**4**

**Understanding Container Engine for Kubernetes**

In today’s world, one of the qualities that sets apart successful businesses is agility, the ability to keep pace with constantly shifting trends and changing customer expectations. It’s not enough for decision makers in the company to adapt; the technology supporting the company must be able to keep up as well. This is one of the areas where cloud native development shines. Cloud native is an approach or methodology that takes advantage of the scalability, resilience, and efficiency of cloud infrastructure. It enables businesses to quickly iterate without sacrificing the quality of their service to gain an advantage over their competition.

This chapter contextualizes the importance of cloud native development and container orchestration using Kubernetes and then dives into how to operate your own Kubernetes cluster using Oracle Container Engine for Kubernetes (OKE). By the end, you should have the tools you need to create clusters and nodes, deploy containers, and securely expose those containers to your users.

**Monoliths and Microservices**

Classic software development relies on monoliths, an approach in which applications consist of a single code base, including the business logic, data, and resources needed to run the application, and are deployed to static physical infrastructure. Monoliths contain a large amount of code, and they commonly include complex dependencies and require interactions between different components. Updating one part of the application means that operators need to build, test, and deploy the application as a whole instead of having to do so for only the specific part being iterated. Similar challenges apply to scaling: Even if only one aspect of the monolith is responsible for throttling performance and needs to be scaled, there is no capability for it to be independently scaled. The entire monolith needs to be deployed to larger or more performant hardware.

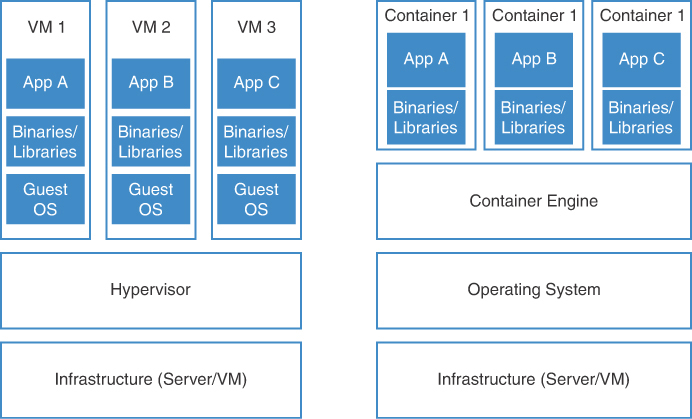
In contrast to the inflexibility of software monoliths, cloud native development relies on microservices, a software architecture approach in which applications are built as sets of small, independent services, each responsible for a specific business function and working in concert to deliver a unified experience. By decoupling monoliths into microservices, applications can be iterated faster. Each service can be developed, tested, and deployed independently. You can make changes to a single service without affecting the rest of the application. This also makes it easier to update and maintain the application. If you see a bottleneck in one aspect of architecture, that microservice can take advantage of the elasticity of the cloud to be independently scaled without the need to scale the application as a whole. Microservices are also designed to be flexible and resilient because they can be deployed on different servers and can continue to function even if one of the services fails. This makes them well suited for distributed systems and cloud environments. All of these qualities enable microservices to address the fundamental requirement of IT infrastructure that supports business agility, making them the de facto standard for cloud native software.

However, microservices can be more complex to design and manage than monolithic applications because they require more effort to coordinate the communication among the different services. They also require a more robust infrastructure to support the deployment and management of the individual services.

**Containers**

After choosing to adopt a microservice approach, you might ask, “How do I package my software into microservices?” Separating the software from the host can be accomplished in multiple ways. One option is to use virtual machines. A virtual machine (VM) allows a computer to run multiple operating systems simultaneously by creating one or more virtual environments on physical hardware, each of which behaves like a separate physical device.

Another option is to use containers, which enables you to package an application, along with its dependencies, runtime, and libraries, into a single unit that can be easily moved and deployed on any system that supports the container technology. A container appears to be a standalone system, with its own root file system. Containers are a combination of namespaces, control groups, and supporting OS features. [Figure 4-1](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04.xhtml#ch04fig01) shows a visual of the differences between containers and virtual machines (VMs).



**Figure 4-1** Architectural Difference Between Virtual Machines and Containers

Software containers are used for microservices instead of VMs for a number of reasons. Containers are much more lightweight than VMs because they do not require a separate operating system to be installed. This means that they can be started and stopped more quickly, and they take up less space on the host machine. Containers are more efficient than VMs because they share the host machine’s operating system kernel and use resource isolation features to ensure that each container has access to the resources it needs to run. This means that they can run more applications on the same hardware, which can lead to cost savings. Containers enable you to define the exact dependencies and configuration required for an application to run, which means that you can deploy the application consistently across different environments, often referred to as portability.

Microservices can make it easier to develop and maintain an application because each service can be developed, tested, and deployed independently of the others. Containers are easier to scale than VMs and can be deployed to multiple servers or cloud environments without the need to reconfigure the underlying infrastructure. Additionally, you can quickly and easily adjust the number of instances of a particular microservice in response to changes in demand. Overall, the use of containers in microservice architectures helps to increase the agility, scalability, and reliability of the overall system.

However, these benefits do not come free of challenges. This leads to questions such as the following:

How do we scale to keep pace with dynamic workload demands?

When I grow from tens or hundreds of containers to thousands of containers, how can I ensure that all of my containers are healthy and running?

**Container Orchestration and Kubernetes**

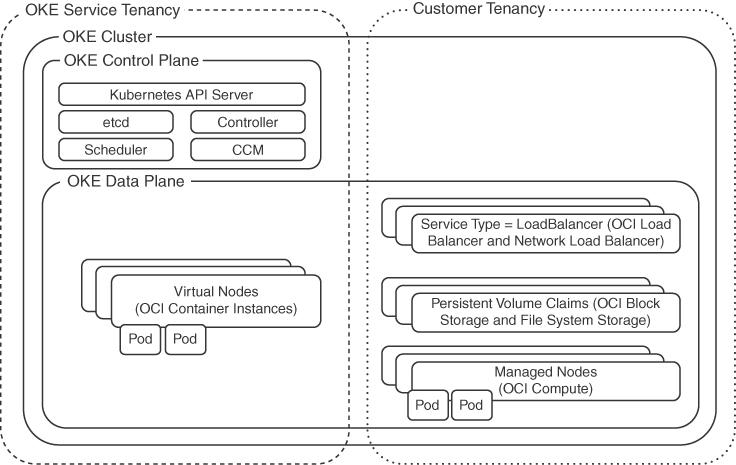
Kubernetes is the answer to many of these questions. At a high level, Kubernetes is an open-source infrastructure abstraction used to orchestrate the deployment, scaling, and management of containerized applications. Kubernetes leverages automation and declarative configuration. It provides a consistent and easy-to-use interface for managing containerized applications, regardless of the underlying infrastructure. Kubernetes enables you to easily scale your application up or down by adding or removing containers as needed. This makes it easier to handle changes in demand or load. Kubernetes can automatically detect when a container or node has failed, and it can automatically restart the container or replace the node to keep the application running. With regard to portability, as with the containers it orchestrates, Kubernetes can run on multiple cloud platforms and also on-premises, making it easy to move applications between environments. Overall, Kubernetes makes it easier to deploy and manage containerized applications in a cloud native environment, which can help organizations increase the agility, scalability, and reliability of their applications. More tangibly, Kubernetes is cluster software with one or more control plane nodes controlling a group of worker nodes. The scheduler deploys work in the form of pods, a unit of one or more containers, to the worker nodes using various patterns.

**Oracle Container Engine for Kubernetes**

Every new software tool comes with a learning curve, and Kubernetes is no exception. Kubernetes comes with a new set of concepts to learn and a large number of control plane and node components that must be deployed and operated, including these:

* **Control plane components:**
  + kube-apiserver
  + etcd
  + kube-scheduler
  + kube-controller-manager
  + cloud-controller-manager
* **Node components:**
  + kubelet
  + kube-proxy
  + The container runtime

[Figure 4-2](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04.xhtml#ch04fig02) illustrates these components.



**Figure 4-2** OKE Architecture, Including Components of a Kubernetes Cluster That Are Managed by the OKE Service and Managed by Customers

In some cases, learning about how to do this, let alone actually implementing it, can be overwhelming and prevents businesses from taking the leap to modernize their IT infrastructure. In other cases, it’s simply too complex, costly, and time-consuming for businesses to build and maintain an environment like this. Managed Kubernetes offerings provide users with a simple way to automatically deploy, scale, and manage Kubernetes. Offloading responsibility to a provider enables developers to create a cluster and deploy containers quickly.

Oracle Container Engine for Kubernetes (OKE) combines production-grade container orchestration of open Kubernetes with the control, security, IAM, and high predictable performance of Oracle Cloud Infrastructure. OKE fully manages the cluster control plane for you and makes it easy to create and manage other components of your cluster, including worker nodes, load balancer services, and more. You can get started quickly by using a Quick Create cluster, which streamlines cluster creation by providing an opinionated provisioning experience and automatically creates networking resources on your behalf. Alternatively, you can create a cluster using existing network resources and with a large number of configuration options using the Custom Create workflow.

**OCI-Managed Components and Customer-Managed Components**

A Kubernetes cluster consists of a group of nodes, physical or virtual machines, that run cluster software. These nodes fall into two categories: a control plane and a data plane.

**Control Plane**

The Kubernetes control plane is a set of several components that work together to manage the overall state of a Kubernetes cluster. The cluster control plane monitors and records the state of the worker nodes in the cluster’s data plane and distributes requested operations to them. It runs on multiple control plane nodes configured for high availability in the OKE service tenancy. When using OKE, the cluster control plane is fully managed by Oracle.

The cluster control plane consists of these components:

* **kube-apiserver** serves as the front end for the Kubernetes control plane by exposing the Kubernetes API. This includes supporting direct API calls and requests from the Kubernetes command-line tool, kubectl.
* **etcd** is a distributed key-value store used to contain the configuration data for the Kubernetes cluster. It provides a reliable and highly available storage solution for the control plane.
* **kube-scheduler** watches for newly created pods and assigns pods to nodes based on resource availability and other constraints.
* **kube-controller-manager** is a collection of controller processes that manage different Kubernetes components.
* **cloud-controller-manager** connects your cluster to your cloud provider’s API and runs a collection of controller processes that are specific to your cloud provider. For example, in the case of Oracle Container Engine for Kubernetes, the oci-cloud-controller-manager uses the nodeController to update and terminate worker nodes and the serviceController to create load balancers when Kubernetes services of type: LoadBalancer are created. The oci-cloud-controller-manager also implements a container-storage-interface, a FlexVolume driver, and a FlexVolume provisioner to manage additional OCI resources.

**Data Plane**

The Kubernetes data plane consists of one or more worker nodes, where you run the containerized applications deployed to your cluster. Each node is managed by the control plane and contains the node components necessary to run pods, including the following:

* **kubelet** is the agent that communicates with the cluster control plane to verify that containers are running.
* **kube-proxy** is a networking proxy used to maintain networking rules. These rules enable network communication to your pods from within and outside the cluster.
* The **container runtime** is the software responsible for running containers. In the case of OKE, this runtime is CRI-O.
* A **node pool** is a set of worker nodes within a cluster that all possess the same properties. Node pools enable you to create groups of nodes within your cluster to accommodate workloads with different requirements. For example, within the same cluster, you might create one pool of virtual machine nodes and another pool of GPU nodes for your HPC workloads. A cluster must have a minimum of one node pool, but a node pool does not need to contain any worker nodes.

When creating a node pool in an OKE cluster, you must specify the type of worker node to create:

* **Virtual nodes:** Virtual node resources are provisioned dynamically, as needed, and exist in the OKE service tenancy. Virtual nodes remove the operational overhead of upgrading your data plane infrastructure and managing the capacity of clusters, providing a “serverless” Kubernetes experience.
  + **Managed nodes:** Managed nodes are compute instances in your tenancy that are managed by a combination of you and the OKE service. Managed nodes come with the flexibility to configure them to meet your specific requirements, but you are responsible for upgrading Kubernetes and host OS versions and ensuring that capacity is properly scaled.

**Billable Components**

Those same categories of the control plane and data plane can be useful to understand the billable components of your cluster. Simply put, your control plane node usage is not metered, but your data plane nodes are metered and billed for their usage.

At the cluster level, you have a choice between enhanced and basic clusters. Enhanced clusters support all available OKE features, including features that are not supported by basic clusters. These features include but are not limited to virtual nodes, add-on lifecycle management, workload access to OCI resources, on-demand node cycling, and additional worker nodes per cluster. Enhanced clusters also come with a financially backed service-level agreement (SLA) tied to the availability of the Kubernetes API server. Basic clusters support all the core functionality provided by Kubernetes and Container Engine for Kubernetes but support none of the enhanced features that Container Engine for Kubernetes provides. Basic clusters come with a service-level objective (SLO) but not a financially backed SLA. A management fee is associated with using enhanced clusters, whereas basic clusters do not have a fee.

Creating a new cluster of the enhanced type enables you to use enhanced features immediately or at any point in the future. If you choose to create a new cluster of the basic type, you can still upgrade the cluster from a basic to an enhanced type at any point in time. Keep in mind that you cannot downgrade from an enhanced type cluster to a basic type cluster.

The data plane consists of worker nodes, which are charged based on the compute shape chosen for the node. In the case of virtual nodes, worker nodes are charged based on the container instance shape chosen for the node and are assessed an additional management fee per virtual node. The data plane is the primary source of costs for running an OKE cluster.

Additionally, there are charges for cluster resources you provision that are backed by OCI resources. For example, creating a Kubernetes service of type LoadBalancer results in the creation of an OCI Load Balancer resource, for which you will be charged. The same is true for other Kubernetes resources, such as persistent volumes and OCI block storage or file system storage.

Those same categories of the control plane and data plane can be useful in understanding the billable components of your cluster. Simply put, you are not charged for your control plane nodes, but you are charged for your data plane nodes.

**Kubernetes Concepts**

The more you know about the intricacies of the Kubernetes API, the more powerful you will become as a cluster operator. Even if ultimate knowledge of the Kubernetes API is not your goal, certain key concepts are worth knowing. The comprehensive reference for Kubernetes concepts is the excellent Kubernetes documentation itself. This book does not attempt to re-create the Kubernetes documentation. Instead, this section serves as a quick refresher for some of the most commonly used Kubernetes resources and concepts:

* **Pods:** Pods are one or more containers with shared resources, including CPU, memory, storage, and networking. The contents of a pod are always scheduled together. Pods are often created and destroyed as a result of being rescheduled onto new nodes or when new versions of an application are rolled out. Each pod has an ephemeral IP address that is assigned when the pod is first created and is released when the pod is terminated. Each pod has a universally unique identifier (UUID).
* **Deployments:** Deployments are used to specify the desired state of an application, including the number of replicas of a pod or pods that should be running at a given time. Deployments can be used to upgrade your application by rolling out new pod versions with zero downtime and to roll back to a previous state.
* **Namespaces:** A Kubernetes cluster can be organized into namespaces, to divide the cluster’s resources among multiple users. Initially, a cluster has the following namespaces: default, for resources with no other namespace, and kube-system, for resources created by the Kubernetes system.
* **Services:** Services are an abstraction that defines a method for accessing a pod or pods. This is the Kubernetes way of decoupling the discovery of an application from the application instances. This allows one application to have a reliable address, regardless of whether the pods that make up the application change. Some parts of an application, such as the front ends, are typically exposed through the ingress controller or directly with an external IP address accessible from outside a cluster. Other applications might choose to use the service abstraction for service discovery because it decouples the consumer of an application service from the pods that make up the application service. Kubernetes ServiceTypes enable you to specify the way you want your pods exposed:
  + A **LoadBalancer** ServiceType creates an Oracle Cloud Infrastructure load balancer on load balancer subnets in your VCN. These load balancers are automatically configured to route to the pods. The load balancer configuration is updated when the pod configuration changes, such as when new pods are added and existing pods are deleted. Kubernetes clusters created by OKE also include capabilities that can automatically update OCI resources such as security lists when load balancers are created and applications are exposed using them.
  + A **NodePort** service is a type of service that exposes a specific port on each node in the cluster. This allows external traffic to access the service by sending requests to a node’s IP address and the node port specified for the service. Node port services do not provide the same level of traffic management and load balancing as other service types, such as LoadBalancer and ClusterIP.
  + A **ClusterIP** service is the default service type that is used when no other service type is specified. It assigns an IP address from an IP pool that the cluster’s networking plug-in (CNI) manages. On Kubernetes clusters created by OKE, you have the choice of two CNIs: Flannel or the OCI Native CNI. This topic is covered in greater depth in the “[Kubernetes Networking](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04.xhtml#ch04lev1sec10)” section.
* **Labels and selectors:** Labels and selectors are key/value pairs attached to objects in a Kubernetes cluster. Labels are used to identify and organize Kubernetes objects, such as to explain which application a pod is associated with.

