# Oracle Cloud Native Environment

## Kubernetes for Release 2

F96196-07
March 2026

Oracle Cloud Native Environment Kubernetes for Release 2,

F96196-07

# Contents

# Preface

This book describes how to use Kubernetes, which is an implementation of the open source, containerized application management platform from the upstream Kubernetes release. Oracle provides extra tools, testing, and support to deliver this technology with confidence. Kubernetes integrates with container products to handle more complex deployments where clustering might be used to improve the scalability, performance, and availability of containerized applications. Detail is provided on the basic features of Kubernetes and how it can be used in Oracle Cloud Native Environment (Oracle CNE).

This document describes functionality and usage available in the most current release of the product.

1

# Introduction to Kubernetes

Introduces Kubernetes in Oracle Cloud Native Environment (Oracle CNE).

Kubernetes is an open source system for automating the deployment, scaling, and management of containerized applications. Kubernetes provides the tools to create a cluster of systems across which containerized applications can be easily deployed and scaled as required.

The Kubernetes project is maintained at:

https://kubernetes.io/

Kubernetes is fully tested on Oracle Linux and includes extra tools developed at Oracle to ease configuration and deployment of a Kubernetes cluster.

2

# Kubernetes Architecture

The version of Kubernetes used in Oracle CNE is based on the upstream Kubernetes project, and released under the CNCF Kubernetes Certified Conformance program. Oracle Container Host for Kubernetes (OCK) simplifies the configuration and set up of Kubernetes to create a Kubernetes cluster and includes backup. Kubernetes integrates with CRI-O to provide a comprehensive container and orchestration environment for the delivery of microservices and next-generation application development.

The core component in Oracle CNE is Kubernetes. Kubernetes includes:

- **Flannel**: The default overlay network for a Kubernetes cluster.
- **CoreDNS**: The DNS server for a Kubernetes cluster.
- **CRI-O**: Manages the container runtime for a Kubernetes cluster.
- **runC**: The default lightweight, portable container runtime for a Kubernetes cluster.
- **Kata Containers**: An optional lightweight virtual machine runtime for a Kubernetes cluster.

For more information about using Kubernetes, see Oracle Cloud Native Environment: Kubernetes.

## Network Planes

This chapter contains information about the Oracle CNE management, control, and data planes.

### Management Plane

Communication between the components is secured using Transport Layer Security (TLS). You can configure the cipher suites to use for TLS for the management plane.

You can set up the X.509 certificates used for TLS before you create a cluster, or use private, automatically generated, certificates.

### Control Plane

The control plane contains the Kubernetes components and any load balancer.

Kubernetes has a sophisticated networking model with many options that lets users finely tune the networking configuration. Oracle CNE simplifies the Kubernetes networking by setting network defaults that align with community best practices. By default, all Kubernetes services are bound to the network interface that handles the default route for the system. The default route is used for both the Kubernetes control plane and the data plane.

### Data Plane

The data plane is the network used by the pods running on Kubernetes.

The same algorithm to decide the default control plane interface is used when instantiating the Kubernetes pod network. The network interface is used for both the Kubernetes control plane

and the data plane. In environments with many networks, this might not be the best choice. Oracle CNE lets you customize the network interface used for pod networking when you create the Kubernetes module. When the CNI is brought up, it uses the network interface you specify for the pod network.

# Storage

Every meaningful workload in the computing industry requires some sort of data storage. Persistent storage is essential when working with stateful applications such as databases, as it's important that you can retain data beyond the lifecycle of the container, or even of the pod itself.

Persistent storage in Kubernetes is handled in the form of PersistentVolume objects which are bound to pods using a PersistentVolumeClaim. You can host a PersistentVolume locally or on networked storage devices or services.

A typical Kubernetes environment involves many hosts and includes some type of networked storage. Using networked storage helps to ensure resilience and lets you take full advantage of a clustered environment. In the case where the node where a pod is running fails, a new pod can be started on another node and storage access can be resumed. This is important for database environments where replica setup has been configured.

## Cloud Native Storage

Several storage projects are associated with the CNCF foundation, and the providers are included by default in Kubernetes. Storage integration is provided using plugins, referred to as the Container Storage Interface (CSI). The plugins adhere to a standard specification.

## Persistent Storage

Persistent storage is provided in Kubernetes using the PersistentVolume subsystem. To configure persistent storage, you must be familiar with the following terms:

- **PersistentVolume**

  A PersistentVolume defines the type of storage that's being used and the method used to connect to it. This is the real disk or networked storage service that's used to store data.

- **PersistentVolumeClaim**

  A PersistentVolumeClaim defines the parameters that a consumer, such as a pod, uses to bind the PersistentVolume. The claim might specify quota and access modes to be applied to the resource for a consumer. A pod can use a PersistentVolumeClaim to gain access to the volume and mount it.

