should break apart each piece of the functions/services/processes into separate containers. These containers can still be within OpenShift PODs and all of the functions that perform a single task should be within the same name space.

There is a quote that describes this best from Lewis and Fowler: "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. "

3.1. Pods

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

A Pod can contain one or more running containers at a time. Applications / containers running in the same Pod have access to several of the same 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>. Same goes for storage volumes, since the containers are in the same Pod, they have access to the same storage namespace and can both mount the same volume. Of course, care must be taken in this configuration to prevent any data corruption.

4. CNF Developer Guide

4.1. 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.

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.

Non-Goal

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

4.2. Avoid Privileged Containers

In OpenShift, it is possible to run a container as a privileged container that has all of the root capabilities of 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. Applications will not be allowed to run in privileged mode 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.

4.3. Avoid Accessing Resource on Host

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

4.3.1. Avoid Mount host directories as volumes

It is not necessary to mount host /sys/ or host /dev/ directory as a volume in pod in order to use a network device such as SR-IOV VF. The moving of network interface into pod network namespace is done by CNI automatically. 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 directly from the host. When moving network interfaces into 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 attached to container via Device Plugin automatically, there is no need to mount the /dev/ directory as a volume in container, application can find device specs under '/dev/vfio/'

4.3.2. Avoid the host's network namespace

Application pods should avoid using hostNetwork if features provided by CNI are needed, as CNI plugin is not invoked for hostNetwork pods. This includes features provided by Multus such as SR-IOV.

4.4. Capabilities

Linux Capabilities allow you to break apart the power of root into smaller groups of privileges. In OpenShift, administrators can use Security Context Constraints (SCCs) to control permissions for pods. Users can also specify the necessary Security Context in the pod annotations.

Refer to [1] for more details on how to define and use the SCC.

DEFAULT Capabilities

[default capabilities list]

```
    "CHOWN",
    "DAC_OVERRIDE",
    "FSETID",
    "FOWNER",
    "NET_RAW",
    "SETGID",
    "SETUID",
    "SETPCAP",
    "NET_BIND_SERVICE",
    "SYS_CHROOT",
    "KILL",
```

[examples of allowed operations via default capabilities]

4.4.1. IPC_LOCK

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

- mlock()
- mlockall()
- shmctl()
- mmap().

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

4.4.2. 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.

4.4.3. (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.

4.4.4. SYS_NICE

In the case that a CNF is using the real-time kernel. SYS_NICE is needed to allow DPDK application to switch to SCHED_FIFO

4.4.5. 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

4.5. Operations that shall be executed by OpenShift

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

- MTU setting
  - For the cluster network, as 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 SRIOV interfaces managed by the Sriov Network Operator. Refer to [6] for more information.
- Link state modification
    - All the links will be set to up before attaching it to a pod.
- IP/MAC address assignment
    - For all the networks, the IP/MAC address will be assigned to the interface during pod creation.
    - Multus also allows users to override the IP/MAC address. Refer to [7] for more information.
- Manipulate pod’s route table
    - By default, the default route of the pod will point to the cluster network, with or without the additional networks. Multus also allows users to override the default route of the pod. Refer to [7] for more information.
    - Non-default routes can be added to pod’s routing table by various IPAM CNI plugins during pod creation
- SRIOV VF setting
    - Besides the functions aforementioned, the SRIOV Network Operator supports to configure the following parameters for SRIOV VFs. Refer to [8] for more information.
        - vlan
        - linkState
        - maxTxRate
        - minTxRate
        - vlanQoS
        - spoofChk
        - trust
- Multicast
    - In OCP, multicast is supported for both the default interface (OVN or OpenShift-SDN) and the additional interfaces (macvlan, SR-IOV...). However, multicast is disabled by default. To enable it please refer to [2] [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.

4.6. 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. And also there are some other functionalities that are not currently supported by any of the OpenShift components. The operations require NET_ADMIN capability:

- Link state modification at runtime

- IP/MAC modification at runtime
- Manipulate pod's route at runtime
- SRIOV VF setting at runtime
- Netlink configuration
    - Take 'ethtool' as an 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
- manage firewall rules in the pod network namespace

4.7. 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 System Tap

Another tool is 'capable' which is part of the BCC tools. It can be installed on RHEL8 with "dnf install bcc". Find more info here.

4.8. 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)
```

4.9. CNF Best Practice

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

4.9.1. Control Plane and Management CNFs

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

5. Cloud-Native CNFs

There are potentially three SCC profiles that can be leveraged. These are listed in 5.1, 5.2 and 5.3.

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

This kind of CNFs shall

1. Uses the default CNI (OVN) network interface.
2. Not request ‘NET_ADMIN’ or ‘NET_RAW’ for advance networking functions.

Recommended SCC definition (default):