**Cloud Controller Manager**

The Kubernetes Cloud Controller Manager (CCM) is a controller used to implement cloud provider–specific control loops required for Kubernetes to function. For example, CCM can implement a node controller that is responsible for updating Kubernetes nodes using a cloud provider’s API and deleting Kubernetes nodes that were deleted on your cloud. Kubernetes introduced the CCM project to decouple the development of cloud features from the core Kubernetes project. Early in the existence of Kubernetes, cloud providers were added in-tree in the kube-controller-manger binary. To increase extensibility and remove the need to directly interact with the Kubernetes code base, external cloud providers were introduced. External cloud providers are Kubernetes controllers that implement cloud provider–specific control loops required for Kubernetes to function, but for out-of-tree providers.

The OCI Cloud Controller Manager, which implements OCI-specific control loops, is an example of an external cloud provider. For example, the oci-cloud-controller-manager implements a NodeController, which is used to update OCI Compute nodes with cloud provider–specific labels and addresses; it also deletes Kubernetes nodes when they are deleted from OCI by scaling down a node pool or using the delete node API. It also implements a ServiceController, which is responsible for creating OCI load balancers when a service of type: LoadBalancer is created in Kubernetes. Another key aspect of this project is related to storage: The OCI CCM implements a Container Storage Interface, a volume provisioner, and a FlexVolume driver for Kubernetes clusters running on OCI.

The CSI plug-in for OCI enables provisioning, attaching, detaching, mounting, and unmounting of OCI block storage volumes to Kubernetes pods via the Container Storage Interface (CSI) plug-in interface. The volume provisioner enables the dynamic provisioning of OCI storage resources, such as block volumes, while the FlexVolume driver enables the mounting of OCI block storage volumes to Kubernetes pods using the FlexVolume plug-in interface.

Similar to the manner in which CCM was introduced to enable extensibility for cloud providers, CSI was developed as a standard for exposing arbitrary block and file storage systems to containerized workloads running on Kubernetes. Using CSI made it simpler for third-party storage providers, such as OCI, to write and deploy plug-ins exposing new storage systems in Kubernetes without ever having to touch the core Kubernetes code. Most OKE users have moved from using FlexVolume to using CSI.

**Nodes and Node Pools**

Worker nodes constitute the cluster data plane. Worker nodes are where you run the applications that you deploy in a cluster. These are compute instances with additional software that communicates with the Kubernetes control plane to make it a worker node that is known to the cluster control plane. Worker node runs a number of processes, including these:

* kubelet to communicate with the cluster control plane
* kube-proxy to maintain networking rules

The cluster control plane processes monitor and record the state of the worker nodes and distribute requested operations among them.

A node pool is a subset of worker nodes within a cluster that all have the same configuration. Node pools enable you to create pools of machines within a cluster that have different configurations. For example, you might create one pool of nodes in a cluster as virtual machines and another pool of nodes as bare metal machines. A cluster must have a minimum of one node pool, but a node pool need not contain any worker nodes. Worker nodes in a node pool are connected to a worker node subnet in your VCN.

When creating a node pool with OKE, you specify that the worker nodes in the node pool are to be created as one of the following:

* **Virtual nodes**, fully managed by Oracle. Virtual nodes provide a “serverless” Kubernetes experience, enabling you to run containerized applications at scale without the operational overhead of upgrading the data plane infrastructure and managing the capacity of clusters. You can create virtual nodes only in enhanced clusters.
* **Managed nodes**, running on compute instances (either bare metal or virtual machine) in your tenancy and at least partly managed by you. Because you are responsible for managing managed nodes, you have the flexibility to configure them to meet your specific requirements. You are responsible for upgrading Kubernetes on managed nodes and for managing cluster capacity. You can create managed nodes in both basic clusters and enhanced clusters.

**Node Pool Properties**

Node pools possess a set of standard properties that are inherited by worker nodes running in the pool. These properties include but are not limited to the following:

* The name of a node pool
* The version of Kubernetes to run on new worker nodes
* The number of worker nodes in a node pool and the availability domains, fault domains, and subnets in which to place them
* The image to use for new worker nodes
* The shape to use for new worker nodes
* The boot volume size and encryption settings to use for new worker nodes
* The cordon and drain options to use when terminating worker nodes
* The cloud-init script to use for instances hosting worker nodes
* The public SSH key to use to access new worker nodes

**Worker Node Images and Shapes**

You can customize the worker nodes in your OKE cluster by specifying the following:

* **The operating system image:** The host image includes the operating system and other software required for the instance to act as a Kubernetes worker node. Note that this option is available exclusively to managed nodes because infrastructure management is abstracted away by the service when using virtual nodes.
* **The shape:** The shape is the number of OCPUs and the amount of memory to allocate to each newly created instance to be used as a worker node. The choice of shape can also dictate other infrastructure properties, such as available network bandwidth and specialized hardware such as GPUs.

**Images**

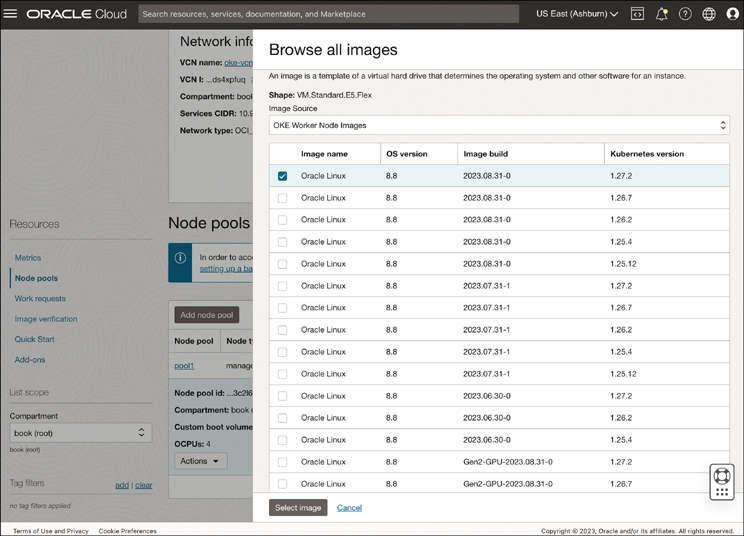
Three types of images are available for use as worker node images, built for all shape architectures supported by OKE:

* **OKE images** are built on top of Oracle Linux platform images and include all the necessary configurations and required software for use as managed nodes. They are optimized to minimize the time it takes to provision managed nodes at runtime when compared to platform images and custom images. The use of OKE images reduces managed node provisioning time by more than half when compared to platform images.

Because the Kubernetes software is prebaked onto the host image, OKE images bundle together the host OS and Kubernetes version. The Kubernetes version that you specify when creating and updating node pools must match the Kubernetes version of your chosen OKE image. In the console, this is automatically done for you.

* **Platform images** also contain the Oracle Linux operating system, but unlike with OKE images, additional Kubernetes software is not prebaked into the image. When the managed node boots up for the first time, OKE downloads, installs, and configures the required software based on the Kubernetes version you select. This is the legacy option available for worker node images.
* **Custom images** are built from OKE or Platform images but are provided by you. They give you the option to bring your own host image configurations. Unlike OKE and Platform images, custom images are not explicitly supported by OKE.

You can find the latest available images in the console in a few ways: When creating a cluster in the Custom Create workflow or when creating or editing a node pool, you can view the list of supported platform images and OKE images in the Browse All Images window (see [Figure 4-3](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04.xhtml#ch04fig03)).



**Figure 4-3** Selection of a Worker Node Host Image—Here, Oracle Linux 8 Running Kubernetes 1.27.2

**Shapes**

OKE supports a growing variety of shapes for use as worker nodes. These include most but not all of the shapes available through Oracle Cloud Infrastructure. The choice of shape can be important when it comes to supporting specific workloads. For example, high-performance computing or machine learning use cases can benefit from the use of GPU shapes. These shapes are divided into two categories: those supported for managed nodes and those supported for virtual nodes.

**Managed Nodes**

The following shapes are supported for use as managed nodes:

* Flexible shapes
* Bare metal shapes (including standard shapes and GPU shapes)
* HPC shapes (except in RDMA networks)
* VM shapes (including standard shapes and GPU shapes)
* Dense I/O shapes

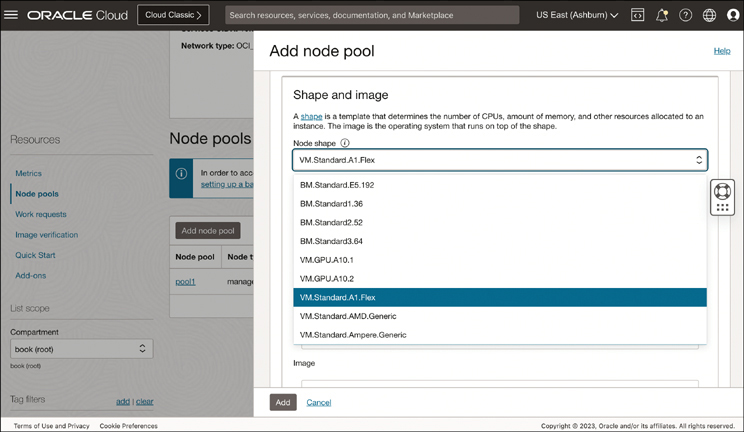
The following shapes are not supported for use as managed nodes:

* Dedicated VM host shapes
* Micro VM shapes
* HPC shapes on bare metal instances in RDMA networks
* Burstable capacity for flexible shapes

**Virtual Nodes**

At the time of writing, the Standard.E3.Flex and Standard.E4.Flex shapes are supported for use as virtual nodes. All other shapes are not supported for use as virtual nodes.

The available shapes increase regularly. The best way to stay on top of shape availability is directly through the console or CLI. You find the latest available shapes in the console in a few ways: When creating a cluster in the Custom Create workflow or when creating or editing a node pool, you can view the list of supported shapes in the Browse All Shapes window (see [Figure 4-4](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04.xhtml#ch04fig04)).



**Figure 4-4** Selection of a Worker Node Shape—Here, the VM.Standard.A1.Flex Shape

When using the CLI, you can view the supported OKE, platform, and custom images in the data.shapes: section of the response of the following command:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04_images.xhtml#f105-01a)

oci ce node-pool-options get --node-pool-option-id all

**Note**

Even if a shape is supported by OKE, you might not be able to select it in your tenancy or in a particular region because of service limits, compartment quotas, or available capacity.

**Custom cloud-init**

To customize your managed worker nodes, OKE gives you the option to add your own logic at node startup time. OKE uses cloud-init, the industry-standard method for cloud instance initialization, to set up compute instances as managed nodes. The first time the instance boots up, cloud-init runs the default startup script:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04_images.xhtml#f105-02a)

#!/bin/bash

curl --fail -H “Authorization: Bearer Oracle” -L0 http://169.254.169.254/opc/v2/

instance/metadata/oke\_init\_script | base64 --decode >/var/run/oke-init.sh

bash /var/run/oke-init.sh

You can customize the default startup script by adding your own logic either before or after the default logic. A custom cloud-init script can be used to do the following:

* Configure the kubelet running on the worker node
* Expand the boot volume with growfs
* Configure a corporate proxy or custom YUM proxies
* Install mandated security tools

The customized startup script runs when a worker node boots up for the first time. If you choose to add your own custom logic, it can be useful to test whether that logic negatively impacts the host’s use as a worker node. To do so, you can run the Node Doctor script to confirm that worker nodes on newly started instances are working as expected.

**Note**

If you customize the default startup script, do not modify the logic provided by OKE. It can be easy to accidentally overwrite the OKE startup script, so double-check that the script is still present before you pass in your changes.

**Using Custom cloud-init Script to Set kubelet-extra-args on Managed Nodes**

A straightforward use of a custom cloud-init script is to use it to configure extra options on the kubelet (the primary node agent) on managed nodes. These extra options, sometimes referred to as kubelet-extra-args, include the option to configure debug log verbosity. [Figure 4-5](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04.xhtml#ch04fig05) shows how to configure a custom cloud-init script in the OCI Console.

1. In the console, navigate to create a new node pool or edit an existing node pool.
2. In the **Show Advanced Options** section, navigate to **Initialization Script**, as illustrated in [Figure 4-5](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04.xhtml#ch04fig05).
3. Begin by clicking **Download Custom Cloud-Init Script Template** to download a boilerplate script. The file contains the default logic required by OKE. You can add your own custom logic either before or after the default logic. Remember not to modify the default logic.
4. You can choose to upload this cloud-init script or paste your script directly into the console.
5. Use the following cloud-init script to configure the debug level log verbosity:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04_images.xhtml#f107-01a)

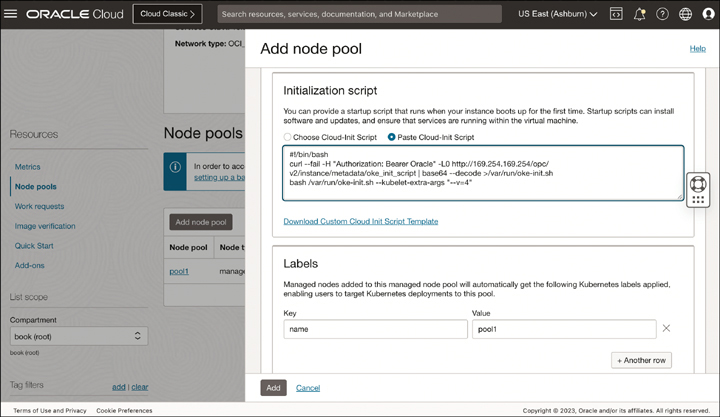
#!/bin/bash

curl --fail -H "Authorization: Bearer Oracle" -L0 http://169.254.169.254/opc/

v2/instance/metadata/oke\_init\_script | base64 --decode >/var/run/oke-init.sh

bash /var/run/oke-init.sh --kubelet-extra-args "--v=4"

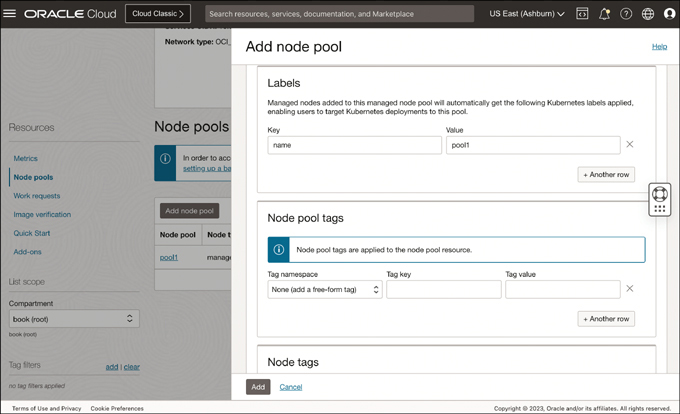
1. After clicking **Add** or **Save Changes**, depending on whether you created a new node pool or are editing an existing node pool, the cloud-init logic will be passed to a newly created node.
2. To confirm the setting of debug level log verbosity, connect to a worker node and use the sudo systemctl status -l kubelet command. This command returns the verbosity level as 4 for all nodes on which the preceding cloud-init script was run.