- **StorageClass**

  A StorageClass is an object that specifies a volume plugin, known as a provisioner, that lets users to define PersistentVolumeClaims without needing to preconfigure the storage for a PersistentVolume. This can be used to provide access to similar volume types as a pooled resource that can be dynamically provisioned for the lifecycle of a PersistentVolumeClaim.

PersistentVolumes can be provisioned either statically or dynamically.

Static PersistentVolumes are manually created and contain the details required to access real storage and can be consumed directly by any pod that has an associated PersistentVolumeClaim.

Dynamic PersistentVolumes can be automatically generated if a PersistentVolumeClaim doesn't match an existing static PersistentVolume and an existing StorageClass is requested in the claim. A StorageClass can be defined to host a pool of storage that can be accessed dynamically. Creating a StorageClass is an optional step that's only required if you intend to use dynamic provisioning.

The process to provision persistent storage is as follows:

1. Create a PersistentVolume or StorageClass.
2. Create PersistentVolumeClaims.
3. Configure a pod to use the PersistentVolumeClaim.

The process for adding and configuring NFS and iSCSI volumes is described in detail in the upstream documentation.

## Container Storage Interface Plugins

The Container Storage Interface (CSI) is an Open Container Initiative standard for controlling storage workloads from container engines. Kubernetes implements this interface to provide automated control for storage workloads inside Kubernetes clusters. For a list of the Kubernetes storage provisioners, see the upstream documentation.

# Cloud Native Networking

The Container Network Interface (CNI) project, incubating under CNCF, seeks to simplify networking for container workloads by defining a common network interface for containers. The CNI plugin is included with Oracle CNE.

# Highly Available Clusters

This chapter contains high level information about the types of highly available (HA) Kubernetes clusters you can deploy using Oracle CNE.

Kubernetes can be deployed with more than one replica of the control plane node. Automated failover to those replicas provides a more scalable and resilient service. This type of cluster deployment is referred to in this document as an HA cluster.

> ⚠ **Caution**
>
> To create an HA cluster you need at least three control plane nodes and two worker nodes.

Creating an HA cluster with three control plane nodes ensures replication of configuration data between them through the distributed key store, `etcd`, so an HA cluster is resilient to a single control plane node failing without any loss of data or up time. If more than one control plane node fails, you can restore the control plane nodes in the cluster from a backup file to avoid data loss.

Oracle CNE implements the Kubernetes stacked `etcd` topology, where `etcd` runs on the control plane nodes. For more information on this topology, see the upstream documentation.

## Load Balancer

An HA cluster needs a load balancer to provide high availability of the control plane nodes. A load balancer communicates with the Kubernetes API Server to maintain high availability of the control plane nodes.

You can use an external load balancer instance, or have the CLI install a load balancer on the control plane nodes.

### Highly Available Cluster with External Load Balancer

When you set up an HA cluster to use an external load balancer, the load balancer is used to manage the resource availability and efficiency of control plane nodes to ensure instances of the Kubernetes API Server on control plane nodes can fail without impacting cluster availability.

To use an external load balancer, it must be set up and ready to use before you perform an HA cluster deployment. The load balancer hostname and port is entered as an option with the CLI when you create the Kubernetes cluster.

### Highly Available Cluster with Internal Load Balancer

When you set up an HA cluster to use an internal load balancer, the CLI installs NGINX and Keepalived on the control plane nodes as a systemd service to enable the deployment of the load balancer. The internal load balancer configures the native active-active high availability solution for the Kubernetes API Server.

NGINX improves the resource availability and efficiency of control plane nodes to ensure instances of the Kubernetes API Server on control plane nodes can fail without impacting cluster availability.

If you use the internal load balancer, you must set aside an IP address on the control plane network to use as a virtual IP address. The Keepalived instance makes sure the virtual IP address is always reachable by monitoring the health of other control plane nodes and appropriating the IP address if a node fails. Keepalived is used to fail over automatically to a standby control plane node if problems occur.

# Container Runtimes

Containers are the fundamental infrastructure to deploy modern cloud applications. Oracle delivers the tools to create and provision Open Container Initiative (OCI)-compliant containers using CRI-O.

CRI-O, an implementation of the Kubernetes CRI (Container Runtime Interface) to enable using Open Container Initiative compatible runtimes, is included with Oracle CNE. CRI-O can run either runC or Kata Containers containers directly from Kubernetes, without any unnecessary code or tooling.

# Authentication

Standard X.509 certificates are used to establish node identity and authentication. Kubernetes nodes require a valid certificate chain for each component to mutually authenticate. Without these certificates, connections between the components and nodes are rejected.

# 3
# Kubernetes Components

Learn about the components in a Kubernetes cluster.