```
kind: SecurityContextConstraints
apiVersion: security.OpenShift.io/v1
metadata:
  name: cnf-catalog-1
users: []
groups: []
priority: null
allowHostDirVolumePlugin: false
allowHostIPC: false
allowHostNetwork: false
allowHostPID: false
allowHostPorts: false
allowPrivilegeEscalation: true
allowPrivilegedContainer: false
allowedCapabilities: null
defaultAddCapabilities: null
requiredDropCapabilities:
- KILL
- MKNOD
- SETUID
- SETGID
- NET_RAW
fsGroup:
  type: MustRunAs
readOnlyRootFilesystem: false
runAsUser:
  type: MustRunAsRange
seLinuxContext:
  type: MustRunAs
supplementalGroups:
  type: RunAsAny
```

```
volumes:
- configMap
- downwardAPI
- emptyDir
- persistentVolumeClaim
- projected
- secret
```

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

The CNFs with following characteristics may fall into this category:

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

This kind of CNF may:

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

Recommended SCC definition:

```
kind: SecurityContextConstraints
apiVersion: security.OpenShift.io/v1
metadata:
  name: cnf-catalog-2
users: []
groups: []
priority: null
allowHostDirVolumePlugin: false
allowHostIPC: false
allowHostNetwork: false
allowHostPID: false
allowHostPorts: false
allowPrivilegeEscalation: true
allowPrivilegedContainer: false
```

```
allowedCapabilities: [NET_ADMIN, NET_RAW]
defaultAddCapabilities: null
requiredDropCapabilities:
- KILL
- MKNOD
- SETUID
- SETGID
fsGroup:
  type: MustRunAs
readOnlyRootFilesystem: false
runAsUser:
  type: MustRunAsRange
seLinuxContext:
  type: MustRunAs
supplementalGroups:
  type: RunAsAny
volumes:
- configMap
- downwardAPI
- emptyDir
- persistentVolumeClaim
- projected
- secret
```

5.3. User-Plane CNFs (Category 3)

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

This kind of CNF shall:

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

   NOTE: *For 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 here

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

Recommended SCC definition:

```
kind: SecurityContextConstraints
apiVersion: security.OpenShift.io/v1
```

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

[1] https://docs.OpenShift.com/container-platform/4.4/authentication/managing-security-context-constraints.html

[2] https://docs.OpenShift.com/container-platform/4.3/networking/OpenShift_sdn/using-multicast.html

[3]
https://docs.OpenShift.com/container-platform/4.3/networking/multiple_networks/configuring-sr-iov.html#nw-configuring-high-performance-multicast-with-sriov_configuring-sr-iov
[4]
https://docs.OpenShift.com/container-platform/4.4/installing/installing_bare_metal/installing-bare-metal-network-customizations.html
[5]
https://docs.OpenShift.com/container-platform/4.4/networking/multiple_networks/understanding-multiple-networks.html
[6]
https://docs.OpenShift.com/container-platform/4.4/networking/hardware_networks/configuring-sriov-device.html
[7]
https://docs.OpenShift.com/container-platform/4.4/networking/multiple_networks/attaching-pod.html#nw-multus-advanced-annotations_attaching-pod
[8]
https://docs.OpenShift.com/container-platform/4.4/networking/hardware_networks/configuring-sriov-net-attach.html

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

6. CNF Expectations and Permissions

6.1. Cloud Native Design Best Practices

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

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.

6.2. High Level CNF Expectations

- CNFs shall be built to be cloud-native
- Do not run containers as root by default - Applications that require elevated privileges will require an exception with HQ Planning
- Privileged pods require a security review providing an analysis of the special permissions required. Avoid Privileged Pods. If Privileged Pods are required, the CNF developer should work with the planning department, and Redhat to determine acceptability of privilege and/or modifications to pods such that elevated privilege is not required.
- Use the main CNI for all traffic
- CNF should leverage service mesh provided by the platform for internal & external communication
- CNFs should leverage platform service mesh for mTLS with other applications
- Pod Security Policy will be dictated by the security team - See Pod Permissions Section
- Naming and Labelling standards for all OpenShift objects (Pods, Services, etc.) will be provided by the planning team
- CNFs should employ N+k redundancy models
- CNFs must define their pod affinity/anti-affinity rules
- implementation in Services Proxy/Load Balancer for Ingress & Egress traffic. CNFs shall support this requirement.
- 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 K8s-native constructs such as ReplicaSets, etc.
- CNFs shall support canary upgrades employing the platform Service Mesh
- CNFs shall self-recover from common failures like pod failure, host failure, and network failure. OpenShift native mechanisms such as health-checks (Liveness, Readiness and Startup Probes) shall be employed at a minimum.

6.3. Platform Restrictions

- CNFs may not deploy Nodeports
- CNFs may not use host networking
- Namespace creation will be performed by the platform team and should not be created by the CNFs deployment method (Helm / Operator)
- CNFs may not perform Role creation
- CNFs may not perform Rolebinding creation
- CNFs may not have Cluster Roles
- CNFs are not authorized to bring their own CNI
- CNFs may not deploy Daemonsets
- CNFs may not modify the platform in any way

Core/Edge

7. OpenShift Platform

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

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

8. Software Core/Edge

8.1. CaaS

8.1.1. Helm v3

Helm v3 is a serverless mechanism for pushing templates that describe a complete OpenShift deployment to a cluster. This allows a template to be built for an application and site/deployment specific values to be provided as input to the template such that multiple deployments can be made in multiple locations. It is roughly analogous to HEAT templates in the OpenStack environment.

https://docs.OpenShift.com/container-platform/4.4/cli_reference/helm_cli/getting-started-with-helm-on-OpenShift-container-platform.html#installing-a-helm-chart-on-an-OpenShift-cluster_getting-started-with-helm-on-OpenShift

8.1.2. 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 and applications interact with the cluster. There are several ways to do this, via the kubectl or oc CLI tools, via web based UIs or interact 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.

8.1.3. 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.

8.1.4. 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 pv and pvc to get an NFS share via the CSI backed by NetApp Trident.

When using storage with Kubernetes, one can leverage storage classes. These allow you to classify storage by capabilities. For example, if you have fast storage that can be one class that production applications can use. If you have slower, cheaper storage that may be a storage class that your developers can use for testing.

Network Functions should clear persistent storage when removing their application from a cluster.

Red Hat Persistent Storage Documentation

https://docs.OpenShift.com/container-platform/4.4/storage/container_storage_interface/persistent-storage-csi.html

8.1.4.1. Block Storage

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

More info on Block Volume storage support here:

https://docs.OpenShift.com/container-platform/4.4/storage/understanding-persistent-storage.html#block-volume-support_understanding-persistent-storage

8.1.5. Container Runtime

OpenShift uses CRIO as a CRI interface for Kubernetes. CRIO manages runC for container image execution. CRI-O is an open-source container engine that provides a stable, performant platform for running OCI compatible runtimes. CRI-O is developed, tested and released in tandem with Kubernetes major and minor releases. Images should be OCI compliant. Images are recommended to be built using Red Hat's open Universal Base Image. See section 8.1.8 for additional information UBI and support.

Red Hat Documentation on CRIO and Read Only Containers
https://blog.OpenShift.com/add-a-layer-of-security-to-OpenShift-kubernetes-with-cri-o-in-read-only-mode/

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

This environment is maintained through the open source tools:

- runc
- skopeo
- buildah
- podman
- crio

8.1.6. CPU Manager/Pinning

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

8.1.7. Host OS

OpenShift will run Red Hat CoreOS on a bare metal environment. There is no hypervisor layer between the containers and the host OS. RHCOS is the next generation container operating system. RCHOS is part of the OpenShift Container Platform and is used as the OS for the Control plane and can also be used for the 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 immutable nature of RHCOS does not support installing RPMs or additional packages in the traditional way. Some 3rd party services or functionalities need to run as agents on nodes of the cluster.

For more information on RHCOS please refer to the following link.

Red Hat Enterprise Linux CoreOS | Architecture | OpenShift Container Platform 4.4

8.1.8. Universal Base Image

https://developers.redhat.com/products/rhel/ubi/#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 Partners so there is a low barrier of entry to getting started.

There are three types of Red Hat UBI: Standard, Minimal, and Multi-Service.

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

9. PaaS Core/Edge

9.1. Distributed Tracing

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

9.2. Pod Security

SELinux will be enabled within the platform and will be used to enforce syscalls that containers make. In addition, OpenShift has another native function called pod security policies. A Pod Security Policy is a cluster-level resource that controls security sensitive aspects of the pod specification. The PodSecurityPolicy objects define a set of conditions that a pod must run with in order to be accepted into the system, as well as defaults for the related fields. Some examples of what you can control with PodSecurityPolicy are:

Volume types
Host filesystems
Linux capabilities
Much much more

9.3. OpenShift API versions

OpenShift supports the full Kubernetes APIs as well as additional API calls that are OpenShift specific. Documentation can be found at the below link:
https://docs.OpenShift.com/container-platform/4.4/rest_api/index.html

10. Pod Permissions

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/4.4/authentication/managing-security-context-constraints.html

Pods will execute on worker nodes where pods will run with default SCC which is the "restricted" SCC

The "restricted" SCC:

- Ensures that pods cannot run as privileged.
- Ensures that pods cannot mount host directory volumes.
- Requires that a pod run as a user in a pre-allocated range of UIDs.
- Requires that a pod run with a pre-allocated MCS label.
- Allows pods to use any FSGroup.
- Allows pods to use any supplemental group.

Any pods requiring elevated privileges must document the required capabilities driven by application syscalls and an exception process to validate the requirements must occur. Upon approval of an exception a custom SCC can be created limiting the syscalls down to only the needed syscals that will be associated with the NFs namespace for the particular PODs that require elevated privileges.

11. OpenShift Best Practices

11.1. Logging

Log aggregation and analysis

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

- Partners are expected to write logs to stdout
- It is possible to deploy an instance of Fluentd which will parse app logs and redirect to logstash listener
- Requires Partner to follow pod/container naming standards
- A logstash listener can be deployed within each site
- Logs will be forwarded to a centralized storage location
- Logs CAN be parsed so that specific Partner logs can be sent back to the CNF if required
- Requires Partner to provide svc/fqdn
- It is suggested to send back logs to the matching namespace using the below tag format
    - vendor-function-000.logs ← Logs for namespace 000
    - vendor-function-001.logs ← Logs for namespace 001

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

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

11.2. Monitoring

Network Functions are expected to bring their own metrics collection functions (i.e. Prometheus) for their application specific metrics. This metrics collector will not be expected to nor able to poll platform level metric data.

11.3. CPU allocation

It is important to note that when the OpenShift scheduler is placing pods, it first reviews the Pod CPU "Request" and schedules it if there is a node that meets the requirements. It 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.

Requesting CPU pinning

OpenShift provides a topology manager which leverages the CPU manager and Device manager to help associate processes to CPUs. The topology manager has different policies that can be applied such as none, best effort, restricted, and single-numa-node. This feature is tech preview as of OpenShift 4.4. For some examples on how to leverage the topology manager, see the following link.

https://docs.OpenShift.com/container-platform/4.4/scalability_and_performance/using-topology-manager.html

11.4. 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.

11.5. Affinity/Anti-affinity

In OpenShift, 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: docker.io/ocpqe/hello-pod
```