**Figure 4-5** Adding a Custom cloud-init Script to a Node Pool to Modify the Verbosity of Logs Generated by the Kubelet Running on the Node

**Kubernetes Labels**

Kubernetes labels are key/value pairs used to specify identifying attributes of objects in a human-readable way. Labels can be used to associate objects with a particular application or a line of business to implement chargeback. Another great use for labels is to organize and select nodes when deploying applications. Each node comes with a set of default labels, including those related to the shape and architecture of the node. Labels enable you to target workloads at specific node pools. You also can use an optional node pool property to add more labels to worker nodes directly through the OKE API and using the OCI Console (see [Figure 4-6](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04.xhtml#ch04fig06)). As with other node pool properties, labels are attached nodes at creation time. You can subsequently add or modify labels using the Kubernetes API.



**Figure 4-6** How to Add Labels to Nodes in a Node Pool Using the OCI Console

To see the labels of a given node, run kubectl describe node [node name]. [Listing 4-1](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04.xhtml#list4_1) shows the default labels applied to a managed node created by OKE.

**Listing 4-1** Labels Automatically Added to a Managed Node Created by OKE—These Labels Will Change, Depending on the Shape Chosen for the Node Pool

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04_images.xhtml#f108-01a)

$ kubectl describe node 10.0.10.151

Name: 10.0.10.151

Roles: node

Labels: beta.kubernetes.io/arch=amd64

beta.kubernetes.io/instance-type=VM.Standard.E3.Flex

beta.kubernetes.io/os=linux

displayName=oke-c2usfphkqza-nctxoizruoq-seoda7iqkwa-3

failure-domain.beta.kubernetes.io/region=uk-london-1

failure-domain.beta.kubernetes.io/zone=UK-LONDON-1-AD-2

hostname=oke-c2usfphkqza-nctxoizruoq-seoda7iqkwa-3

internal\_addr=10.0.10.151

kubernetes.io/arch=amd64

kubernetes.io/hostname=10.0.10.151

kubernetes.io/os=linux

last-migration-failure=get\_kubesvc\_failure

name=NC

node-role.kubernetes.io/node=

node.info.ds\_proxymux\_client=true

node.info/compartment.name=oracle-cloudnative

node.info/kubeletVersion=v1.25

node.kubernetes.io/instance-type=VM.Standard.E3.Flex

oci.oraclecloud.com/fault-domain=FAULT-DOMAIN-2

oke.oraclecloud.com/node.info.private\_subnet=false

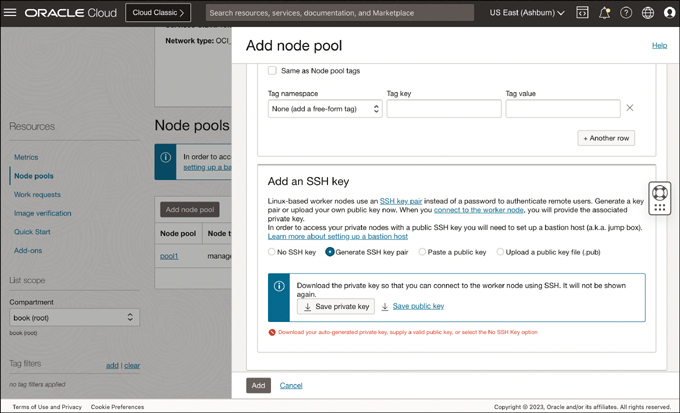
oke.oraclecloud.com/node.info.private\_worker=true

topology.kubernetes.io/region=uk-london-1

topology.kubernetes.io/zone=UK-LONDON-1-AD-2

**SSH Keys**

SSH keys are another optional node pool property. Adding your public portion of an SSH key pair to the node pool enables you to access the nodes directly through SSH. The public key is added to all worker nodes in the cluster. If you don’t specify a public SSH key, you will not have SSH access to the worker nodes. [Figure 4-7](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04.xhtml#ch04fig07) shows a user adding an SSH key to the node pool using the OCI Console. Note that you cannot use SSH to directly access worker nodes in private subnets because they have private IP addresses only; they are accessible only by other resources inside the VCN. You can use the Oracle Cloud Infrastructure Bastion service to enable external SSH access to worker nodes in private subnets.



**Figure 4-7** Adding an SSH Key to a Node Pool Using the OCI Console

**Tagging Your Resources**

Oracle Cloud Infrastructure Tagging allows you to add metadata to resources, which enables you to define keys and values and then associate them with resources. At their most basic, tags can be used to organize resources based on your business needs; however, tags opens up a lot of possibilities, from cost tracking to access control.

OCI offers tagging in two flavors: *free-form tags* and *defined tags*. Most of the advanced capabilities of tagging are applicable to defined tags, in which you create a tag namespace and then create a series of well-defined tag keys for which you can use a multitude of tag values. Although all defined tags can be used for cost analysis and usage reporting, defined tags that are designated as cost-tracking tags allow you to use them in OCI budgets. Budgets can track and forecast the cost for resources and alert you proactively when the forecast or actual consumption crosses a threshold of the budget you have set.

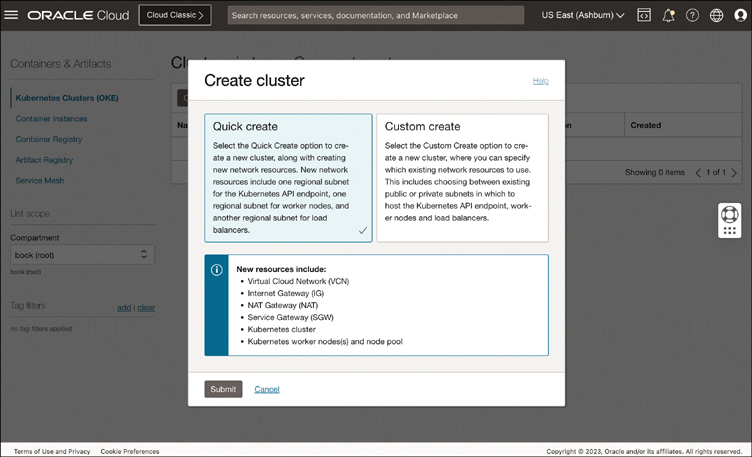
Consider a scenario in which you have a single cluster used by several applications or teams. You might want to implement both cost tracking and access control on a per-application (or per-team) basis on this shared infrastructure. Assume that each application is deployed on its own dedicated node pool. The applications can also create and use other resources, such as load balancers and storage, dynamically. To get an accurate estimate of the cost for each application, you can use tagging to tag the resources this application uses. Similarly, you can write access policies that restrict access for each team to only the resources that are used by their application, using tags.

Tags can be set on clusters, node pools, load balancers, and storage attachments. When creating a cluster, you can set tag defaults for the various types of resources, which you can also override with resource annotations when needed. With the resources tagged, you can implement features such as cost tracking, setting access controls, or setting budgets for the various applications. Similarly, you can use tags to keep track of the resources used by the dev, test, and prod environments for an individual application.

Tags present a flexible way of attaching additional metadata to resources: How you use these tags is up to you. This affords a tremendous amount of flexibility in the types of tags you can create and how you can leverage them.

**Creating a Cluster**

You can use OKE to create new Kubernetes clusters in many ways, including using the console (web UI), the CLI, automation tools such as Terraform, or the APIs directly. The console, in particular, offers two workflows to get you started: the Quick Create and Custom Create workflows (see [Figure 4-8](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04.xhtml#ch04fig08)). The Quick Create workflow is the fastest way to create a new cluster. This approach automatically creates new network resources, including regional subnets for the Kubernetes API endpoint, for worker nodes, and for load balancers. This workflow is ideally suited if you are new to Kubernetes and want to get started quickly.



**Figure 4-8** Choosing Between the Quick Create and Customer Create Workflows

**Note**

To create a cluster, you must belong to either the tenancy’s Administrators group or a group to which a policy grants the CLUSTER\_MANAGE permission.

**Quick Create Cluster Workflow**

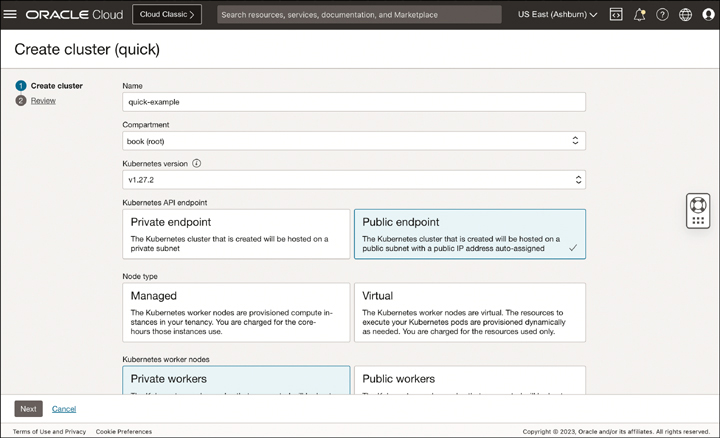
In the Console, open the navigation menu and click Developer Services. Under Containers & Artifacts, click Kubernetes Clusters (OKE).

Step 1.Choose a compartment, and click **Create Cluster**.

Step 2.Select **Quick Create** and then click **Submit**.

Step 3.Either accept the default configurations (see [Figure 4-9](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04.xhtml#ch04fig09)) or choose alternatives:

1. Give a name to your cluster.
2. Choose the compartment where you want your cluster control plane and related networking resources created.
3. Choose a Kubernetes version for your cluster control plane.
4. Specify whether you want a private or public Kubernetes API endpoint. In the case of a private subnet, the Kubernetes API endpoint will be hosted on a private subnet and assigned a private IP address. In the case of a public subnet, the Kubernetes API endpoint will be hosted on a public subnet with a public IP address automatically assigned.
5. Choose between managed and virtual nodes. Managed Kubernetes worker nodes are compute instances in your tenancy. Managed nodes come with the flexibility to configure them to meet your specific requirements, but you are responsible for upgrading Kubernetes and host OS versions and for ensuring that capacity is properly scaled. In the case of virtual nodes, the resources to execute your Kubernetes pods are provisioned dynamically, as needed, and exist in the OKE service tenancy. Virtual nodes remove the operational overhead of upgrading your data plane infrastructure and managing the capacity of clusters.



**Figure 4-9** The First Step in the Quick Cluster Creation Workflow

Step 4.Depending on your chosen node type, the following steps will differ:

1. Choosing managed nodes gives you a choice between creating a private subnet or public subnet to host your Kubernetes worker nodes. It also give you a choice of image to use for your worker node hosts. These images determine the operating system and other software used for managed nodes. Selecting managed nodes also gives you expanded options for the shape of your nodes, compared to virtual nodes. Additionally, the choice of managed nodes enables you to customize the size and encryption options for the boot volumes of nodes in the node pool. Select the **Specify a Custom Boot Volume Size** check box, and enter a custom size from 50GB to 32TB to specify a custom size for the boot volume. The specified size must be larger than the default boot volume size for the selected image. If you increase the boot volume size, you must also extend the partition for the boot volume, to take advantage of the larger size using the oci-growfs utility. Nodes with the VM instance chosen as the shape allow you to optionally select the **Use In-Transit encryption** check box. This is not configurable for bare metal instances. Bare metal instances that support in-transit encryption have it enabled by default. Boot volumes are encrypted by default, but you can optionally use your own Vault service encryption key to encrypt the data in this volume. To use the Vault service for your encryption needs, select the **Encrypt This Volume with a Key That You Manage** check box. Then select the Vault compartment and Vault that contain the master encryption key you want to use. Also select the master encryption key compartment and master encryption key. If you enable this option, this key is used for both data-at-rest encryption and in-transit encryption.
2. Choosing virtual nodes gives you a choice of pod shape, which determines the processor type on which to run the pod. Note that you explicitly specify the CPU and memory resource requirements for virtual nodes in the pod spec. Choosing virtual nodes also gives you the option to apply taints to nodes in the virtual node pool. Taints allow virtual nodes to repel pods, thereby ensuring that pods do not run on virtual nodes in a particular virtual node pool.
3. Both options enable you to choose the number of nodes created in the default node pool. Both options also allow you to optionally specify Kubernetes labels. These labels are added to the set of default labels already on the node and are used to target workloads at specific node pools.

Click **Next** to review the details you entered for the new cluster. If you have not selected any features restricted to enhanced clusters, you can choose to create a basic cluster. To do so, check the **Create a Basic Cluster** check box on the Review page. Otherwise, leave the box unchecked to create an enhanced cluster. Click **Create Cluster** to create the new network resources and the new cluster. Click **Close** to return to the Console.

**Custom Create Cluster Workflow**

The Custom Create workflow gives you the most control over creating a new cluster. It allows you to explicitly define the new cluster’s properties and specify which existing network resources to use, including the existing public or private subnets in which to create the Kubernetes API endpoint, worker nodes, and load balancers. Because the Custom Create workflow opens up more features and configuration options, it is better suited for more advanced scenarios, such as when you want to bring your own networking resources or when you need to configure advanced capabilities.

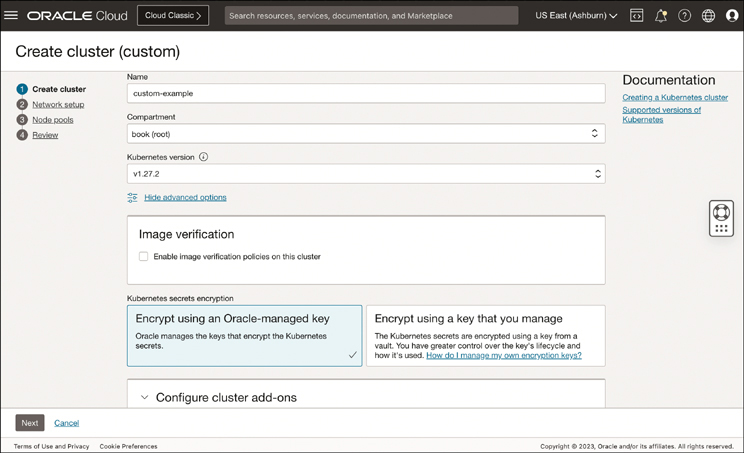
Step 1.In the Console, open the navigation menu and click **Developer Services**. Under Containers & Artifacts, click **Kubernetes Clusters (OKE)**.

Step 2.Choose a compartment and click **Create Cluster**.

Step 3.Select **Custom Create** and then click **Submit**.

Step 4.You can accept the default configurations or choose alternatives (see [Figure 4-10](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04.xhtml#ch04fig10)):

1. Give a name to your cluster.
2. Choose the compartment where you want your cluster control plane and related networking resources created.
3. Choose a Kubernetes version for your cluster control plane.



**Figure 4-10** The First Step in the Custom Cluster Creation Workflow

Step 5.Click **Show Advanced Options** to view other options available for cluster configuration.

1. Specify whether to allow the deployment of images from Oracle Cloud Infrastructure Registry only if they have been signed by particular master encryption keys. To enforce the use of signed images, select **Enable Image Verification Policies on This Cluster**, and then specify the encryption key and the vault that contains it.
2. **Encrypt using an Oracle-managed key:** Encrypt Kubernetes secrets in the etcd key-value store using a master encryption key that is managed by Oracle.
3. **Encrypt using a key that you manage:** Encrypt Kubernetes secrets in the etcd key-value store using a master encryption key (stored in the Vault service) that you manage.
4. Specify how to manage cluster add-ons. Select **Configure Cluster Add-ons** to enable or disable specific add-ons, select add-on versions, opt into and out of automatic updates by Oracle, and manage specific customizations. Select the appropriate cluster add-on and set options as appropriate. See “Configuring Cluster Add-ons.”
5. Specify whether to add cluster tags to the cluster, initial load balancer tags to load balancers created by Kubernetes services of type LoadBalancer, and initial block volume tags to block volumes created by Kubernetes persistent volume claims. Tagging enables you to group disparate resources across compartments and also allows you to annotate resources with your own metadata.

Step 6.After clicking **Next**, choose between VCN-native pod networking and Flannel overlay for your network type. VCN-native pod networking allows Kubernetes pods to connect directly to a VCN subnet and communicate natively through a VCN with other pods, other services, and the Internet. Flannel overlay configures an overlay network for pod communication. Note that if you choose the Flannel overlay option, you will not be able to create virtual nodes or specify a subnet for pod communication.