This section outlines the common components of Kubernetes within Oracle CNE. The descriptions provided are brief, and largely intended to help provide a glossary of terms and an overview of the architecture of a typical Kubernetes environment. Upstream documentation can be found at:

https://kubernetes.io/docs/concepts/

## Nodes

Introduces the node types in a Kubernetes cluster.

Kubernetes Node architecture is described in detail at:

https://kubernetes.io/docs/concepts/architecture/nodes/

### Control Plane Nodes

Describes Kubernetes control plane nodes.

The control plane node is responsible for cluster management and for exposing the API that's used to configure and manage resources within the Kubernetes cluster. Kubernetes control plane node components can be run within Kubernetes itself, as a set of containers within a dedicated pod. These components can be replicated for High Availability (HA) of the control plane nodes.

The following components are required for a control plane node:

- **API Server** (`kube-apiserver`): The Kubernetes REST API is exposed by the API Server. This component processes and validates operations and then updates information in the Cluster State Store to trigger operations on the worker nodes. The API is also the gateway to the cluster.
- **Cluster State Store** (`etcd`): Configuration data relating to the cluster state is stored in the Cluster State Store, which can roll out changes to the coordinating components such as the Controller Manager and the Scheduler. It's important to have a backup plan in place for the data stored in the Cluster State Store.
- **Cluster Controller Manager** (`kube-controller-manager`): This manager is used to perform many cluster-level functions and overall application management, based on input from the Cluster State Store and the API Server.
- **Scheduler** (`kube-scheduler`): The Scheduler automatically decides where to run containers by monitoring availability of resources, quality of service, and affinity specifications.

The control plane node can be configured as a worker node within the cluster. Therefore, the control plane node also runs the standard node services: the `kubelet` service, the container runtime, and the `kube-proxy` service. Note that it's possible to *taint* a node to prevent workloads from running on an inappropriate node. The `kubeadm` utility automatically taints the

control plane node so that no other workloads or containers can run on this node. This ensures that the control plane node is never placed under any unnecessary load and simplifies the backup and restore of the control plane node.

If the control plane node becomes unavailable for a period, the ability to change the cluster state is suspended but the worker nodes continue to run container applications without interruption.

For single node clusters, when the control plane node is offline, the API is unavailable, so the environment is unable to respond to node failures and no new operations that affect the overall cluster state, such as creating new resources or editing or moving existing resources, can be performed.

An HA cluster, with several control plane nodes, ensures that more requests for control plane node functionality can be handled, and control plane replica nodes help improve cluster uptime.

## Control Plane Replica Nodes

Describes Kubernetes control plane replica nodes.

Control plane replica nodes are responsible for duplicating the functionality and data contained on control plane nodes within a Kubernetes cluster configured for HA. To improve uptime and resilience, you can host control plane replica nodes in different zones, and configure them to load balance for the Kubernetes cluster.

Replica nodes are designed to mirror the control plane node configuration and the current cluster state in real time. If the control plane nodes become unavailable, the Kubernetes cluster can fail over to the replica nodes automatically. If a control plane node fails, the API remains available so that the cluster can continue to respond automatically to other node failures and service requests for creating and editing existing resources within the cluster.

## Worker Nodes

Describes Kubernetes worker nodes.

Worker nodes within the Kubernetes cluster are used to run containerized applications and handle networking to route traffic between applications within and outside of the cluster. The worker nodes perform any actions triggered by the Kubernetes API, which runs on the control plane node.

All nodes within a Kubernetes cluster must run the following services:

- **Kubelet Service** (`kubelet`): The agent that controls communication between each worker node and the API Server running on the control plane node. This agent is also responsible for managing pod tasks, such as mounting volumes, starting containers, and reporting status.
- **Container Runtime**: An environment where containers can be run. In this release, the container runtimes are either runC or Kata Containers. For more information about the container runtimes, see Creating Kata Containers.
- **Kube Proxy Service** (`kube-proxy`): A service that translates service definitions to networking rules. These handle port forwarding and IP redirects to ensure that network traffic from outside the pod network can be transparently proxied to the pods in a service.

In all cases, these services are run from `systemd` as daemons.

# Pods

Describes Kubernetes pods.

Kubernetes introduces the concept of *pods*, which are groupings of one or more containers and their shared storage, and any specific options on how these are to be run together. Pods are used for tightly-coupled applications that would typically run on the same logical host and which might require access to the same system resources. Typically, containers in a pod share the same network and memory space and can access shared volumes for storage. These shared resources enable the containers in a pod to communicate internally in a seamless way as if they were installed on a single logical host.

You can easily create or destroy pods as a set of containers. This makes it possible to do rolling updates to an application by controlling the scaling of the deployment. You can scale up or down easily by creating or removing replica pods. For more information on pods, see the upstream Kubernetes documentation.

# ReplicaSet, Deployment, and StatefulSet Controllers

Describes Kubernetes ReplicaSets, Deployments, and StatefulSet Controllers.