https://docs.OpenShift.com/container-platform/4.4/nodes/scheduling/nodes-scheduler-pod-affinity.html#nodes-scheduler-pod-affinity

11.6. Taints and Tolerations

Taints and tolerations allow the Node to control which Pods should (or should not) be scheduled on them. A taint allows a node to refuse a pod to be scheduled unless that pod has a matching toleration.
You apply taints to a node through the node specification (NodeSpec) and apply tolerations to a pod through the pod specification (PodSpec). A taint on a node instructs the node to repel all pods that do not tolerate the taint.
Taints and tolerations consist of a key, value, and effect. An operator allows you to leave one of these parameters empty.

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

See
https://docs.OpenShift.com/container-platform/4.4/nodes/scheduling/nodes-scheduler-taints-tolerations.html for more information.

11.7. Requests/Limits

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

https://docs.OpenShift.com/container-platform/4.4/applications/quotas/quotas-setting-per-project.html

Do not plan to overcommit resources in the production environment. It is possible to overcommit node resources in development environments. See below notes on overcommitment:

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/4.4/nodes/clusters/nodes-cluster-overcommit.html

11.8. Pods

11.8.1. No naked pods

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

11.8.2. Image tagging

An image tag is a label applied to a container image in a repository that distinguishes a specific image from other images. Image tags may

https://docs.OpenShift.com/container-platform/4.4/OpenShift_images/managing_images/tagging-images.html

11.8.3. “One process per container”

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

When possible, try to map a single container to a pod. This can help in a number of ways, troubleshooting, upgrades, efficient scaling.

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

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

Keep in mind when using multiple containers in a single pod that when debugging, you need to attach to a specific container with a -c flag. This will allow you to pull logs, and connect to a specific container.

More information about pods can be found here.

https://docs.OpenShift.com/container-platform/4.4/nodes/pods/nodes-pods-using.html

11.8.4. init containers

Init containers can be used for running tools / commands / or any other action that needs to be done before the actual pod is started. E.g: Load a database schema, create some folder, write a config file, etc.

https://docs.OpenShift.com/container-platform/4.4/nodes/containers/nodes-containers-init.html

11.9. Security/RBAC

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

ClusterRole / ClusterRoleBinding - A ClusterRole represents a set of permissions at the Cluster level. E.g: A given user is admin on the cluster. The ClusterRoleBinding is used for granting the permissions defined in a ClusterRole to a user or group of users. Do not plan to allow cluster roles for network functions.

https://docs.OpenShift.com/container-platform/4.4/authentication/using-rbac.html

11.10. Multus

Multus is a meta CNI that allows multiple CNIs that it delegates to. This allows pods to get additional interfaces beyond eth0 via additional CNIs. OpenShift may have additional CNIs for SR-IOV and MacVLAN interfaces. This allows 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 corner cases include, bandwidth requirements that necessitate SR-IOV and protocols that are unable to be supported by the load balancer. The OVN based pod network should be used for every interface that can be supported from a technical standpoint.

https://docs.OpenShift.com/container-platform/4.4/networking/multiple-networks/understanding-multiple-networks.html

11.10.1. Multus SR-IOV / MACVLAN

SR-IOV and MACVLAN interfaces are able to be requested for protocols that do not work with the default CNI or for exceptions where a network function has not been able to move functionality onto the CNI. These are exception use cases. Multus interfaces will be defined by the platform operations team for the network functions which can then consume them. Multus interfaces will have to be requested via the planning tools ahead of time by your personnel. VLANs will be applied by the

SR-IOV VF, thus the VLAN/network that the SR-IOV interface requires must be part of the request for the namespace.

https://docs.OpenShift.com/container-platform/4.4/networking/hardware_networks/about-sriov.html

By configuring the SR-IOV Network CRs exposed by the SR-IOV Operator (https://docs.OpenShift.com/container-platform/4.2/networking/multiple_networks/configuring-sr-iov.html#configuring-sr-iov-networks_configuring-sr-iov) , named Network Attachment Definitions are created 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. In case any is available, get in touch with the cluster administrator.

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:

SR-IOV currently has a limitation of assigning IP addresses to interfaces so SR-IOV interfaces need an additional annotation for the IP and MAC address of the pod.