Step 7.Choose the networking setup for your cluster:

1. Select an existing VCN in which to provision your cluster.
2. Optionally, choose a Kubernetes load balancer service subnet to host load balancers. The load balancer subnet must be different from worker node subnets, can be public or private, and can be regional or AD specific.
3. Select a public or private Kubernetes API endpoint subnet to act as a regional subnet to host the cluster’s Kubernetes API endpoint. The Kubernetes API endpoint is assigned a private IP address. If you selected a public subnet for the Kubernetes API endpoint, you can also assign a public IP address to the API endpoint. Note that assigning a public IP address makes this cluster accessible from the Internet.
4. Optionally, you can use security rules defined for one or more network security groups (NSGs) to control access to the cluster’s Kubernetes API endpoint.

Step 8.After clicking **Next**, you can define the properties for node pools for your cluster.

1. Begin by choosing a name, compartment, and Kubernetes version for your node pool. By default, the version of Kubernetes specified for the control plane nodes is selected. The Kubernetes version on worker nodes must be either the same version as that on the control plane nodes or an earlier version that is still compatible.
2. Choose between managed and virtual nodes. Managed Kubernetes worker nodes are compute instances in your tenancy. Managed nodes come with the flexibility to configure them to meet your specific requirements, but you are responsible for upgrading Kubernetes and host OS versions and for ensuring that capacity is properly scaled. In the case of virtual nodes, the resources to execute your Kubernetes pods are provisioned dynamically, as needed, and exist in the OKE service tenancy. Virtual nodes remove the operational overhead of upgrading your data plane infrastructure and managing the capacity of clusters.

Step 9.Depending on your chosen node type, the following steps will differ:

1. Choosing managed nodes gives you a choice of creating a private or public subnet to host your Kubernetes worker nodes. It also give you a choice of image to use for your worker node hosts. These images determine the operating system and other software used for managed nodes. This also gives you expanded options for the shape of your nodes, compared to virtual nodes. Additionally, the choice of managed nodes enables you to customize the size and encryption options for the boot volumes of nodes in the node pool. Select the **Specify a Custom Boot Volume Size** check box, and enter a custom size from 50GB to 32TB to specify a custom size for the boot volume. The specified size must be larger than the default boot volume size for the selected image. If you increase the boot volume size, you must also extend the partition for the boot volume to take advantage of the larger size using the oci-growfs utility. Nodes with the VM instance chosen as the shape allow you to optionally select the **Use In-transit Encryption** check box. This is not configurable for bare metal instances; bare metal instances that support in-transit encryption have it enabled by default. Boot volumes are encrypted by default, but you can optionally use your own Vault service encryption key to encrypt the data in this volume. To use the Vault service for your encryption needs, select the **Encrypt This Volume with a Key That You Manage** check box. Then select the Vault compartment and Vault that contain the master encryption key you want to use. Also select the master encryption key compartment and master encryption key. If you enable this option, this key is used for both data at rest encryption and in-transit encryption.
2. Choosing virtual nodes give you a choice of pod shape, which determines the processor type on which to run the pod. Note that you explicitly specify the CPU and memory resource requirements for virtual nodes in the pod spec. Choosing virtual nodes also provides you with the option to apply taints to nodes in the virtual node pool. Taints enable virtual nodes to repel pods, thereby ensuring that pods do not run on virtual nodes in a particular virtual node pool. You must also choose the subnet that will host your virtual nodes.
3. Both options allow you to choose the number of nodes created in the default node pool. Optionally, you can specify Kubernetes labels. These labels are added to the set of default labels already on the node and are used to enable targeting workloads at specific node pools. Additionally, you must specify a placement configuration for your node pool. This configuration determines the subnets, availability domain, and (optionally) fault domain in which to place worker nodes. Finally, you must also select the subnet to be used for pod communication. Here, you can define the number of pods per node, as well as the security rules defined in the network security group (NSG) to control access to the pod subnet.

Step 10.Click **Next** to review the details you entered for the new cluster. If you have not selected any features restricted to enhanced clusters, you can choose to create a basic cluster. To do so, check the **Create a Basic Cluster** check box on the Review page. Otherwise, leave the box unchecked to create an enhanced cluster.

Step 11.Click **Create Cluster** to create the new network resources and the new cluster.

Step 12.Click **Close** to return to the console.

**Using the OCI Command-Line Interface**

You can also create clusters with ease using the OCI CLI. This approach enables you to create clusters using automation tools and shell scripts. It is an option for when Terraform might present a steep learning curve or when you simply want to automate the creation of these resources and you do not necessarily care about maintaining these resource configurations or tracking drift from its initial configuration.

Because the OCI CLI works on individual OCI resources, it should be noted that the Kubernetes cluster is a separate resource from the node pools that belong to the cluster. When creating clusters, you use both of these resources to create a functional Kubernetes cluster using OKE. Node pools always belong to a cluster, so you first create a cluster resource. This cluster resource represents the control plane elements that are managed by Oracle. When you have a cluster and a cluster OCID, you add a node pool to this cluster. This can be done with the cluster and node pool resources, as follows:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04_images.xhtml#f117-01a)

oci ce cluster create [OPTIONS] # use –- help for full list of options

oci ce node-pool create [OPTIONS]

The first command, cluster create, creates a cluster with the properties that are set by the options for this command. The second command, node-pool create, creates a node pool and includes a reference to the cluster OCID for the cluster created by the first command. This creates the node pool that belongs to that cluster. The CLI can also include options to manage control flow in scrips, such as to wait for an action to complete. A more complete example in [Listing 4-2](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04.xhtml#list4_2) showcases a variety of options that you can pass when you are creating a cluster.

**Listing 4-2** Options Available When Creating an OKE Cluster Using the OCI CLI

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04_images.xhtml#f118-01a)

oci ce cluster create \

--name demo-cluster \

--kubernetes-version v1.26.2 \

--compartment-id … \

--vcn-id … \

--type ENHANCED\_CLUSTER \

--endpoint-public-ip-enabled true \

--endpoint-subnet-id … \

--service-lb-subnet-ids '["…"]' \

--wait-for-state SUCCEEDED \

--wait-interval-seconds 10 \

--max-wait-seconds 600

The command in [Listing 4-2](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04.xhtml#list4_2) creates a cluster named demo-cluster in the specified compartment and attached to a specific VCN. The cluster type is specified as an ENHANCED\_CLUSTER, and the Kubernetes API endpoint is configured to have a public IP address. The command also includes options to block until the creation of that cluster has entered a state named SUCCEEDED. The command waits a maximum of 600 seconds for this to occur and checks the progress every 10 seconds.

Similarly, [Listing 4-3](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04.xhtml#list4_3) demonstrates a more complete example for creating a node pool.

**Listing 4-3** Options Available When Creating a Managed Node Pool for an OKE Cluster Using the OCI CLI

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04_images.xhtml#f118-02a)

oci ce node-pool create \

--cluster-id … \

--name my-node-pool \

--node-image-id … \

--compartment-id … \

--kubernetes-version v1.26.2 \

--node-shape VM.Standard.E4.Flex \

--node-shape-config "{\"memoryInGBs\": 8, \"ocpus\": 1}" \

--pod-subnet-ids "[\"…\"]" \

--placement-configs "[{\"availability-domain\":\"xxxx:US-ASHBURN-AD-2\",

\"subnet-id\":\"…\", \"faultDomains\":[\"FAULT-DOMAIN-3\", \"FAULT-DOMAIN-1\"]},

{\"availability-domain\":\"xxxx:US-ASHBURN-AD-1\", \"subnet-id\":\"…\",

\"faultDomains\": [\"FAULT-DOMAIN-1\", \"FAULT-DOMAIN-2\"]}]" \

--size 1 \

--region=us-ashburn-1

The command creates a new node pool for the cluster that is identified with the provided cluster-id. The new node pool is named my-node-pool and uses Kubernetes version 1.26.2. The shape of the nodes within the node pool is set to E4 flex; because this is a flexible shape, the shape configuration option specifies the number of CPUs and the amount of memory each of the nodes should have. The placement configuration is another JSON-formatted attribute that describes how the nodes within this node pool are to be placed across availability domains and fault domains.

When using the CLI, you might want to know the supported values for the various configuration options. You can view the supported values using the following command:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04_images.xhtml#f119-01a)

oci ce node-pool-options get --node-pool-option-id all

The command lists the various supported values for every option you can set for a node pool. The data.sources element in the response from the command describes the options and their possible values. These values can often change as well. For instance, the OS images are published every month, and getting the latest image is recommended when you create new clusters. Similarly, permissible values for the other options can change, as with the list of supported shapes, which is constantly being expanded to include new shapes that OCI rolls out for general use. Similarly, the supported Kubernetes versions change periodically as new versions of Kubernetes are released.

**JSON-Formatted Configuration**

As you can see from the previous examples, OKE exposes several configuration options to control various aspects of your cluster and its node pools. This can make CLI commands lengthy and verbose. It is often desirable to encapsulate these verbose configuration options into a single document, which can potentially be source controlled for change tracking.

To make this possible, the OCI CLI includes an option to provide the entire cluster configuration as a JSON document. This includes all the parameters that are configurable and presented to the CLI in a predefined JSON format that the CLI then parses before creating or updating the resource. To understand the expected structure of this JSON document, you can use the following commands:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04_images.xhtml#f119-02a)

# For the cluster resource

oci ce cluster create --generate-full-command-json-input

# For the node-pool resource

oci ce node-pool create --generate-full-command-json-input

This generates a JSON document template that shows the various options that can be provided. Not all options are mandatory; the official documentation describes the mandatory and optional parameters. You can customize this document with your own values and configurations. This JSON document now serves as a template for your cluster or your node pool; when you create new clusters or node pools, you can provide this JSON document as input instead of using a verbose CLI command that is hard to read and often difficult to reproduce. The example that follows demonstrates creating a cluster and a node pool from a preexisting JSON template:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04_images.xhtml#f119-03a)

# For the cluster resource

oci ce cluster create --from-json my-oke.json

# For the node-pool resource

oci ce node-pool create --from-json my-oke-nodes.json

The advantage of this approach is that the document that describes the configuration can be source controlled to track changes to it over time. It also enables automation workflows, in case you need to repeatedly stamp out clusters of predefined configurations.

When using the JSON-based configuration, option values can still be provided on the command line. If an option is configured in both the JSON document and the command line, the value specified on the command line takes precedence. Think of this as overriding the JSON template-based value with the one on the command line for that individual command invocation.

**Using Custom Images for Your Nodes**

Unlike using the console, using the CLI, Terraform, or the API directly enables you to also specify a custom operating system image for your nodes. Using a custom OS image for your node pool is common when you have additional software that you need to include in the base image for your Kubernetes nodes. This could be endpoint protection software such as threat monitoring and antivirus, agents for observability, or basic tools, programs, and operating system configurations that are mandated by your organization.

Under most circumstances, you want to use an official Oracle OKE image as the base for your customizations. OKE publishes operating system images for various operating system versions, with a monthly cadence. These are published as OCI images in every region, and the OCIDs of these images are published in the Oracle documentation.[**1**](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04.xhtml#ref4_1a) To use one of these images as the basis for your custom image, you can import that image and customize it with tools such as Ansible. Alternatively, you can simply create a compute instance with the desired image, perform your customizations, and then create an image from the customized compute instance. Note that you should not create an image from an existing Kubernetes node on which you have performed customizations. This is because when a compute instance has joined a Kubernetes cluster as a worker node, that node has data about the specific cluster it is part of. Creating an image from this node would also capture the cluster’s identity information, which would create a conflict when this image is used to create another node that needs to join a new Kubernetes cluster.

[Listing 4-4](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04.xhtml#list4_4) shows the CLI command to create a new compute instance using one of the publicly available OKE node images.

**Listing 4-4** Process to Create a Compute Instance Using an OKE Worker Node Image

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04_images.xhtml#f120-01a)

oci compute instance launch \

--display-name OKE\_NODE\_CUSTOM \

--compartment-id … \

--availability-domain xxxx:US-ASHBURN-AD-1 \

--shape VM.Standard.E4.Flex \

--shape-config "{\"memoryInGBs\": 8, \"ocpus\": 1}" \

--subnet-id … \

--image-id … # Image OCID for public OKE Image

This creates a compute instance using an OKE node image. This compute instance does not join any cluster because the instance was not launched by a node pool (and did not have an OKE cloud-init script) and, therefore, does not have any Kubernetes cluster information configured within it. After the compute instance has been launched, you can customize the instance with additional software or configuration. When the required customization has been done, you can generate an image from this instance that includes the customization you have performed. The image still retains the Kubernetes components, such as the kubelet, which remains uninitialized.

Now you can use your customized image to launch your OKE nodes by providing the image ID of your custom image when you create a node pool. [Listing 4-5](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04.xhtml#list4_5) shows a complete example of what this might look like.

**Listing 4-5** The Process of Creating a Node Pool with a Customized Image Using the OCI CLI

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04_images.xhtml#f121-01a)

oci ce node-pool create \

--cluster-id … \

--name my-node-pool \

--node-image-id … \ # OCID of your customized image

--compartment-id … \

--kubernetes-version v1.26.2 \

--node-shape VM.Standard.E4.Flex \

--node-shape-config "{\"memoryInGBs\": 8, \"ocpus\": 1}" \

--pod-subnet-ids "[\"…\"]" \

--placement-configs "[{\"availability-domain\":\"xxxx:US-ASHBURN-AD-2\",

\"subnet-id\":\"…\", \"faultDomains\":[\"FAULT-DOMAIN-3\", \"FAULT-

DOMAIN-1\"]}]" \

--size 1 \

--region=us-ashburn-1

**Note**

After the image has been created, it can still be edited. For instance, you can change the image’s launch mode or network attachment mode. This can be useful when you work with highly network sensitive applications that need to leverage hardware-assisted networking or SRIOV networking for the network attachments made to instances launched from this image. Changing the network attachment mode from VIRTIO mode to SRIOV mode exposes the underlying virtual function from the hardware network card to the operating system directly without involving any virtualization layers.

**Using the Terraform Provider and Modules**

Kubernetes clusters can be created with OKE using APIs and automation as well. In fact, for most production use, you should prefer the Terraform-based automation or the OCI CLI approach. The advantage of using automation is consistency and repeatability, along with all the advantages of infrastructure as code described in [Chapter 2](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02.xhtml#ch02), “[Infrastructure Automation and Management](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch02.xhtml#ch02).” OCI provides a full-featured Terraform provider that includes the Kubernetes cluster[**2**](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04.xhtml#ref4_2a)and node pool[**3**](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04.xhtml#ref4_3a) resources. Data sources[**4**](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04.xhtml#ref4_4a) also are available that can query the existing resources and their properties. In addition to the Terraform provider, OCI has made available a Terraform module[**5**](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04.xhtml#ref4_5a) that can quickly create a cluster and its associated resources as a single unit. The module also provides several preconfigured examples to get you started quickly.

[Listing 4-6](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04.xhtml#list4_6) shows a snippet of the Terraform code from the example application showcased in [Chapter 10](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch10.xhtml#ch10), “[Bringing It Together: MuShop](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch10.xhtml#ch10),” that is used to create an OKE cluster using Terraform. The Terraform configuration allows for extensive templating and customizations that enable you to create okay clusters and related resources in a consistent and configurable manner. The Terraform configuration expresses the desired state for the cluster and its associated resources. When the Terraform configuration is executed, Terraform introspects the existing resources and creates a plan that identifies the delta between the currently existing resources and how they need to be changed to match the intended configuration that is expressed in the Terraform configuration. Moreover, because Terraform tracks the state of these resources, you can use it to detect changes to these configurations over time. These changes that occur to the configuration overtime are called drift. Terraform can be rerun to return the configuration to its intended state. Note that although most properties of the resource can be updated in place, some properties of a resource might cause Terraform to delete the resource and then re-create it. When working with Kubernetes clusters, it is important to understand what these properties are so that inadvertently updating a property does not result in the deletion and recreation of a resource such as a node pool.