Kubernetes provides various controllers that define how pods are set up and deployed within the Kubernetes cluster. These controllers can be used to group pods together according to their runtime needs, control the order in which the pods start up, and configure pod replication.

Define a set of pods to be replicated with a *ReplicaSet*, which provides the configuration for each of the pods in the group and which resources they have access to. Using ReplicaSets makes it easy to scale or reschedule an application and perform rolling or multi track updates to an application. For more information on ReplicaSets, see the upstream Kubernetes documentation.

Use a *Deployment* to manage pods and *ReplicaSet*s. *Deployment*s make it easy to roll out changes to ReplicaSets, or rollback to an earlier *Deployment* revision. Create a newer revision of a *ReplicaSet* with a *Deployment* and then migrate existing pods from a previous *ReplicaSet* into the new revision. The *Deployment* can then manage the cleanup of older unused *ReplicaSet*s. For more information on Deployments, see the upstream Kubernetes documentation.

Use *StatefulSet*s to create pods that guarantee start up order and unique identifiers, which are then used to ensure that the pod maintains its identity across the life cycle of the *StatefulSet*. This feature makes it possible to run stateful applications within Kubernetes, as the persistence of components such as storage and networking are guaranteed. Furthermore, when you create pods they're always created in the same order and allocated identifiers that are applied to host names and the internal cluster DNS. Those identifiers ensure stable and predictable network identities for pods in the environment. For more information on StatefulSets, see the upstream Kubernetes documentation.

# Services

Describes Kubernetes services.

Use services to expose access to one or more mutually interchangeable pods. As pods can be replicated for rolling updates and for scalability, clients accessing an application must be directed to a pod running the correct application. Pods might also need access to applications

outside of Kubernetes. In either case, you can define a service to make access to these resources transparent, even if the actual backend changes.

Typically, services consist of port and IP mappings. How services function in network space depends on the service type.

The default service type is `ClusterIP`, which exposes the service on the internal IP of the cluster, so that the service is reachable only from within the cluster. Use this service type to expose services for applications that need to access each other from within the cluster.

Often, clients outside of the Kubernetes cluster might need access to services within the cluster. Use the `NodePort` service type for this. This service type takes advantage of the Kube Proxy service that runs on every worker node and reroutes traffic to a `ClusterIP`, which is created automatically along with the `NodePort` service. The service is exposed on each node IP at a static port, called the `NodePort`. The Kube Proxy routes traffic destined to the `NodePort` into the cluster to be serviced by a pod running inside the cluster. This means that if a `NodePort` service is running in the cluster, it can be accessed from any node in the cluster, regardless of where the pod is running.

Building on top of these service types, the `LoadBalancer` service type can expose a service externally by using a cloud provider's load balancer. The external load balancer can handle redirecting traffic to pods directly in the cluster from the Kube Proxy. A `NodePort` service and a `ClusterIP` service are automatically created when a `LoadBalancer` service is provisioned.

> ⚠ **Caution**
>
> When adding services for different pods, ensure that the network is configured appropriately for each service declaration. Any external-facing ports exposed by a `NodePort` or `LoadBalancer` service must also be accessible through any firewalls running on the nodes.

For more information on services, see the upstream Kubernetes documentation.

# Volumes

Describes Kubernetes volumes.

In Kubernetes, a *volume* is storage that persists across the containers within a pod for the lifespan of the pod itself. When a container within the pod is restarted, the data in the Kubernetes volume is preserved. Kubernetes volumes can be shared across containers within the pod, providing a file store that different containers can access locally.

Kubernetes provides various volume types that define how the data is stored and how it's persisted, which are described in detail in the upstream Kubernetes documentation.

The lifetime of a Kubernetes volume typically matches the lifetime of the pod, and data in a volume persists while the pod using that volume exists. Containers in the pod can be restarted and the data remains intact. However, if the pod is destroyed, the data is usually destroyed with it.

For situations in which the volume data must outlive the pod, Kubernetes introduces the concepts of the *PersistentVolume* and the *PersistentVolumeClaim*.

*PersistentVolume*s are similar to *Volume*s except that they exist independently of a pod. They define how to access a storage resource type, such as NFS, or iSCSI. Configure a *PersistentVolumeClaim* to use the resources available in a *PersistentVolume*. The

*PersistentVolumeClaim* specifies the quota and access modes to be applied to the resource for a consumer. Use the *PersistentVolumeClaim* to gain access to these resources with the appropriate access modes and size restrictions applied.

# Namespaces

Describes Kubernetes namespaces.

Kubernetes implements and maintains strong separation of resources by using namespaces. Namespaces effectively run as virtual clusters on the same physical cluster and are intended for use in environments where Kubernetes resources must be shared across use cases.