Example:

```
apiVersion: v1
kind: Pod
metadata:
 annotations:
  k8s.v1.cni.cncf.io/networks: '[
    {
        "name": "ens3f0nicvf2",
        "mac": "20:20:20:20:20:01"
      }
    ]'
```

11.11. Upgrades

11.11.1. Handling platform upgrades

- CNF Partners should expect that OpenShift shall be upgraded to new versions at customer sites on an ongoing basis employing CI/CD runtime deployment without any advance notice to CNF Partners.
- During OpenShift platform upgrades, the Kubernetes API deprecation policy defined in https://kubernetes.io/docs/reference/using-api/deprecation-policy/ shall be followed
- CNFs are expected to maintain service continuity during Platform Upgrades, and during CNF version upgrades.
- CNFs need to be prepared for nodes to reboot and go down.
- CNFs shall configure pod disruption budget appropriately to maintain service continuity during platform upgrades
- Applications **may NOT** deploy pod disruption budgets that prevent zero pod disruption.
- Applications should not be tied to a specific version of Kubernetes or any of its components

11.12. CNV/kubevirt

11.12.1. OpenShift Virtualization and VMs (CNV) best practices

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

CNV should be installed according to its documentation, and use only documented supported features, unless an explicit exception is listed below.

In order to improve overall virtualization performance and reduce CPU latency, critical VNFs can take advantage of CNV'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://kubevirt.io/user-guide/#/creation/dedicated-cpu?id=virtualmachineinstance-with-dedicated-cpu-resources
[2]
https://kubevirt.io/user-guide/#/creation/dedicated-cpu?id=requesting-dedicated-cpu-for-qemu-emulator
[3]
https://kubevirt.io/user-guide/#/creation/disks-and-volumes?id=iothreads-with-qemu-emulator-thread-and-dedicated-pinned-cpus
[4] https://kubevirt.io/user-guide/#/creation/cloud-init?id=device-role-tagging

11.12.2. VM Image Import Recommendations (CDI)

CNV 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. CNV provides a component called the Containerized Data Importer (CDI) which automates the preparation of pre-populated persistent disks for VMs. CDI integrates with KubeVirt to synchronize VM creation and deletion with disk preparation by using a custom resource called a DataVolume. Using DataVolumes, data can be imported into a PV from various sources including container registries and HTTP servers.

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

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

Working with large VM disk images

In contrast to container images, VM disk images can be quite large (30GiB or more is common). It is important to consider the costs of transferring large amounts of data when planning workflows involving the creation of VMs (especially when scaling up the number of VMs). The efficiency of an image import depends on the format of the file and also the transfer method used. The most efficient workflow, for two reasons, is to host a gzip-compressed raw image on a server and import via HTTP. Compression avoids transferring zeros present in the free space of the image, and CDI can stream the contents directly into the target PV without any intermediate conversion steps. In contrast, images imported from a container registry must be transferred, unarchived, and converted prior to being usable. These additional steps increase the amount of data transferred between a node and the remote storage.

12. Operator Best Practices

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

13. Container Best Practices

13.1. 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 OpenShift. One way is the pod can remain alive but non-functional. Another way is 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 perhaps write to logs that will 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

13.2. Graceful Termination

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

13.3. Pod Resource Profiles

OpenShift comes with a default scheduler that is responsible for being aware of the current available resources on the platform, and placing containers / applications on the platform appropriately. 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 (if they aren't using integral CPUs)

13.4. 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.

Use emptyDir with requested local storage limits instead of writing to the container directories

13.5. Liveness and Readiness Probes

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

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

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

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

13.6. Use imagePullPolicy: IfNotPresent

If there is a situation where the container dies and needs to be restarted, the image pull policy becomes important. There are three image pull policies available: always, never and ifNotPresent. It is generally recommended to have a pull policy of ifNotPresent. This means that the pod gets stopped, starts and looks at the policy, and if the policy is ifNotPresent the kubelet will check on the node where the pod is starting and if it has been downloaded before it will not pull a new one but keep using the one that exists. The reason for this is that it could be possible if the image pull policy is set to always that it will pull the latest image available which hasn't necessarily gone through the proper checks - and run that on the platform. Not to mention the amount of traffic that could be generated in the case where many containers need to be pulled at once due to a policy of always.

13.7. Automount Services for Pods

Set automountServiceAccountToken: false on all pods unless the pod needs to access the OpenShift API

13.8. Disruption budgets

When managing the platform there are at least two types of disruptions that can occur. They are voluntary and involuntary. When dealing with voluntary disruptions a pod disruption budget can be set that determines how many replicas of the application must remain running at any given time. For example, consider the case where an administrator is shutting down a node for maintenance and the node has to be drained. If 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 and then draining the current node.

14. Networking Overview

OpenShift can be a multi-tenant environment. NFs will be deployed within a single namespace. Supporting applications like an OAM platform for multiple NFs from the same Partner should be run in an additional separate namespace.

Multus will be supported within the platform for additional NICs within containers. However Multus should be used only for corner cases that cannot be supported by the load balancer.