**Listing 4-6** Snippet of Terraform Code from an Example Application

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04_images.xhtml#f122-01a)

resource "oci\_containerengine\_cluster" "oke\_cluster" {

compartment\_id = local.oke\_compartment\_ocid

kubernetes\_version = (var.k8s\_version == "Latest") ? local.k8s\_latest : var.k8s\_version

name = "${var.app\_name} (${random\_string.deploy\_id.result})"

vcn\_id = oci\_core\_virtual\_network.oke\_vcn[0].id

endpoint\_config {

is\_public\_ip\_enabled = (var.cluster\_endpoint\_visibility == "Private") ? false

: true

subnet\_id = oci\_core\_subnet.oke\_k8s\_endpoint\_subnet[0].id

nsg\_ids = []

}

options {

service\_lb\_subnet\_ids = [oci\_core\_subnet.oke\_lb\_subnet[0].id]

add\_ons {

is\_kubernetes\_dashboard\_enabled = var.is\_kubernetes\_dashboard\_enabled

is\_tiller\_enabled = false # Default is false, left here for

reference

}

admission\_controller\_options {

is\_pod\_security\_policy\_enabled = var.is\_pod\_security\_policy\_enabled

}

kubernetes\_network\_config {

services\_cidr = lookup(var.network\_cidrs, "KUBERNETES-SERVICE-CIDR")

pods\_cidr = lookup(var.network\_cidrs, "PODS-CIDR")

}

}

image\_policy\_config {

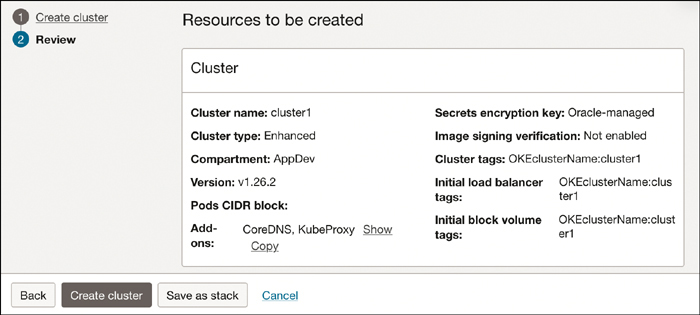
is\_policy\_enabled = false

}

}

**Automation and Terraform Code Generation**

If you are new to Terraform and the learning curve looks steep, the OCI console offers a way for you to still get the benefits of infrastructure as code. Both the Quick Create and Custom Create workflows offer a Save as Stack option at the end of the workflow (see [Figure 4-11](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04.xhtml#ch04fig11)). Choose this option to generate Terraform code that captures the configuration you specified for the cluster and its components in the console. The generated Terraform code with the configuration values is packaged as a resource manager stack. This stack can be executed from within the Oracle Resource Manager service, which runs Terraform to build a Kubernetes cluster based on the configuration that was captured. Now you can also use the features of Resource Manager, such as drift detection, to ensure that your clusters’ configuration is not drifting from its expected values due to manual or ad-hoc changes.



**Figure 4-11** Summary of the Resources That Will Be Created Using the Oracle Resource Manager Service; the Save as Stack Generates a Terraform Configuration from the Options Provided to the Wizard

**Asynchronous Cluster Creation**

Regardless of the method you use to create a cluster, the act of setting up a new Kubernetes cluster control plane, and subsequently the data plane, is asynchronous. This means that the cluster creation API returns immediately (regardless of whether it is invoked through the console, CLI, or Terraform) and the cluster moves into a provisioning state. The progress of cluster creation can be tracked through a work request that is available under the cluster details page. The data plane or the node pools are created when the control plane creation is complete. In most cases, however, you do not need to wait for the data plane to be fully provisioned before deploying workloads. When the Kubernetes API endpoint is available, you can deploy pods or other Kubernetes resources to the cluster even if no data plane nodes are available. The resources simply remain in pending status until those data plane nodes are available.

**Cluster Topology Considerations**

OKE offers a set of flexible options in structuring your cluster topology. Various choices can help you decide how and where you create your data plane nodes, to meet your objectives. For example, you might choose to have a topology with a node pool that places its nodes across availability domains, a topology that restricts the node to a single availability domain, or a topology for which you need multiple node pools, each connected to a separate subnet for network-level isolation. The Node Pool resource in OKE provides these options for creating a wide range of topologies.

**Using Multiple Node Pools**

Because each node pool can be configured with a unique set of parameters, using multiple node pools enables you to deploy workloads running in the same cluster to the pool that best matches the needs of the workloads. Using multiple OKE node pools can provide more flexibility and control over your Kubernetes cluster, allowing you to optimize resources, improve security, increase availability, and scale your applications more efficiently. Specifically:

* **Resource requirements:** Each node pool can be configured with a specific worker node’s shape, depending on the needs of your workloads. For example, if you have some workloads that require GPU resources and others that require high memory resources, you might create two separate node pools, each with the shape optimized for the specific use case.
* **Availability:** Each node pool can be configured with placement configurations that control the distribution of nodes across availability domains (ADs) and fault domains (FDs). By creating multiple node pools with worker nodes in different ADs and/or FDs, you can ensure that your cluster is more resilient to failures. If one AD or FD goes down, your workloads can still run on the nodes in the other pools.
* **Cost optimization:** You might use multiple node pools to optimize the cost of running your cluster. By using different types of worker nodes with different prices, you can save money by paying for only the resources that you actually need. For example, you could use nodes backed by low-cost preemptible instances for fault-tolerant or development workloads and then use higher-cost nodes for production workloads that require higher performance.
* **Security and isolation:** You might also use multiple node pools to improve the security of your cluster. For example, you could create a dedicated node pool for workloads that need to be isolated from other workloads for security reasons. This can be achieved by running the workload on a separate set of nodes with stricter access controls.

**Scheduling Workloads on Specific Nodes**

Under most circumstances when using Kubernetes, you should let the Kubernetes scheduler manage which nodes in the cluster are chosen for your workloads. The Kubernetes scheduler examines your pod resource requests and matches them with nodes that can satisfy those resource requests. This also allows the control plane to reschedule workloads onto available capacity when failures are detected. This automated and hands-off approach to workload management is one of the major advantages of using Kubernetes. However, in some scenarios, you need to exert control over what nodes your workload can be scheduled on. For instance, if your workload requires access to specialized hardware on specific nodes, you want to make this known to the Kubernetes control plane so that the scheduler can take this into consideration when making scheduling decisions. To schedule your workloads to specific Kubernetes node pools in OKE, you can use node selectors or node affinity, taints and tolerations, and other Kubernetes constructs, such as topology spread constraints.

*Node selectors* enable you to specify a label on a node pool and then use that label in your pod specification to select the node pool where you want to schedule your workload. For example, if you have a node pool labeled as gpu with GPU shapes selected, you can use the following node selector in your pod specification:

spec:

nodeSelector:

nodepool: gpu

This ensures that your pod is scheduled on a node in the gpu node pool. The same approach can be used with other node shapes, such as ARM.

*Node affinity* enables you to specify more complex rules to match nodes based on labels or other attributes. For example, [Listing 4-7](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04.xhtml#list4_7) demonstrates using node affinity to match nodes with a specific label and not match nodes with another label.

**Listing 4-7** A Manifest File That Includes a Node Affinity and Expression to Match the Pod to a Node with the gpu Node Selector

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04_images.xhtml#f126-01a)

spec:

affinity:

nodeAffinity:

requiredDuringSchedulingIgnoredDuringExecution:

nodeSelectorTerms:

- matchExpressions:

- key: nodepool

operator: In

values:

- gpu

- key: nodepool-type

operator: NotIn

values:

- preemptible

This ensures that your pod is scheduled on a node in the gpu node pool that is not labeled as preemptible.

*Taints and tolerations* are a way for you to influence the scheduler and make sure that pods are not placed on nodes that need to be reserved for more critical workloads. For instance, if you have GPU nodes in your cluster, you likely want to deploy workloads that take advantage of those GPUs onto those nodes. A pod that selects a GPU node with a node selector will be assigned to an available GPU node, but this does not prevent other workloads that do not leverage or need the GPUs from being scheduled on the GPU node as well. For instance, a MySQL pod that does not use the GPU can also be scheduled on the GPU node. This can potentially lead to resource exhaustion because the MySQL pod has been allocated CPU resources, and it does not consume GPUs. If a few of these pods are allocated to the GPU node, when a real GPU workload is deployed, it could fail to schedule because there are no available CPUs on the GPU node, even though all the GPUs might be free. In these cases, you need a way for these specific nodes (such as the GPU node) to reject or repel a pod (such as the MySQL pod) unless the pod is specifically designed to use the node, like an actual GPU workload. This is exactly what taints and tolerations do.

Taints are added to nodes, and tolerations are defined in the pod specification. When you taint a node, it will repel all the pods except those that have a toleration for that taint. A node can have one or many taints associated with it.

*Pod topology spread constraints* offer you a declarative way to configure pod scheduling that is based on some topology key. This is done by grouping the nodes into “domains,” on the basis of having the same node label and value. In the example in [Listing 4-8](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04.xhtml#list4_8), the topology key used is kubernets.io/arch. This is a well-known label that kubelet sets based on the CPU architecture of the node. If the cluster had both x86- and ARM-based nodes, the x86 nodes could have the value amd64 and the ARM nodes could have the value arm64 for the same node label. Pod topology spread constraints use this to split the nodes into two groups or domains, the x86 nodes and the Arm nodes. The scheduler tries to achieve balance among all the groups, so in the example, a similar number of pods will be allocated across each group. Similarly, by choosing another key, such as the availability domain or the fault domain, you can spread your pods across these groups, to prevent pods from being co-scheduled and potentially leading to a larger impact in case of a disruption.

**Listing 4-8** Manifest File That Includes a Topology Spread Constraint, a Topology Key, and the Action to Take Depending on Whether the Key Is Present

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04_images.xhtml#f127-01a)

spec:

topologySpreadConstraints:

- maxSkew: 1

topologyKey: kubernetes.io/arch

whenUnsatisfiable: DoNotSchedule

labelSelector:

matchLabels:

app: wordpress

**Kubernetes Networking**

OKE relies on Oracle Cloud Infrastructure (OCI) virtual cloud networks (VCN). These are virtual versions of traditional network architectures that enable you to connect your OCI resources, such as compute instances and storage volumes, to each other and to the Internet. VCNs provide a logically isolated network environment in the cloud that you can customize to meet your needs. VCNs allow you to define the IP address range for your network, create subnets, and specify security rules to control inbound and outbound traffic.

OKE uses VCNs to provide a secure, isolated networking environment for Kubernetes clusters. When you create a Kubernetes cluster using OKE, you can choose to create a new VCN through the cluster Quick Create workflow or use an existing one through the Custom Create workflow. If you choose to create a new VCN, OKE creates a new VCN with a default set of subnets, security lists, and routing tables. If you choose to bring your own VCN, it is important to make sure that the rules are set up correctly, to ensure proper communication throughout your cluster.

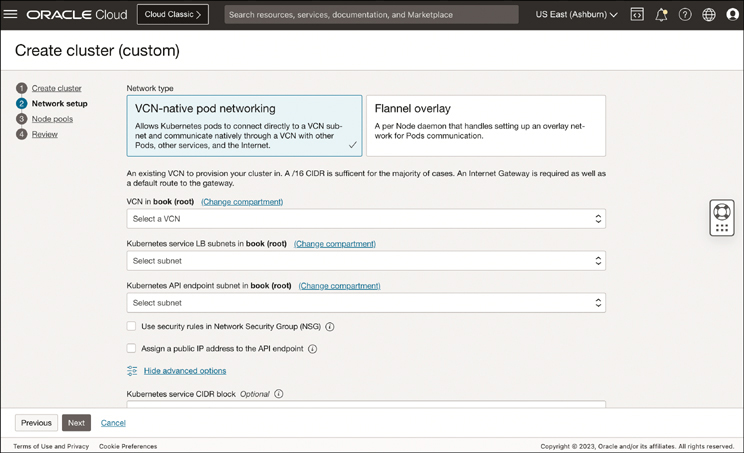
Each node in the cluster is launched in a subnet. OKE supports network security policies, which allow you to define fine-grained rules to control traffic between pods in your Kubernetes cluster. These policies are enforced by the Kubernetes network plug-in running on each node in the cluster.

**Container Network Interface (CNI)**

The Kubernetes networking model assumes that pods have unique and routable IP addresses within a cluster. In the Kubernetes networking model, pods use those IP addresses to communicate with each other, the cluster control plane, other resources (for example, storage), and the Internet. Kubernetes clusters use Container Network Interface (CNI) plug-ins for network resource management, such as to implement network connectivity for pods running on worker nodes. The CNI consists of a specification and libraries for writing plug-ins to configure network interfaces in Linux containers, along with a number of supported plug-ins. CNI plug-ins configure network interfaces, provision IP addresses, and maintain connectivity. All the node pools in a cluster use the same CNI plug-in.

OKE supports two types of pod networking out of the box: VCN-native pod networking and Flannel overlay (see [Figure 4-12](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04.xhtml#ch04fig12)). When creating a Kubernetes cluster with OKE, the network type you select for the cluster determines the CNI plug-in that is used for pod networking.

* Choosing VCN-native pod networking as the network type deploys the OCI VCN-Native Pod Networking CNI plug-in to your nodes.
* Choosing the Flannel overlay network type deploys the flannel CNI plug-in to your nodes.



**Figure 4-12** Process of Choosing a CNI During the Cluster Creation Process

**Note**

You can use the OCI VCN-Native Pod Networking CNI plug-in with both virtual node pools and managed node pools. You can use the Flannel CNI plug-in with managed node pools.

**OCI VCN-Native Pod Networking CNI**

The OCI VCN-Native Pod Networking CNI plug-in uses the VCN’s CIDR block to provide IP addresses to pods and enables other resources within the same subnet (or a different subnet) to communicate directly with pods in a Kubernetes cluster. Pod IP addresses are directly routable from other VCNs connected to that VCN, from on-premises networks, and from the public Internet.

Because pods are directly routable, you can use *native* VCN functionality to control access to and from pods using security rules defined as part of network security groups or security lists. The security rules apply to all pods in all the worker nodes connected to the pod subnet specified for a node pool. You can also use VCN flow logs to observe the traffic to, from, and between pods, which is useful for troubleshooting and compliance auditing purposes. This also enables you to use route tables and routing rules to route incoming requests to pods based on routing policies. Apart from these management features, because pods are directly connected to the virtual cloud network, no encapsulation is involved in packet transmission; this generally offers consistent performance for workloads that are sensitive to small amounts of latency.

Worker nodes running in clusters using the OCI VCN-Native Pod Networking CNI plug-in are connected to two subnets specified for the node pool: a *worker node subnet* and a *pod subnet*. The worker node subnet supports communication between processes running on the cluster control plane (such as kube-apiserver, kube-controller-manager, and kube-scheduler) and processes running on the worker node (such as kubelet and kube-proxy). The worker node subnet can be private or public and can be a regional subnet or an AD-specific subnet. The pod subnet supports communication between pods and direct access to individual pods using private pod IP addresses. The pod subnet can be private or public, and it must be a regional subnet. The pod subnet enables communication between pods on the same worker node or on other worker nodes, with OCI services (through a service gateway), and with the Internet (through a NAT gateway). You specify a single pod subnet for all the pods running on worker nodes in a node pool. You can specify the same pod subnet or different pod subnets for different node pools in a cluster. You can specify the same pod subnet for node pools in different clusters. The worker node subnet and the pod subnet must be in the same VCN and can be the same subnet. If they are in the same subnet, you should define security rules in network security groups to route network traffic to worker nodes and pods.

Something important to note about VCN-Native Pod Networking is that you might find yourself limited by the number of VNICs available to your chosen worker node shape. A minimum of two VNICs are attached to each worker node: One is connected to the worker node subnet, and the other is connected to the pod subnet. By default, 31 IP addresses are assigned to the VNIC for use by pods running on the worker node. These IP addresses are preallocated in the pod subnet before pods are created in the cluster. If you want more than 31 pods on a single worker node, the shape you specify for the node pool must support more than the minimum two VNICs. The additional VNICs can be connected to the pod subnet, to provide further IP addresses to support more pods. Similarly, VCN-Native Pod Networking consumes IP addresses from the virtual cloud network, which can pose challenges when working in an environment with a constrained IP space.