Kubernetes uses namespaces to separate cluster management and specific Kubernetes controls from user-defined resources. All the pods and services specific to the Kubernetes system are found within the `kube-system` namespace. All other deployments for which no namespace has been specified run in the `default` namespace.

For more information on namespaces, see the upstream Kubernetes documentation.

# CRI-O

Describes CRI-O, an implementation of the Kubernetes Container Runtime Interface.

When you deploy Kubernetes worker nodes, CRI-O is also deployed. CRI-O is an implementation of the Kubernetes Container Runtime Interface (CRI) to enable using Open Container Initiative (OCI) compatible runtimes. CRI-O is a lightweight alternative to using Docker as the runtime for Kubernetes. With CRI-O, Kubernetes can use any OCI-compliant runtime as the container runtime for pods.

CRI-O delegates containers to run on appropriate nodes, based on the configuration set in pod files. *Privileged* pods can be run using the runC runtime engine (`runc`), and *unprivileged* pods can be run using the Kata Containers runtime engine (`kata-runtime`). The status of containers as trusted or untrusted is configured in the Kubernetes pod or deployment file.

For information on how to set the container runtime, see Creating Kata Containers.

# 4
# Using Kubernetes

Learn how to use Kubernetes and the Kubernetes CLI (`kubectl`).

This chapter describes how to get started using Kubernetes to deploy, maintain, and scale containerized applications. In this chapter, we describe basic usage of the `kubectl` command to get you started creating and managing containers and services within a cluster.

The `kubectl` utility is fully documented in the upstream Kubernetes documentation.

## Getting Node Information

Use the `kubectl` command to display information about nodes in a Kubernetes cluster.

1. List all cluster nodes.

   To list all nodes in a cluster and the status of each node, use the `kubectl get` command. This command can be used to list any kind of Kubernetes resource. The following example lists resources that are `nodes`:

   ```
   kubectl get nodes
   ```

   The output looks similar to:

   ```
   NAME                  STATUS   ROLES           AGE   VERSION
   ocne-control-plane-1  Ready    control-plane   1h    version
   ocne-worker-1         Ready    <none>          1h    version
   ocne-worker-2         Ready    <none>          1h    version
   ```

2. Get details about resources.

   You can get more detailed information about any resource using the `kubectl describe` command. If you specify the name of the resource, the output is limited to information about that specific resource. Otherwise, details of all resources are displayed. For example, to get detailed information about a specific node:

   ```
   kubectl describe nodes ocne-worker-1
   ```

   The output looks similar to:

   ```
   Name:               ocne-worker-1
   Roles:              <none>
   Labels:             beta.kubernetes.io/arch=amd64
                       beta.kubernetes.io/os=linux
                       kubernetes.io/arch=amd64
                       kubernetes.io/hostname=ocne-worker-1
                       kubernetes.io/os=linux
   Annotations:        flannel.alpha.coreos.com/backend-data:
   {"VNI":1,"VtepMAC":"3a:41:1a:ce:e0:d0"}
                       flannel.alpha.coreos.com/backend-type: vxlan
   ```

```
                    flannel.alpha.coreos.com/kube-subnet-manager: true
                    flannel.alpha.coreos.com/public-ip: 192.168.122.130
                    kubeadm.alpha.kubernetes.io/cri-socket:
unix:///var/run/crio/crio.sock
                    node.alpha.kubernetes.io/ttl: 0
                    volumes.kubernetes.io/controller-managed-attach-
detach: true
...
```

# Running an Application in a Pod

Use the `kubectl` command to create a pod running an NGINX container in a Kubernetes cluster.

1. Create a pod.

   To create a pod with a single running container, use the `kubectl create` command. For example, to create a pod with the `nginx` container image from the Oracle Container Registry:

   ```
   kubectl create deployment --image container-registry.oracle.com/olcne/
   nginx:1.17.7 hello-world
   ```

   This example uses `hello-world` as the name for the deployment. The pods are named by using the deployment name as a prefix.

   > **Tip**
   >
   > Deployment, pod, and service names conform to a requirement to match a DNS-1123 label. These must consist of lowercase alphanumeric characters or `-`, and must start and end with an alphanumeric character. The regular expression that's used to validate names is `[a-z0-9]([-a-z0-9]*[a-z0-9])?`. If you use a name for the deployment that doesn't validate, an error is returned.

   Many more optional parameters can be used when you run a new application within Kubernetes. For example, at run time, you can specify how many replica pods are to be started, or you might apply a label to the deployment to make it easier to identify pod components. To see a full list of options available to you, use the `kubectl run --help` command.