The POD and Services networks are unrouted address space. The POD network will be NATed as traffic egresses the load balancer. Traffic inbound will be destination NATed to Service/Pod IP addresses.

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

14.1. OVN-kubernetes CNI

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

14.2. IPv4 & IPv6

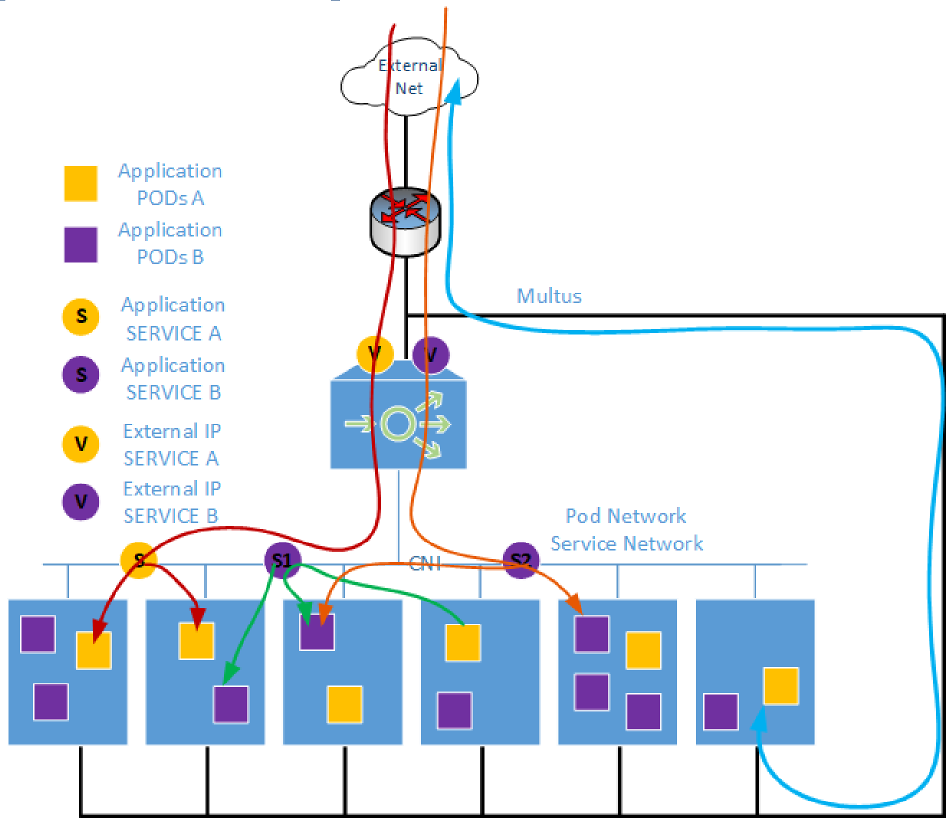
Dual stack will be supported in early 2021 in production.

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.

14.3. Routing Instances

Routing Instances 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.

15. Application Service Exposure to External Networks



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

The second mechanism of service delivery depicted by the green line is to make a service available within the platform to either an application itself (internally to the application) or to other applications within the platform.