**Note**

Regardless of whether you are using the add-on API, a feature used to gain control over operational software deployed to OKE clusters, Oracle is responsible for deploying updates to the OCI VCN-Native Pod Networking CNI plug-in. The updates are applied only when worker nodes are next rebooted.

**Flannel CNI**

The Flannel overlay network is a simple private overlay virtual network that attaches IP addresses to containers. The Flannel overlay network uses its own CIDR block to provision pods and worker nodes with IP addresses instead of using IP addresses from a VCN’s CIDR block. Because the pods in the private overlay network are accessible only from other pods in the same cluster, you can specify the same Flannel CIDR block for multiple clusters.

Because Flannel provides overlay network, it can be advantageous when you have a lot of pods, each of which requires an IP address, and your network has limited IP space in your cloud network. In these cases, Flannel creates the pods on its own overlay network, and your IP space is left untouched. In other words, if the density of pods per node presents an obstacle using the OCI VCN-Native Pod Networking CNI, consider using the Flannel CNI plug-in because the number of pods per worker node is not determined by the node shape. The disadvantage of using Flannel is that it can involve packet encapsulation, which incurs a performance hit and might not be appropriate for workloads with a high sensitivity to the network performance.

**Kubernetes Storage**

When a container is deleted or re-created, data stored inside the root file system can disappear. You can use persistent volumes (PVs) to store data outside containers to prevent data loss. PVs are simply storage in the cluster provisioned either dynamically using storage classes or by an administrator. You can think of it as a resource in a cluster just like a node is a resource in a cluster. Persistent volumes provide a mechanism for keeping your stored data intact even when the containers using the storage are terminated. To request persistent storage, you create a persistent volume claim (PVC), which is then bound to a persistent volume. Whereas a pod is a request by a user to consume node resources, a PVC is a request by a user to consume storage resources in the form of persistent volumes. Just as users can request specific levels of CPU and memory resources through pods, users can request specific size and access modes, including ReadWriteOnce, ReadOnlyMany, and ReadWriteMany, through persistent volume claims.

OKE provides two options for provisioning PVCs for OCI resources:

* The Oracle Cloud Infrastructure Block Volume service, which uses either the FlexVolume or CSI (Container Storage Interface) volume plug-ins to connect to OKE clusters
* The Oracle Cloud Infrastructure File Storage service, which uses the CSI volume plug-in to connect to OKE clusters

This section discusses the various storage options available to OKE, including their pros and cons and ways to use these storage options effectively for your workloads. Because OKE has the capability to create and manage these storage resources on your behalf, you need to configure OCI IAM policies to work with storage services. The policies that are required can vary, depending on the topology that you set up; see the official product documentation.

**StorageClass: Flex Volume and CSI Plug-ins**

A StorageClass provides a way for cloud providers or administrators to describe the “classes” of storage that they offer. The StorageClass specified for a PVC controls which volume plug-in to use to connect to Block Volume service volumes. Two storage classes are defined by default, oci-bv for the CSI volume plug-in and oci for the FlexVolume plug-in. If you don’t explicitly specify a value for storageClassName in the YAML file that defines the PVC, the cluster’s default storage class is used. The cluster’s default storage class is initially set according to the Kubernetes version that was specified when the cluster was created. Before Kubernetes 1.24, this was the oci StorageClass used for the FlexVolume plug-in. For Kubernetes 1.24 and onward, this is oci-bv StorageClass for the CSI volume plug-in. OKE adds new functionality only to the CSI volume plug-in. The CSI plug-in comes with benefits, including the CSI topology feature, which ensures that worker nodes and volumes are colocated in the same availability domain, and the CSI volume plug-in does not require access to underlying operating system and root file system dependencies.

**Updating the Default Storage Class**

For clusters created on Kubernetes version 1.23 and earlier, and subsequently upgraded to Kubernetes version 1.24 or later, the default storage class is not changed during the upgrade process. This means that unless you manually update your default storage class, it remains oci. To shift from oci to oci-bv as the default storage class, perform the following steps.

This removes oci as the default storage class:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04_images.xhtml#f131-01a)

kubectl patch storageclass oci -p '{"metadata": {"annotations": {"storageclass.

beta.kubernetes.io/is-default-class":"false"}}}'

This adds oci-bv as the default storage class:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04_images.xhtml#f131-02a)

kubectl patch storageclass oci-bv -p '{"metadata": {"annotations":

{"storageclass.kubernetes.io/is-default-class":"true"}}}'

When provisioning a persistent volume claim, you can explicitly specify the volume plug-in to use to connect to the Block Volume service. This is done by specifying a value for storageClassName in the YAML file that defines the PVC:

* Specify storageClassName: "oci-bv" to use the CSI volume plug-in.
* Specify storageClassName: "oci" to use the FlexVolume volume plug-in.

If the cluster administrator has not created any suitable PVs that match the PVC request, you can dynamically provision a block volume using the CSI plug-in specified by the oci-bv storage class’s definition (provisioner: blockvolume.csi.oraclecloud.com). For example, you can define a PVC in a file called example-csi-pvc.yaml, as demonstrated in [Listing 4-9](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04.xhtml#list4_9).

**Listing 4-9** Manifest File Used to Create a Persistent Volume Claim

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04_images.xhtml#f132-01a)

apiVersion: v1

kind: PersistentVolumeClaim

metadata:

name: exampleclaim

spec:

storageClassName: "oci-bv"

accessModes:

- ReadWriteOnce

resources:

requests:

storage: 50Gi

Then you can create the PV:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04_images.xhtml#f132-02a)

kubectl create -f example-csi-pvc.yaml

You can verify that the PVC was created by running the following:

kubectl get pvc

You can use this PVC when creating other Kubernetes objects, such as pods. For example, you can use the pod definition in [Listing 4-10](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04.xhtml#list4_10) to create a pod that uses the exampleclaim PVC as the volume, which is mounted at /data by the pod.

**Listing 4-10** Example Manifest File Used to Create a Pod That Uses the Previously Created Persistent Volume Claim

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04_images.xhtml#f132-03a)

apiVersion: v1

kind: Pod

metadata:

name: example

spec:

containers:

- name: example

image: example:latest

ports:

- name: http

containerPort: 80

volumeMounts:

- name: data

mountPath: /usr/share/example/html

volumes:

- name: data

persistentVolumeClaim:

claimName: exampleclaim

You can verify that the pod is using the new persistent volume claim by entering the following command:

kubectl describe pod example

**File System Storage**

The Oracle Cloud Infrastructure File Storage Service (FSS) provides a scalable and distributed network file system that uses the NFS v3 protocol. This makes FSS ideal for Kubernetes use cases in which shared storage is required. FSS also scales dynamically without any upfront provisioning, making it simple to use and scale. The CSI volume plug-in that is included with OKE provides support for Kubernetes Persistent Volumes that are backed by the FSS. These persistent volumes can be shared by pods that simultaneously call all reads and writes to the volume, otherwise known as the ReadWriteMany (RWX) access mode.

Before you can successfully leverage FSS storage in your Kubernetes cluster, it is important to understand FSS-specific terminology. Because FSS is an NFS V3 file system, you need an IP address or a DNS name that you can use to mount the file system. This is provided by a *mount target* in OCI. A mount target can be used to make multiple file systems available to users. An NFS client connects to the mount target to access a file system. An *export* controls how an NFS client accesses the file system when connecting to a mount target. Exporting is the act of making a file system available through a mount target. A file system must have at least one export in a mount target for instances to access and mount that file system. When you export a file system, a path to uniquely identify the file system within the mount target is specified. You can associate multiple file systems to a single mount target; this path is called the export path.

You can use the CSI volume plug-in to connect clusters to file systems created by the File Storage service. The CSI volume plug-in supports dynamic provisioning to dynamically create the required resources, such as the mount target and the file system itself, when a persistent volume claim is presented to the cluster. Although this dynamic provisioning capability is handy, FSS file systems are usually long-lived storage solutions that are used by multiple pods as durable shared storage. For this typical use case, it is more desirable to mount and use an existing file system than to create new file systems in an ephemeral fashion. The CSI volume plug-in also supports this model of using a preexisting file system and mount target.

To work with the File Storage service using the CSI volume plug-in, you need to define a storage class that sets up the parameters required when creating and managing file systems and mount targets. The StorageClass definition provides a template for creating the underlying resources. Consider the StorageClass definition in [Listing 4-11](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04.xhtml#list4_11).

**Listing 4-11** Manifest Used to Define a StorageClass for Managing FSS File Systems

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04_images.xhtml#f134-01a)

kind: StorageClass

apiVersion: storage.k8s.io/v1

metadata:

name: oci-fss

provisioner: fss.csi.oraclecloud.com

parameters:

availabilityDomain: US-ASHBURN-AD-1

mountTargetOcid: …

compartmentOcid: …

kmsKeyOcid: …

exportPath: /shared

exportOptions: [ { "source" : "0.0.0.0/0", "requirePrivilegedSourcePort" :

false, "access" : "READ\_WRITE", "identitySquash" : "ROOT" } ]

encryptInTransit: "false"

This storage class can create new persistent volumes backed by the file system service. The file systems are created in the availability domain specified in the configuration. The mountTargetOcid specifies the amount target to use for the file systems created by this storage class. Instead of providing the mountTargetOcid, you can also provide a mountTargetSubnetOcid. If the mountTargetSubnetOcid is provided instead of the mountTargetOcid, a new mount target is created in the given subnet. Note that, under most circumstances, you want to use a mountTargetOcid. The number of mount targets can be limited, and you typically want to associate multiple file systems to a single mount target. The kmsKeyOcid specifies the OCID for your own encryption key that is managed in the OCI vault, for encrypting the data stored on the volume. Data in the file system service is always encrypted, and this option replaces the Oracle managed key that is used by default to encrypt your data in the file system service with your own key. The exportPath and the exportOptions determine how the file system is made available through the mount target.

With the storage class created, you can now use a persistent volume claim that refers to this storage class to dynamically create the file system, export it, and attach it to your workload. Consider the persistent volume claim definition in [Listing 4-12](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04.xhtml#list4_12).

**Listing 4-12** Manifest Used to Define a Persistent Volume Claim for an FSS File System

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04_images.xhtml#f134-02a)

apiVersion: v1

kind: PersistentVolumeClaim

metadata:

name: fss-claim

spec:

accessModes:

- ReadWriteMany

storageClassName: "oci-fss"

resources:

requests:

storage: 100Gi

This volume claim specifies a storage class name but not a persistent volume name. This means that the CSI volume driver uses dynamic provisioning to create the required resources, such as the file system. The example references the storage class OCI FSS shown in the prior listing, which specifies a mount target OCID. If the storage class is configured with a mountTargetSubnetOcid, the CSI volume driver also creates a mount target in the given subnet as part of the dynamic provisioning process, to satisfy this volume claim.

It is worthwhile to note that, although you need to set a storage size that is required by the PersistentVolumeClaim object, this size has no effect on the file system. This is because the OCI file system service transparently scales on demand as more data is written to the file system.

When using dynamic provisioning as shown in [Listing 4-11](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04.xhtml#list4_11), the export path configured on the storage class is used for exporting the file system that is created. The export path determines the path under the mount point where the file system will be made available. Therefore, when using this approach, you are usually limited to creating only a single file system with a given storage class object. This is because the export path for each file system has to be unique within a mount target, and the storage class sets this statically. This means that there can be only one volume claim referring to this storage class for dynamic provisioning; a second volume claim using this storage class for dynamic provisioning will create a file system and would attempt to use the same export path, which would result in an error. If you used the mountTargetSubnetOcid as well, you can create a new mount target each time you use dynamic provisioning; however, you might hit the tenancy limit for the number of mount targets you can have.

For these reasons, in most cases, you typically will want to pre-create a mount target and reference that in the storage class. You can also leave out the **exportPath** parameter from the storage class, which will result in a new PV and file system being generated dynamically each time a PVC is created. The new file system will be exposed by the mount target with a default export path that corresponds to the display name of the file system generated by the CSI volume plugin. Under most circumstances, this will be the behavior you need.

You could also exert more control by provisioning all persistent volumes beforehand. The following example shows how to create a persistent volume object and reference it from your persistent volume claim to get more control over the resources.

To use an existing mount target and a file system, you must create a persistent volume definition that refers to these existing resources. Consider the example in [Listing 4-13](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04.xhtml#list4_13).

**Listing 4-13** Manifest Used to Define a Persistent Volume Backed by an FSS File System

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04_images.xhtml#f136-01a)

apiVersion: v1

kind: PersistentVolume

metadata:

name: fss-volume

spec:

capacity:

storage: 100Gi

volumeMode: Filesystem

accessModes:

- ReadWriteMany

persistentVolumeReclaimPolicy: Retain

csi:

driver: fss.csi.oraclecloud.com

volumeHandle: ocid1.filesystem.xxx:10.0.0.1:/shared

The persistent volume definition describes a volume that can be used by many pods as shared storage due to the ReadWriteMany access mode. The CSI section of the definition specifies the driver to be used and the volume handle. The volume handle points to the existing resources to be used for this volume. It has three elements to it: the OCID of the file system, the IP address of the mount target, and the export path. Once a persistent volume representing an existing file system and the options for accessing it has been created, you can use a persistent volume claim to bind to this volume. Consider the example in [Listing 4-14](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04.xhtml#list4_14), which demonstrates using a persistent volume claim that refers to this persistent volume.

**Listing 4-14** Manifest Used to Create a Persistent Volume Claim Bound to an Existing Persistent Volume Backed by an FSS File System

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04_images.xhtml#f136-02a)

apiVersion: v1

kind: PersistentVolumeClaim

metadata:

name: fss-claim

spec:

accessModes:

- ReadWriteMany

storageClassName: ""

resources:

requests:

storage: 100Gi

volumeName: fss-volume

This volume claim is similar to the one used for dynamic provisioning, with a few notable differences. The first obvious difference is that the volume name attribute is set; it directly refers to the persistent volume that has been created to interact with the preexisting file system. This binds the persistent volume to this claim after verifying the storage class and the storage requirements. The second difference is that the storage class name has been set to an empty string. This explicitly disables dynamic provisioning for this volume claim. Note that omitting the storage class from the configuration is not the same as setting it to an empty string. This is because, when the storage class is omitted, the default storage class is used, if one is configured. Setting the storage class to the empty string requires the persistent volume claim to be bound to a persistent volume that has no storage class. As before, the storage request does not have any impact on the actual file system; the storage request is considered when binding the persistent volume to the persistent volume claim. If the volume specifies a storage capacity that is lower than the requested capacity in the claim, the volume claim will not be bound to the volume. Therefore, it is best to use the same storage capacity values for both the volume and the volume claim when using FSS with your Kubernetes cluster.

**Kubernetes Load Balancer Support**

When you create a service that has its type set to LoadBalancer, Kubernetes attempts to create a load balancer to expose the service. When running on a cloud provider, the Cloud Controller Manager (CCM) is responsible for calling the appropriate APIs on the cloud provider to create and wire the load balancer to the underlying pods. The OCI CCM performs this task on OKE and enables you to manage traffic and the properties of the load balancers. The OCI CCM runs on the OKE control plane and is present on all OKE clusters.

OCI offers two types of load balancers: the standard Load Balancer service, which offers Layer 7 capabilities, and a network load balancer, which offers Layer 4 (TCP/UDP) level load balancing. When creating Service objects, you can add *annotations* to the Service definition’s metadata that tell the CCM to create and set up the load balancer in a certain way. Annotations are additional name-value elements in the spec that are read by the CCM that control its behavior. The complete list of annotations and configuration parameters is provided in the CCM documentation.[**6**](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04.xhtml#ref4_6a) The annotations for the Layer 7 Load Balancer service are with either service.beta.Kubernetes.io or oci.oraclecloud.com (for OCI-specific features), and the annotations for network load balancers are prefixed with oci-network-load-balancer.oraclecloud.com.