2. List the pods.

   To check that a new application deployment has created one or more pods, use the `kubectl get pods` command:

   ```
   kubectl get pods
   ```

   The output looks similar to:

   ```
   NAME                           READY   STATUS    RESTARTS   AGE
   hello-world-5db76fbd7d-99s8h   1/1     Running   0          1m
   ```

3. Show details about a pod.

Use `kubectl describe` to show a more detailed view of pods, including which containers are running and what image they're based on, including which node is hosting the pod:

```
kubectl describe pods
```

The output looks similar to:

```
Name:             hello-world-5db76fbd7d-99s8h
Namespace:        default
Priority:         0
Service Account:  default
Node:             <nodename>/<IP_address>
Start Time:       <date> 11:08:37 +0000
Labels:           app=hello-world
                  pod-template-hash=5db76fbd7d
Annotations:      <none>
Status:           Running
IP:               10.244.1.26
IPs:
  IP:           10.244.1.26
Controlled By:  ReplicaSet/hello-world-5db76fbd7d
Containers:
  nginx:
    Container ID:   cri-o://
6f4ce80153cefbaea327dd011b035cfb2112eb31085ca358b9c894fa775...
    Image:          container-registry.oracle.com/olcne/nginx:1.17.7
    Image ID:       container-registry.oracle.com/olcne/
nginx@sha256:78ce89068e7feb15ec...
    Port:           <none>
    Host Port:      <none>
    State:          Running
      Started:      <date> 11:08:43 +0000
    Ready:          True
    Restart Count:  0
    Environment:    <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from kube-api-access-
d9q4t (ro)
Conditions:
  Type                        Status
  PodReadyToStartContainers   True
  Initialized                 True
  Ready                       True
  ContainersReady             True
  PodScheduled                True
Volumes:
  kube-api-access-d9q4t:
    Type:                    Projected (a volume that contains injected
data
                                                                from multiple
sources)
    TokenExpirationSeconds:  3607
    ConfigMapName:           kube-root-ca.crt
    ConfigMapOptional:       <nil>
    DownwardAPI:             true
```

```
QoS Class:                   BestEffort
Node-Selectors:              <none>
Tolerations:                 node.kubernetes.io/not-ready:NoExecute
op=Exists for 300s
                             node.kubernetes.io/unreachable:NoExecute
op=Exists for 300s
Events:
...
```

# Scaling a Pod Deployment

Use the `kubectl` command to scale a deployment in a Kubernetes cluster.

1. Scale the replicas in a deployment.

   To change the number of instances of the same pod that you're running, you can use the `kubectl scale deployment` command. For example:

   ```
   kubectl scale deployment --replicas=3 hello-world
   ```

2. Check the deployment is scaled.

   You can check that the number of pod instances has been scaled appropriately:

   ```
   kubectl get pods
   ```

   The output looks similar to:

   ```
   NAME                           READY   STATUS    RESTARTS   AGE
   hello-world-5db76fbd7d-99s8h   1/1     Running   0          3m46s
   hello-world-5db76fbd7d-g6lrm   1/1     Running   0          15s
   hello-world-5db76fbd7d-h496h   1/1     Running   0          15s
   ```

# Deleting a Service or Deployment

Use the `kubectl` command to delete a service or deployment in a Kubernetes cluster.

Objects can be deleted easily within Kubernetes so that the environment can be cleaned up. Use the `kubectl delete` command to remove an object.

To delete an entire deployment, and all pod replicas running for that deployment, specify the `deployment` object and the name that you used to create the deployment. For example:

```
kubectl delete deployment hello-world
```

# Viewing Pods in Namespaces

Use the `kubectl` command view pods in a namespace in a Kubernetes cluster.

Namespaces can be used to further separate resource usage and to provide limited environments for particular use cases. By default, Kubernetes configures a namespace for Kubernetes system components and a standard namespace called `default` for all other deployments for which no namespace is defined.

To view existing namespaces, use the `kubectl get namespaces` and `kubectl describe namespaces` commands.

The `kubectl` command only displays resources in the `default` namespace, unless you specify a different namespace. For example, to view the pods specific to the Kubernetes system, use the `--namespace` option to set the namespace to `kube-system`:

```
kubectl get pods --namespace kube-system
```

The output looks similar to:

```
NAME                                          READY   STATUS    RESTARTS
AGE
coredns-f7d444b54-bw446                       1/1     Running   0
63m
coredns-f7d444b54-tsx8v                       1/1     Running   0
63m
etcd-ocne-control-plane-1                     1/1     Running   0
63m
kube-apiserver-ocne-control-plane-1           1/1     Running   0
63m
kube-controller-manager-ocne-control-plane-1  1/1     Running   0
63m
kube-proxy-ksl7l                              1/1     Running   0
63m
kube-proxy-lzdmr                              1/1     Running   0
62m
kube-proxy-t942q                              1/1     Running   0
62m
kube-scheduler-ocne-control-plane-1           1/1     Running   0
63m
```

5

# Creating Kata Containers

Learn about using Kata Containers in Kubernetes.

By default, containers are created using the default `runc` runtime engine. You can also use the `kata-runtime` runtime engine to create Kata Containers.