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

16. Service Mesh for Inter/Intra NF

16.1. Service Mesh Introduction

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

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

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

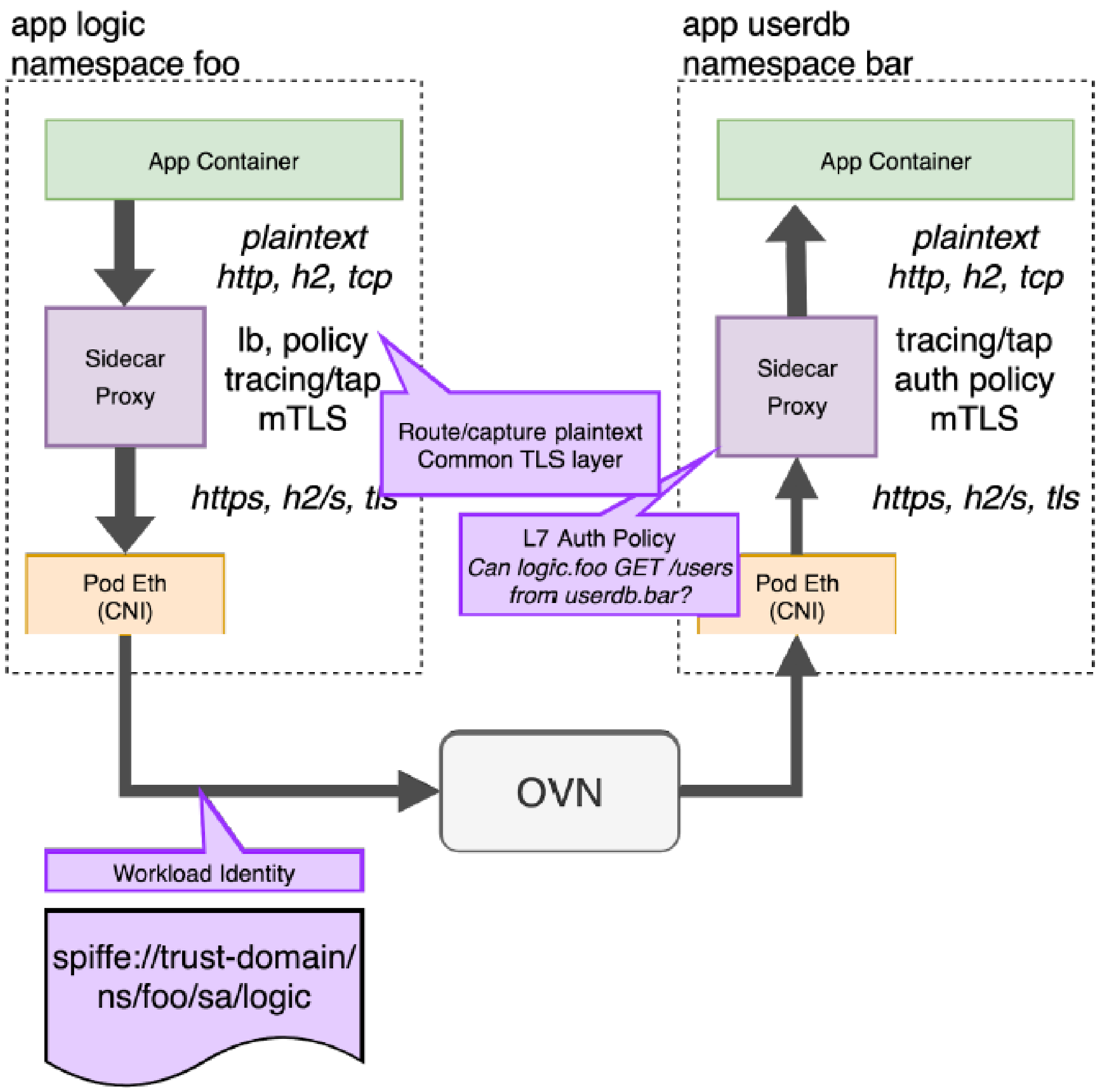
The sidecar is transparently injected into the Kubernetes pod without requiring any intervention by the main application container. Application containers do not need to communicate in any special way with the sidecar directly. Generally, application containers should be unaware of the presence of the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

Service Mesh is also capable of doing CSR generation, signing and installation into the namespace as part of the solution. This offloads all PKI efforts from the CNF and also makes the certificates available for NFs within the application via kubernetes secrets as a volume mount should an application require the keys for doing any TLS via Multus based interfaces.

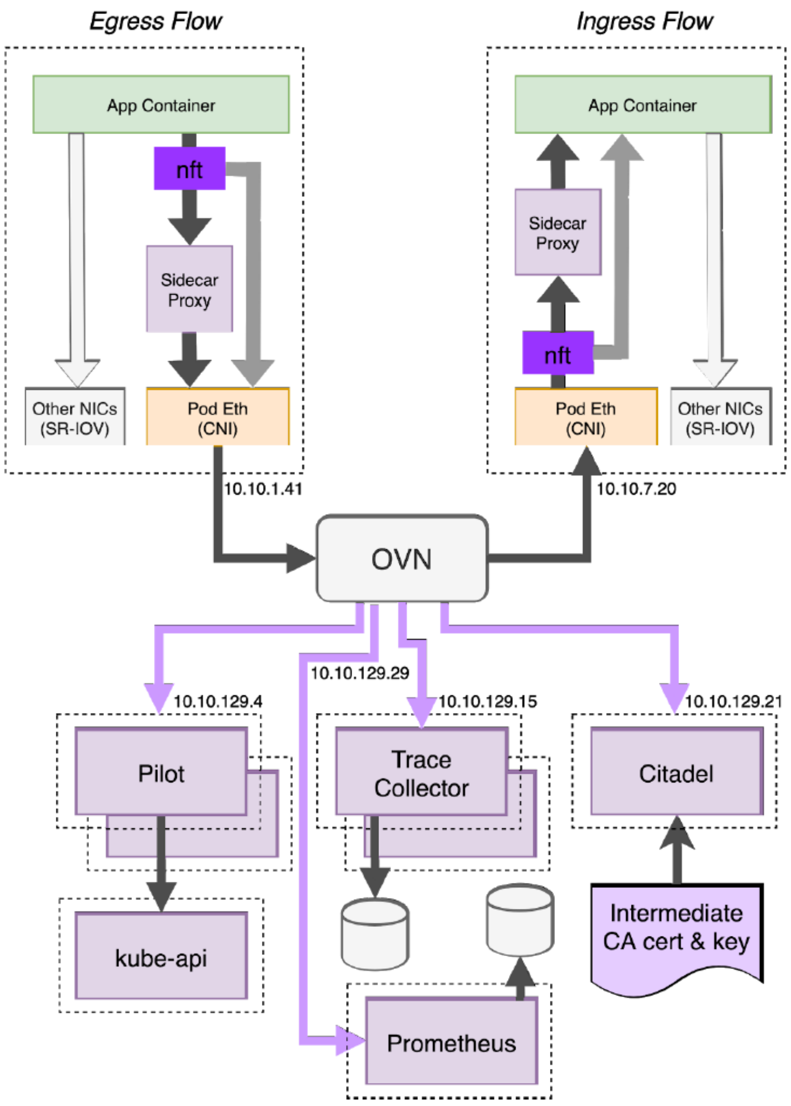
Additionally Service Mesh allows distributed tracing for HTTP based flows that can be analyzed via the Jaeger UI. This provides a consistent visibility mechanism across NFs for SBI based communication.