**Working with the OCI Load Balancer Service**

The OCI Load Balancer service is the default choice used by the OCI CCM when creating service objects that are of type LoadBalancer. This OCI service offers a highly available and fault-tolerant proxy that can be located across multiple availability domains. Because this is a Layer 7 load balancer service, it can support advanced HTTP routing policies, and it has additional features, such as SSL termination. This is a flexible infrastructure service, meaning that the service automatically scales between a minimum bandwidth value that is always guaranteed and an optional maximum bandwidth value as required by actual real-time traffic, without any intervention. It offers you a choice of public or private IP addresses and is appropriate for load balancing most applications.

When a cluster is created, the user is prompted to allocate a subnet for placing load balancers. The user also has the choice of configuring the subnets for the node pools. When a service of type LoadBalancer is created, the CCM creates a load balancer within the subnet and wires them to the pods located on the compute nodes within the node pools. The traffic flow among these various subnets is governed by the VCN’s network security groups and security lists. Traffic from external sources such as clients and users outside the OKE cluster, or even the network, is managed and routed by the various configurations for the NSGs and security lists. In OKE, you can decide whether to let the CCM automatically configure these elements for you so that, as you deploy a service, the associated network configuration is updated to route traffic to it by opening the required ports and adding the required access rules. Alternatively, you can manage this yourself if you want a more predictable configuration and you do not want to provide access to the service to make changes to your network’s traffic and security settings. The configuration choices are controlled by a set of annotations that you can provide in the *ServiceSpec* or the YAML that defines the service.

To understand the various annotations, their effects, and how best to use them, consider [Listing 4-15](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04.xhtml#list4_15), which is the most basic service definition for a Kubernetes service, along with a pod that it can point to.

**Listing 4-15** Manifest Used to Create an NGINX Pod, Along with a Service of Type LoadBalancer, Backed by the OCI Load Balancer Service

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04_images.xhtml#f138-01a)

apiVersion: v1

kind: Pod

metadata:

name: nginx

labels:

app.kubernetes.io/name: proxy

spec:

containers:

- name: nginx

image: nginx:stable

ports:

- containerPort: 80

name: http-web-svc

---

apiVersion: v1

kind: Service

metadata:

name: nginx-service

spec:

type: LoadBalancer

selector:

app.kubernetes.io/name: proxy

ports:

- name: http

protocol: TCP

port: 80

targetPort: http-web-svc

This creates a single pod that runs nginx, pulling the image tagged stable. The pod is labeled with app.kubernetes.io/name: proxy. Next, the Service definition or the ServiceSpec defines the service as type: LoadBalancer. The selector causes the service to route traffic to pods with the label app.kubernetes.io/name: proxy. Port 80 on the Service (load balancer) is exposed, and it points to the port named http-web-svc exposed by the pod (also port 80). So much is clear from the definition. When OKE encounters this service spec, it sets up the required infrastructure resources, such as the actual load balancer, listeners, health checks, security rules, and more. Because you did not specify any annotations, the CCM uses the default values for its configuration and sets up the following resources:

* A Layer 7 load balancer, 100Mbps
* An ephemeral public IP, if the load balancer subnet is a public subnet (chosen at cluster creation)
* Instructions to round-robin among all back ends (this example has just a single one)
* Updates to security lists for both the load balancer and the node subnets
  + On the node subnet’s security list, an ingress rule is added to allow for ingress on the host port that is opened for pod traffic.
  + On the load balancer subnet, an ingress rule for port 80 is added where the service is exposed externally, and egress rules are added to enable egress from the load balancer to the node subnet on the host port for pod traffic.

Although OKE offers some secure and sensible defaults, in most circumstances, you need to exert some control over these defaults or be explicit in these configurations, for better visibility and predictability with the configuration in a production setting. The previous service spec can be rewritten explicitly as shown in [Listing 4-16](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04.xhtml#list4_16).

**Listing 4-16** How to Add Annotations to the Load Balancer Manifest to Configure Your LoadBalancer Service

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04_images.xhtml#f139-01a)

... Pod Spec Truncated ...

---

apiVersion: v1

kind: Service

metadata:

name: nginx-service

annotations:

oci.oraclecloud.com/load-balancer-type: "lb"

service.beta.kubernetes.io/oci-load-balancer-security-list-management-mode :

"All"

service.beta.kubernetes.io/oci-load-balancer-shape: "flexible"

service.beta.kubernetes.io/oci-load-balancer-shape-flex-min: "100"

service.beta.kubernetes.io/oci-load-balancer-shape-flex-max: "100"

oci.oraclecloud.com/loadbalancer-policy:"ROUND\_ROBIN"

spec:

type: LoadBalancer

selector:

app.kubernetes.io/name: proxy

ports:

- name: http

protocol: TCP

port: 80

targetPort: http-web-svc

**Note**

The annotation service.beta.kubernetes.io/oci-load-balancer-shape can select a fixed shape, such as 100Mbps. However, Oracle plans to deprecate these fixed shapes in the future. Using flexible load balancers with the upper and lower bounds set to the same number is a way to achieve a similar configuration.

**SSL Termination with OCI Load Balancer**

The OCI Load Balancer service supports SSL termination at the load balancer. When using Kubernetes services, you can configure and manage this directly from the Kubernetes cluster using standard Kubernetes tooling. To set this up, you need to leverage the following two annotations:

* service.beta.kubernetes.io/oci-load-balancer-ssl-ports This annotation configures the ports to enable SSL termination on the corresponding load balancer listener.
* service.beta.kubernetes.io/oci-load-balancer-tls-secret This annotation references a TLS secret, which is a built-in secret type in Kubernetes for storing certificates and their associated keys. You need to create a Kubernetes secret of type kubernetes.io/tls and populate it with the certificate and the private key. Then refer to the secret object by its name in this annotation to install the certificate on the load balancer listeners and have SSL enabled.

To examine this in action, consider the example in [Listing 4-17](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04.xhtml#list4_17).

**Listing 4-17** The Same Manifest Previously Used to Define the LoadBalancer Service, but Now with Additional Annotations and an HTTPS Port

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04_images.xhtml#f141-01a)

... Pod Spec Truncated ...

---

apiVersion: v1

kind: Service

metadata:

name: nginx-service

annotations:

oci.oraclecloud.com/load-balancer-type: "lb"

service.beta.kubernetes.io/oci-load-balancer-security-list-management-mode :

"All"

service.beta.kubernetes.io/oci-load-balancer-shape: "flexible"

service.beta.kubernetes.io/oci-load-balancer-shape-flex-min: "100"

service.beta.kubernetes.io/oci-load-balancer-shape-flex-max: "100"

oci.oraclecloud.com/loadbalancer-policy:"ROUND\_ROBIN"

service.beta.kubernetes.io/oci-load-balancer-ssl-ports: "443"

service.beta.kubernetes.io/oci-load-balancer-tls-secret: ssl-certificate-

secret

spec:

type: LoadBalancer

selector:

app.kubernetes.io/name: proxy

ports:

- name: http

protocol: TCP

port: 80

targetPort: http-web-svc

- name: https

port: 443

targetPort: http-web-svc

This is the same service definition as in the earlier example, with the two new added annotations and an added port for HTTPS. Configuring SSL termination requires that you configure the SSL certificate on the load balancer. When configuring SSL termination at the load balancer for a Kubernetes service, Kubernetes can configure the SSL certificate for the load balancer. The expectation here is that the required SSL certificate and private key is provided to the CCM as a Kubernetes secret of type TLS. Kubernetes provides a built-in secret type kubernetes.io/tls for storing a certificate and its associated keys that are typically used for TLS. The annotation service.beta.kubernetes.io/oci-load-balancer-tls-secret should point to this secret, and you can see this referencing ssl-certificate-secret in [Listing 4-17](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04.xhtml#list4_17). The secret itself should have a key and a certificate that contains the private key and the certificate that you want to use. This can be created with the --key and the --cert flags for kubectl create. For example, if there were a certificate and key named tls.crt and tls.key in the current directory, you could use the following command to create the secret of type kubernetes.io/tls:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04_images.xhtml#f142-01a)

kubectl create secret tls ssl-certificate-secret --key tls.key --cert tls.crt

**Working with the OCI Network Load Balancer Service**

The network load balancer is a nonproxy load balancer in OCI that performs pass-through load balancing of Layer 4 (TCP/UDP/ICMP) traffic. It is a highly available load balancer that provides a regional virtual IP (VIP) address. The load balancer can elastically scale without requiring a minimum or maximum bandwidth configuration. It also provides the benefits of flow logs and source and destination IP address preservation. The network load balancer does not directly respond to a client ICMP or TCP/UDP ping packet. Instead, the network load balancer directs the packet to a back-end server in accordance with the load balancing policy. The back-end server then returns a response to the client. The primary advantages of the network load balancer are its capability to preserve source and destination IP addresses, low latency and high throughput load balancing, and the ability to handle UDP traffic. To choose network load balancer when creating a Kubernetes service, you can set the annotation oci.oraclecloud.com/load-balancer-type: nlb on your service definition, as shown in [Listing 4-18](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04.xhtml#list4_18).

**Exposing UDP Applications and Preserving IP Addresses**

If you choose the network load balancer to expose a UDP application or to preserve IP addresses, you need additional configuration to support these use cases. For exposing UDP applications, you need to set the protocol field in the service definition’s port configuration. The default value for protocol is TCP.

Similarly, to preserve source IP addresses, you need to configure the externalTrafficPolicy parameter for the service and set up your security lists to allow traffic from the source IP range. Although enabled by default, you use the annotation oci-network-load-balancer.oraclecloud.com/is-preserve-source: "true" to explicitly enable source IP preservation. The externalTrafficPolicy is set to Local to ensure that Kubernetes does not relay the request through other nodes, which is the default behavior. Services that want to have the source IP preserved should always include this parameter and set externalTrafficPolicy: Local in the service definition. Additionally, the security list or the NSG rules for the nodes also need to allow traffic from these original sources to reach the nodes. This is different than with the Layer 7 load balancer because, in that case, the clients communicate with the load balancer, which then proxies the requests to the nodes. The nodes thus would be receiving traffic from the load balancer, and the security lists or NSG rules for the nodes could be configured to accept traffic from the load balancer. When preserving client IP, the nodes’ security list or NSG rules need to be configured to allow traffic from the clients directly because the load balancer is not proxying the requests. [Table 4-1](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04.xhtml#ch04tab01) outlines an example security list rule.

**Table 4-1** Example Security List Rule

| **State** | **Source** | **Protocol/Destination Port** | **Description** |
| --- | --- | --- | --- |
| Stateful | 0.0.0.0/0 or subnet CIDR | ALL/30000-32767 | Allows worker nodes to receive connections through the OCI network load balancer with the source IP preserved |

If the client IPs are known (or are from an internal/known subnet), the security list rule or NSG rule can restrict the source to the known CIDR block for the source. If the service is exposed publicly, the source CIDR needs to be set to 0.0.0.0/0 for allowing traffic from anywhere.

[Listing 4-18](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04.xhtml#list4_18) shows these configurations in an example.

**Listing 4-18** Manifest Used to Create a Network Load Balancer

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04_images.xhtml#f143-01a)

... POD SPEC TRUNCATED ...

---

apiVersion: v1

kind: Service

metadata:

name: nginx-service

annotations:

oci.oraclecloud.com/load-balancer-type: "nlb"

oci-network-load-balancer.oraclecloud.com/is-preserve-source: "true"

spec:

type: LoadBalancer

externalTrafficPolicy: Local

selector:

app.kubernetes.io/name: proxy

ports:

- port: 80

protocol: UDP

targetPort: 80

This listing shows the nginx service example, as shown earlier; however, this time the configuration explicitly requests a network load balancer instead of the standard (Layer 7) load balancer by setting the annotation oci.oraclecloud.com/load-balancer-type: "nlb". By setting the protocol:UDP in the port configuration, OKE ensures that the listener created for the network load balancer is configured to accept UDP. Additionally, the configuration to preserve source IPs is enabled by providing the annotation oci-network-load-balancer.oraclecloud.com/is-preserve-source: "true" and setting the parameter externalTrafficPolicy: Local. It is assumed that the NSG rule or the security list rule to allow traffic from the source IPs to the nodes has been created.

**Specifying Reserved Public IP Addresses**

When you create a public load balancer or network load balancer with OKE, an ephemeral public IP address is assigned to that load balancer. In many circumstances, you might want to have a predefined and known IP address for your service. OCI allows you to reserve public IP addresses. When you create a public load balancer or network load balancer with OKE, you can choose to assign one of your reserved public IP addresses to that load balancer. To configure a public IP address for your load balancer, you need to specify the IP address field in your service definition, and your load balancer must be created in a public subnet. Suppose that 150.136.125.124 is one of your reserved IP addresses. To assign that IP to a load balancer created by the OKE, consider the example in [Listing 4-19](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04.xhtml#list4_19).

**Listing 4-19** Manifest Used to Create a LoadBalancer Service with a Reserved Public IP Address

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04_images.xhtml#f144-01a)

apiVersion: v1

kind: Service

metadata:

name: nginx-service

spec:

type: LoadBalancer

loadBalancerIP: 150.136.125.124

selector:

app.kubernetes.io/name: proxy

ports:

- name: http

protocol: TCP

port: 80

targetPort: http-web-svc

**Commonly Used Annotations**

The complete list of annotations is documented on the GitHub page for the Cloud Controller Manager; however, this section and the list herein covers the most common and frequently used annotations. Note the prefixes to the various annotations. Certain OCI-specific annotations and those that are common to both load balancers and network load balancers are prefixed with oci.oraclecloud.com. Load balancers use a mix of service.beta.kubernetes.io and oci.oraclecloud.com prefixes, while network load balancer–specific annotations are prefixed with oci-network-load-balancer.oraclecloud.com.

* oci.oraclecloud.com/load-balancer-type This annotation is used to switch between the type of load balancer. The possible values are lb for the OCI load balancer (Layer 7) or nlb for the OCI network load balancer (Layer 4).
* service.beta.kubernetes.io/oci-load-balancer-shape This template determines the load balancer’s capacity (bandwidth) for ingress and egress traffic. It should be set to flexible. Fixed-bandwidth shapes such as 100Mbps, 400Mbps, and 8000Mbps are now deprecated. This should be used in conjunction with service.beta.kubernetes.io/oci-load-balancer-shape-flex-min, which sets the minimum guaranteed bandwidth, and the optional service.beta.kubernetes.io/oci-load-balancer-shape-flex-max, which sets the load balancer’s maximum capacity that it will scale to.
* service.beta.kubernetes.io/oci-load-balancer-subnet1 This is the OCID of a subnet to attach the load balancer to when the default choice selected at cluster creation needs to be overridden for a specific service. If the subnet provided is regional, only a single subnet needs to be configured. When using AD-specific subnets, a value also needs to be provided for service.beta.kubernetes.io/oci-load-balancer-subnet2 to maintain high availability of the load balancer.
* service.beta.kubernetes.io/oci-load-balancer-health-check-timeout This is the maximum time, in milliseconds, to wait for a reply to a health check. A health check is successful only if a reply returns within this timeout period. By default, this is set to 3000, or 30 seconds. You should consider increasing this value for services that are backed by pods that can potentially take more time to start and respond to health check requests. You can also use this in conjunction with service.beta.kubernetes.io/oci-load-balancer-health-check-retries, which sets the number of times a failed health check is retried before the back-end server is marked as unhealthy, and service.beta.kubernetes.io/oci-load-balancer-health-check-interval, which lets you control the interval between health checks.
* service.beta.kubernetes.io/oci-load-balancer-security-list-management-mode This annotation determines how the CCM handles security list updates, to allow traffic among multiple components as services are exposed. This is a crucial setting in most cases, and is covered in detail in the next section. The permissible values for it are All (the default), Frontend, and None.
* oci.oraclecloud.com/oci-network-security-groups When using network security groups (NSGs) to manage and secure traffic flow, this annotation is used to designate the newly created load balancer for the given NSG. The NSG rules that you create and associate with this NSG dictate how traffic flow and security are handled.
* oci.oraclecloud.com/loadbalancer-policy This specifies the load balancer algorithm used to distribute traffic to the back-end servers. The default is to use ROUND\_ROBIN, which treats the back ends as a list, sends each incoming request to the next server in the list, and wraps around to the start of the list after each pass through the list. This type of distribution ensures that all back-end servers get relatively the same number of requests, but it does not account for when the request is a simple one that can be completed quickly or one that can be time-consuming. It also assumes that the back ends are all fairly similar in their capabilities for the load to be well balanced across all back ends. You can choose to use the LEAST\_CONNECTIONS or IP\_HASH algorithms. Use LEAST\_CONNECTIONS to have the service choose a back-end server that has the least active connections at that moment to route the request to. This ensures that there is no imbalance between servers that are handling long-lived requests and servers that are receiving smaller requests. The IP\_HASH algorithm uses a hashing function to calculate a hash value from the client’s IP address so that requests from the same IP address are always routed to the same back-end server. Although this offers the capability to achieve some level of stickiness, which could be important to some applications, it can also create an imbalance if a large number of clients connect to the service through a proxy. In this case, the proxy’s IP would be perceived as the source IP, and the hashing function would always send these requests to the same back-end server. If many clients are behind a proxy, that can cause the load to not be well balanced and can potentially overburden an individual back-end server.