Kata Containers uses lightweight Virtual Machines (VMs) to provide extra security and isolation of workloads. Kata Containers are quick to develop and deploy, and plug directly into the container ecosystem.

Kata Containers is a component of Oracle CNE and based on a stable release of the upstream Kata Containers project. Differences between Oracle versions of the software and upstream releases are limited to Oracle-specific fixes and patches for specific bugs.

For upstream Kata Containers documentation, see:

https://github.com/kata-containers/documentation

For more information about Kata Containers, see:

https://katacontainers.io/

## Checking Hardware

Check whether the hardware can run Kata Containers using the `kata-runtime kata-check` command. This requires a running Kubernetes deployment.

1. Start an administration console on a worker node.

   Some operations described in this and later steps must be run directly on a worker node, using an administration console. You can start an administration console on any Kubernetes node using the `ocne cluster console` command. The syntax is:

   ```
   ocne cluster console
   [{-d|--direct}]
   {-N|--node} nodename
   [{-t|--toolbox}]
   [-- command]
   ```

   For more information on the syntax options, see Oracle Cloud Native Environment: CLI.

   Start an administration console on any worker node, `chroot`ed to the root of the node's file system. Run the following command, replacing `ocne-worker-1` with the name of a worker node:

   ```
   ocne cluster console --direct --node ocne-worker-1
   ```

2. Run the `kata-check` command on the worker node.

At the administration console prompt on the worker node, run the following command:

```
sudo kata-runtime kata-check
```

For more information on using the `kata-runtime` command, use the `kata-runtime --help` command.

3. Exit the administration console on the worker node.

   Exit the administration console on the worker node by typing `exit` at the console prompt.

   ```
   exit
   ```

# Setting Runtime Classes

Create a `kata-runtime` runtime class to run Kata Containers in a Kubernetes cluster.

CRI-O uses the Kubernetes `RuntimeClass` resource in the pod configuration file to specify whether to run a pod using the default runtime `runc`, or using `kata-runtime`.

The examples in this book use the name `native` to specify the use of `runc`, and the name `kata-containers` to specify the use of the Kata Containers runtime, but the names aren't important.

1. Create a runtime class file for Kata Containers.

   Create a file for a runtime class for Kata Containers named `kata-runtime.yaml` with the following contents:

   ```
   kind: RuntimeClass
   apiVersion: node.k8s.io/v1
   metadata:
       name: kata-containers
   handler: kata
   ```

2. Load the runtime class file.

   Load the runtime class to the Kubernetes deployment:

   ```
   kubectl apply -f kata-runtime.yaml
   ```

   The runtime class `kata-containers` can now be used in pod configuration files to specify that a container is to be run as a Kata Container, using the `kata-containers` runtime. For examples of creating pods using this runtime class, see Creating Kata Containers.

3. (Optional) Create a runtime class file for `runc`.

   Use the same approach to specify a runtime for `runc`. This is an optional configuration step. As `runc` is the default runtime, pods automatically run using `runc` unless you specify otherwise. This file is named `runc-runtime.yaml`:

   ```
   kind: RuntimeClass
   apiVersion: node.k8s.io/v1
   metadata:
       name: native
   handler: runc
   ```

4. (Optional) Load the runtime class file.

   Load the runtime class to the Kubernetes deployment:

   ```
   kubectl apply -f runc-runtime.yaml
   ```

   The runtime class `native` can be used in pod configuration files to specify that a container runs as a runC container, using the `runc` runtime.

5. List the runtime classes.

   You can see a list of the available runtime classes for a Kubernetes cluster using the `kubectl get runtimeclass` command:

   ```
   kubectl get runtimeclass
   ```

   The output looks similar to:

   ```
   NAME              HANDLER   AGE
   kata-containers   kata      7m29s
   native            runc      7m7s
   ```

# Creating Kata Containers

Create an NGINX pod that runs as a Kata Container using a Kubernetes `RuntimeClass`.

This task shows how to create a container using `kata-runtime` as the runtime engine. To create Kata Containers, set up a Kubernetes `RuntimeClass` resource for `kata-runtime`. For information on setting up a `RuntimeClass`, see Setting Runtime Classes.

This example uses a Kubernetes pod configuration file to create a Kata Container running an NGINX web server.

1. Create pod configuration file.

   On a host that's set up to use the `kubectl` command to connect to the Kubernetes cluster, create a Kubernetes pod configuration file. Use the notation `runtimeClassName: kata-containers` in the pod file. When CRI-O finds this runtime class in a pod file, it uses `kata-runtime` to run the container.

   This pod file is named `kata-nginx.yaml`.