The following diagram depicts Service Mesh at a high level.

Service Mesh High Level Overview



Service Mesh Detailed View



This diagram depicts the full suite of components that are involved with Service Mesh delivery. Pilot is the Istio Control plane, the sidecar proxies are Envoy based proxies. Trace collector and prometheus provide statistics and traces. Not depicted is Jaeger which allows viewing of traces.

16.2. Service Mesh Tapping

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

16.3. Service Mesh Requirements for CNF

16.3.1. The application MUST declare all listening ports as containerPorts in the Pod specification it provides to Kubernetes.

16.3.1.1. The application MUST NOT listen on any other ports that are undeclared.

16.3.1.2. The service mesh MAY be configured to block connections to these ports.

16.3.1.3. These ports MUST be named in the pod specification with the protocol they implement.

16.3.1.3.1. The name field in the ContainerPort section must be of the form <protocol>[-<suffix>] where <protocol> is one of the below, and the optional <suffix> can be chosen by the application.

16.3.1.3.2. Preferred prefixes: grpc , grpc-web , http , http2

16.3.1.3.3. Fallback prefixes: tcp , udp

16.3.1.3.4. Valid Example: http-webapi or grpc

16.3.2. The application MUST communicate with Kubernetes Services by their service IP instead of selecting Pods in that service individually.

16.3.2.1. The service mesh will select the appropriate pod.

16.3.3. The application MUST NOT encrypt outbound traffic on the cluster network interface.

16.3.3.1. The service mesh will apply policy, authenticate servers and encrypt outbound traffic before it leaves the application pod.

16.3.4. The application MUST NOT decrypt inbound traffic on the cluster network interface.

16.3.4.1. The service mesh will decrypt, authenticate clients and apply policy before redirecting traffic to the application container.

16.3.5. The application SHOULD NOT manage certificates related to communication over the cluster network interface.

16.3.5.1. The service mesh will manage, rotate and validate these certificates.

16.3.6. The application MUST NOT provide nftables or iptables rules.

16.3.7.

16.3.8. The application MUST NOT define Kubernetes Custom Resources in these namespaces:

16.3.8.1. *.istio.io

16.3.8.2. *.aspenmesh.io

16.3.9. The application MUST NOT define Kubernetes resources in the istio-system namespace.

16.3.10. The application MUST propagate tracing headers when making outgoing requests based on incoming requests.

16.3.10.1. Example: If an application receives a request with a trace header identifying that request with traceid 785a908c8d93b2d2 , and decides based on application logic that it must make a new request to another application pod to fulfill that request, it must annotate the new request with the same traceid 785a908c8d93b2d2.

16.3.10.2. The application MUST propagate the tracing headers by copying any header value from the original request to the new request.

16.3.10.3. 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)

16.3.10.4. If some or none of the headers are present, the application SHOULD NOT create them.

16.3.10.5. If an application makes a new request and it is not in service of exactly one incoming request, it MAY omit all tracing headers.

16.3.10.5.1. Note: The application does not have to generate headers in this case. It could generate headers if it implements e.g. OpenTracing, and the service mesh would use and propagate those IDs. This is optional.

16.3.10.5.2. If there are no tracing headers, the service mesh will generate a new trace (detailed below).

17. Standards

17.1. Container Naming Standards

OpenShift is targeting a container naming standard to enable tools to determine CNF and its components.

Container Labeling Standards

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

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

17.2. Image Standards

It is recommended that applications be built utilizing Red Hat's Universal Base Image as container images built utilizing this UBI image are guaranteed to operate on an OpenShift cluster and have support from Red Hat as well as a solid security baseline.

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

- Work with Red Hat’s Restricted SCC[1]
- Work with Red Hat’s Default SELinux Context[1]
- 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.
    - Inherits certification such as FIPS from the base OS
    - 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 Partner 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 Partner’s released version.

[2] RedHat UBI meets these requirements

17.2.1. Universal Base Image information

https://developers.redhat.com/products/rhel/ubi/#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 CNFs on as they will leverage the fully supported Red Hat ecosystem. In addition, once a CNF is standardized on a Red Hat UBI, the image can be become Red Hat certified.

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

There are three types of Red Hat UBI: Standard, Minimal, and Multi-Service.

18. Security

18.1. Elevated privilege container capabilities

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

18.2. Image Security

Images may be scanned by OpenShift CVE scanners while stored in Red Hat's Internal Registry. Vulnerabilities found during scanning will result in flags and deployment of images with vulnerabilities will require exceptions.

19. Contributors

Name	Title	Email	Area of Contribution

20. Document History

Version	Date	Change	Version POC

21. Document approvals

Name	Title	Company	Date of approval