   ```
   apiVersion: v1
   kind: Pod
   metadata:
     name: kata-nginx
   spec:
     runtimeClassName: kata-containers
     containers:
       - name: nginx
         image: container-registry.oracle.com/olcne/nginx:1.17.7
         ports:
         - containerPort: 80
   ```

2. Start the pod.

Create the Kata Container using the `kata-nginx.yaml` file with the `kubectl apply` command:

```
kubectl apply -f kata-nginx.yaml
```

3. Verify the pod is running.

   To check the pod has been created, use the `kubectl get pods` command:

   ```
   kubectl get pods
   ```

   The output looks similar to:

   ```
   NAME         READY   STATUS    RESTARTS   AGE
   kata-nginx   1/1     Running   0          40s
   ```

4. Show more information about the pod.

   Use the `kubectl describe` command to show a more detailed view of the pod, including the runtime, which worker node is hosting the pod, and the Container ID.

   ```
   kubectl describe pod kata-nginx
   ```

   The output looks similar to:

   ```
   Name:               kata-nginx
   Namespace:          default
   Priority:           0
   Runtime Class Name: kata-containers
   Service Account:    default
   Node:               ocne-worker-1/<IP_address>
   Start Time:         Wed, 23 Oct 2024 12:07:35 +0000
   Labels:             <none>
   Annotations:        <none>
   Status:             Running
   IP:                 10.244.1.29
   IPs:
     IP:  10.244.1.29
   Containers:
     nginx:
       Container ID:   cri-o://
   ca0559ab7c77deddb2a5baf681fff39ae620a5a0696ee4535ad53fff...
       Image:          container-registry.oracle.com/olcne/nginx:1.17.7
       Image ID:       container-registry.oracle.com/olcne/
   nginx@sha256:78ce89068e7feb1...
       Port:           80/TCP
       Host Port:      0/TCP
       State:          Running

   ...
   ```

5. Start an administration console on the worker node running the Kata Container pod.

You can start an administration console on any Kubernetes node using the `ocne cluster console` command. The syntax is:

```
ocne cluster console
[{-d|--direct}]
{-N|--node} nodename
[{-t|--toolbox}]
[-- command]
```

For more information on the syntax options, see Oracle Cloud Native Environment: CLI.

Start an administration console on the worker node running the `kata-container` pod identified in the output of the previous step, by entering the following command, replacing the name of the node as appropriate:

```
ocne cluster console --direct --node ocne-worker-1
```

6. List the pods running on a worker node.

   List the pods running on a worker node using the `crictl pods` command by running the following command at the administration console prompt:

   ```
   sudo crictl pods
   ```

   The output looks similar to:

   ```
   POD ID         CREATED         STATE  NAME
   NAMESPACE       ...
   02ab970089cd1  11 seconds ago  Ready  console-ocne-worker-1...    ocne-
   system     ...
   52af794c70dce  4 minutes ago   Ready  kata-nginx
   default         ...
   430c83360e934  6 days ago      Ready  control-plane-capi-cont...  capi-
   kubeadm-con...
   ac94aebe63b51  6 days ago      Ready  bootstrap-capi-controll...  capi-
   kubeadm-boo...
   ...
   ```

   You can see the `kata-nginx` container is running on this worker node.

   For more information on using the `crictl` command, use the `crictl --help` command.

7. List details about the containers running on a worker node.

   To get more detailed information about the containers on a worker node, use the `crictl ps` command. For example:

   ```
   sudo crictl ps
   ```

   The output looks similar to:

   ```
   CONTAINER       IMAGE          ... NAME                              POD
   ID        ...
   43d8e4fba2698   9a7fadacb497dbc... console-ocne-worker-1
   2e4655ea682e5 ...
   ```

```
ca0559ab7c77d   ...nginx@sha256... nginx
52af794c70dce ...
1556b7459a2be   container-regis... olcne/kubeadm-control-plane-cont
430c83360e934 ...
...
```

Note the Container ID is a shortened version of the `Container ID` shown in the pod description.

8. List more details about a pod.

   To get detailed information about a pod, run the `crictl inspectp` command using the `POD ID`. For example:

   ```
   sudo crictl inspectp 52af794c70dce
   ```

   The output looks similar to:

   ```
   {
     "status": {
       "id":
   "52af794c70dce199e1bdab40b9dfe196def5a791266240a11e3477ea66b1421e",
       "metadata": {
         "attempt": 0,
         "name": "kata-nginx",
         "namespace": "default",
         "uid": "331dc2b0-769b-4a5e-b1eb-a521f8c75670"
       },
       "state": "SANDBOX_READY",
       "createdAt": "<date>",
       "network": {
         "additionalIps": [],
         "ip": "<IP_address>"
       },
   ...
   ```

9. Exit the administration console.

   Exit the administration console on the worker node by typing `exit` at the console prompt.

   ```
   exit
   ```

10. Delete the pod.

    You can delete the pod using the `kubectl delete` command on the host:

    ```
    kubectl delete pod kata-nginx
    ```