**Understanding Security List Management Modes**

The annotation service.beta.kubernetes.io/oci-load-balancer-security-list-management-mode controls how the CCM manages security lists for the load balancer subnet and the node subnet. Appropriate ingress and egress rules on these subnets are required for traffic flow because the default behavior for subnets is to disallow all traffic. The annotation can have the following values:

* All**:** When the value is set to All, the CCM manages security lists for both the node subnets and the load balancer subnets. When a pod is started on a node and is exposing a set of container ports, these ports are within the network namespace for the pod. Ports are exposed on the host as well to route traffic from the host namespace to the pod namespace. From a VCN networking perspective that sees the hosts and not the pods or containers running on them, these ports on the host need to have security list rules that allow traffic to them. The CCM looks up the subnets for each of the nodes where the pods are running because each node pool can potentially be located on a different subnet; then the CCM adds security list rules that allow these specific host ports to receive traffic from the load balancer subnet.

Similarly, the CCM updates the security list rules for the load balancer subnet to send traffic to these ports on the node subnet. Because the load balancer is exposing the application and listening on the configured load balancer port, the CCM also adds a security list rule to allow incoming traffic to the load balancer port.

In this mode, because the CCM is updating resources such as the security list, it is possible for infrastructure management tools that track resource state, such as Terraform, to flag these modifications as configuration drift.

* Frontend**:** When the value is set to Frontend, the CCM manages security lists for only the front end, or the ingress for the load balancer subnet. Here, it is assumed that you have already configured security rules that open the node port range for your node subnets to the load balancer subnet, with egress rules on the load balancer subnet and ingress rules on the node subnets.
* None**:** In this mode, the CCM does not do any security list rule management. You need to configure these rules externally, such as with Terraform, using well-known ports on the load balancer ingress side and using the node port range between the load balancer and node subnets.

**Using Node Label Selectors**

As discussed previously in the chapter, the node label selector annotation in OKE is used to organize a subset of nodes in your cluster. In the context of exposing your application using a service, these label selectors enable you to define a subset of nodes to act as the back end for your service. By default, when you deploy a set of pods and expose it using a service, the pods can be located in any node within your cluster, based on how Kubernetes schedules them. In some cases, however, you might want to exert control over what nodes are chosen to be in the back-end set for a service. This can be useful in scenarios such as node upgrades, during which you want to control when traffic moves to a new set of nodes without impacting your existing workloads.

The key idea with using node label selectors is that you label your nodes first; then you can update your service to include a node label selector, which causes the CCM to update the load balancer and include only nodes that match the label selector in the back-end set for the load balancer listener. Consider the example of updating a set of nodes and wanting to update the service to switch over to the new fleet without impacting the workloads. The example here starts off with the existing service and labeled nodes. For example, consider the following command:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04_images.xhtml#f147-01a)

kubectl label nodes 10.0.1.2 10.0.1.2 10.0.1.3 app-tier=frontend

This gives three nodes the label app-tier=frontend. With the nodes labeled, you can deploy a workload that is pinned to these nodes with a definition like the one in [Listing 4-20](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04.xhtml#list4_20) to ensure that the pods are all created on nodes with the specified node selector.

**Listing 4-20** Manifest Used to Define a Deployment That Ensures All Pods It Creates Are Scheduled onto Nodes with the Frontend Selector

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04_images.xhtml#f147-02a)

apiVersion: apps/v1

kind: Deployment

metadata:

name: frontend-deployment

labels:

app: frontend

spec:

replicas: 3

selector:

matchLabels:

app: frontend

template:

metadata:

name: frontend

labels:

app: frontend

spec:

affinity:

nodeAffinity:

requiredDuringSchedulingIgnoredDuringExecution:

nodeSelectorTerms:

- matchExpressions:

- key: app-tier

operator: In

values:

- frontend

containers:

- name: frontend

image: frontend:1.0.0

ports:

- containerPort: 80

name: http-web-svc

**Note**

In this example, it is assumed that there are several nodes in the cluster, some of which do not match the label selector used here.

The nodeAffinity for this deployment restricts the pods running the frontend:1.0.0 container image to the three nodes that were labeled with app-tier=frontend. A service definition can now use the node label selector annotation to select the specific nodes to be added to the load balancer back-end set, as shown in [Listing 4-21](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04.xhtml#list4_21).

**Listing 4-21** Manifest Used to Define a Service with a Back-End Set That Includes Only Nodes Created with the Frontend Selector

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04_images.xhtml#f148-01a)

apiVersion: v1

kind: Service

metadata:

name: frontend-service

annotations:

oci.oraclecloud.com/node-label-selector: app-tier=frontend

spec:

type: LoadBalancer

selector:

app: frontend

ports:

- name: http

protocol: TCP

port: 80

targetPort: http-web-svc

The annotation ensures that only nodes that carry the label app-tier=frontend will be included in the back-end set for the load balancer’s listener. Label selectors can be in several formats, including exclusions such as app-tier!=database, which selects all that have the key app-tier but whose value is not database.

**Note**

Kubernetes supports a feature gate named ServiceNodeExclusion to label nodes that should be excluded from a load balancer. OKE enables this feature gate by default. This means that you can label your nodes with node.kubernetes.io/exclude-from-external-load-balancers to keep the node from being added to the back-end set of a service.

Now imagine that you want to cycle the nodes that are running the front-end application, and you create three new nodes. You can label them as app-tier=frontend-v2, as shown here:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04_images.xhtml#f149-01a)

kubectl label nodes 10.0.1.4 10.0.1.5 10.0.1.6 app-tier=frontend-v2

The deployment can now be updated to include the new nodes labeled app-tier=frontend-v2, and the service definition can be updated to include both the new nodes and the old nodes:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04_images.xhtml#f149-02a)

oci.oraclecloud.com/node-label-selector: app-tier=frontend,app-tier=frontend-v2

When you have ensured that the pods are available on the new node pool, you can drop the old node pool from the back-end set for the load balancer by setting the annotation as follows:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04_images.xhtml#f149-03a)

oci.oraclecloud.com/node-label-selector: app-tier=frontend-v2

The old nodes can now be cordoned and drained without impacting traffic because the load balancer directs traffic only to the new nodes.

**Security Considerations for Your Cluster**

As you deploy your applications into a Kubernetes cluster and operate it, security for your cluster and your application becomes important. Unauthorized access can potentially cause application outages that impact business. Worse, it can lead to data breaches that have long-term impacts. This section helps you understand the various facets of a cluster and its infrastructure that needs to be secured, along with ways to achieve it. A wider discussion of securing both your applications and your infrastructure, understanding attack vectors, considering the cloud native security ecosystem, and developing a system’s overall security posture is covered in [Chapter 6](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch06.xhtml#ch06), “[Cloud Native Security](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch06.xhtml#ch06).”

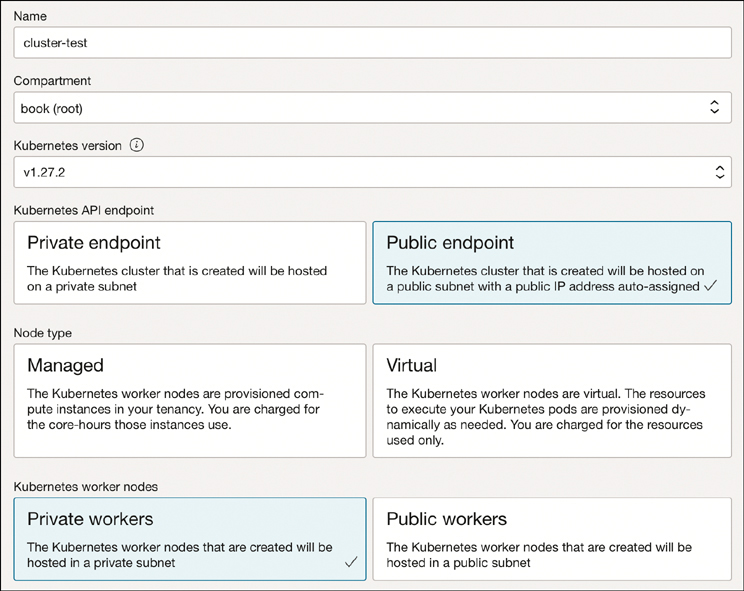
Security for your cluster can broadly be categorized into securing the runtime infrastructure and securing access to the cluster. Securing the runtime infrastructure refers to how security principles are applied to the cluster infrastructure topology and its configuration. Setting up secure access to the cluster considers the controls and configuration for the access paths, authentication mechanisms, and authorization mechanisms to ensure that users can be provided with only the necessary capabilities. On the other hand, securing access to the cluster and establishing good practices for authorization prevents attack vectors that originate from within an organization. Observability through metrics, log analytics, and auditing is also a key component in having a good security strategy.

**Cluster Topology and Configuration Security Considerations**

The benefit of having a well-thought-out strategy for infrastructure security is the ability to prevent intrusions into your infrastructure components and data—in other words, the capability to shield yourself from attack vectors that originate from the outside. The following sections look at several considerations from an infrastructure perspective.

**Cluster Component Visibility**

When you create an OKE cluster, one of your first choices is to select the subnets and visibility for the Kubernetes API endpoint and the worker nodes. When using the Quick Create workflow, you are asked to choose the visibility of your Kubernetes API endpoint and your worker nodes. You can opt to make either component public or private. When you use the Custom Create workflow, you are asked to choose the subnets where you want to place your Kubernetes API endpoint, your load balancers, and your worker nodes. If you choose a public subnet, you can assign public IP addresses for these elements; on the other hand, if you choose a private subnet, these elements remain private. [Figure 4-13](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04.xhtml#ch04fig13) shows the choices in the Quick Create workflow.



**Figure 4-13** Choosing Between Private and Public Endpoints for the Kubernetes API and the Kubernetes Worker Node Subnets

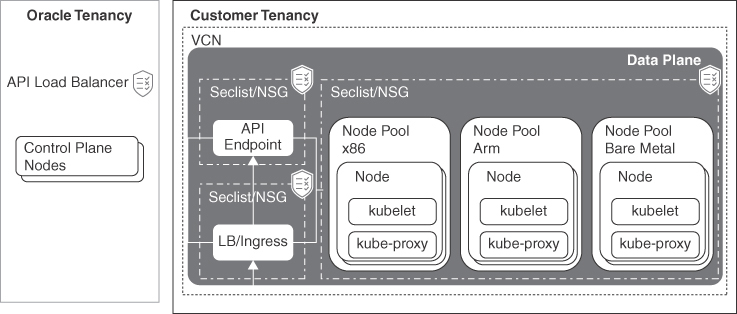
When you expose the Kubernetes endpoint to be public, your Kubernetes endpoint is available via a public IP. This means that, from a visibility or network reachability standpoint, it is open to the Internet for anyone to realize that a Kubernetes API server exists at this location. Access to the API server’s resources is still controlled by the user’s authorization credentials, so an unauthorized user will be refused service by the API server itself. A public API endpoint is desirable for ease of use because it allows you to connect directly to the API server over the Internet. On the other hand, for most production applications, a private API endpoint can be considered a better and more secure choice because it limits the visibility and reachability of the API server to locations within the virtual cloud network where your cluster is located. To access the API server and interact with it using a client such as kubectl, users typically need a bastion host (also sometimes called a jump host) from where they can connect to the API server.

Following similar reasoning, the visibility of the worker nodes should also be limited to be private so that they can be reached only from within the version cloud network. This means that, to access the nodes directly or log into them using SSH, they would need to be within the virtual cloud network, such as on a bastion server. Having private worker nodes does not mean that you cannot expose your workloads externally. A service exposed as a load balancer will cause an OCI load balancer to be created and wired to the pods, and the communication between the load balancer and the private nodes running the pods will be traffic that is private to the virtual cloud network. This will work as long as the load balancer and the worker node subnets are configured to allow communication between each other. If you are using the Quick Create workflow, these security list rules are set up by the service; if you are using the Custom Create workflow, the required rules are documented in the official documentation for a static configuration. It is also worthwhile to note that the OKE CCM (Cloud Controller Manager) is set up by default to update the security lists for the subnets when services are created and destroyed. This ensures that only the required ports are opened, and only while they are needed.

**Setting Up NSG Rules and Security List Rules**

When you set up an OKE cluster using the Quick Create workflow, the service creates the networking components using a simple dedicated network and configures the components appropriately. In a production scenario, however, you will likely have a different and more complex network topology that is shared with other resources and is perhaps managed by a dedicated team. When you are reusing existing network components, it is important to configure them in a manner that allows the OKE components, especially the control plane and data plane, to effectively communicate with each other. You can perform these configurations using security list rules, NSG rules, or both. Recall that security lists are attached to subnets, and the rules take effect at a subnet level. NSG rules, on the other hand, are applied at a VNIC level. If you use both approaches to control network traffic security, the effect is additive: The effective rule should be the union of the NSG rules and the security list rules. If either of them allows communication between two components, the traffic is allowed.

As a general rule, it is recommended to separately manage the ingress and egress rules for the Kubernetes API endpoint, the load balancers, and the worker nodes. This allows for maximum flexibility without compromising security. When using NSGs, you can have all these elements in the same subnet but still treat the ingress and egress separately by creating NSG rules because NSG rules are applied at a VNIC level. If you are using security list rules, it is important to place these components in separate subnets so that their ingress and egress can be controlled appropriately using security list rules. Consider the diagram in [Figure 4-14](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04.xhtml#ch04fig14).



**Figure 4-14** Architecture of Network Security Groups and Security Lists

The dashed boundary lines represent security lists or NSGs. Security lists contain rules that govern egress and ingress traffic for the entirety of the subnet. NSGs contain rules that control ingress and egress for traffic on the VNIC for each component, such as the API server endpoint, load balancers, or nodes. In this regard, NSGs offer more fine-grained control at the VNIC level for securing traffic. Regardless of the mechanism used to control ingress and egress traffic, the general recommendation is to use either separate subnets with their own security lists or separate NSGs to secure traffic to the various OKE components. This is because, in most cases, you will need to secure each component differently. For example, you will want to ensure that your public load balancer allows incoming traffic from the Internet, but you wouldn’t want to allow external traffic from the Internet to reach your nodes. Having these components in separate subnets or NSGs will make it easy to implement traffic security to meet your needs.

When choosing to place the components in separate subnets, the Kubernetes API endpoint subnet can be small, with room for just three IP addresses. The size of the load balancer subnet depends on how many load balancers you will have within this cluster; this number is generally low, so this subnet can also be relatively small, similar to the Kubernetes APU endpoint subnet. The node subnet, on the other hand, can potentially be large, in case you need to support a fairly large number of nodes within your cluster. You can also have node pools in separate subnets to implement use cases for when you might want isolation at a fundamental level between workloads.

When considering how to set up the ingress and egress rules between these components, the fundamental rules to remember are those outlined in [Figure 4-14](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04.xhtml#ch04fig14). They are listed here in a brief form. The complete network ingress and egress rules are detailed in the official documentation.[**7**](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04.xhtml#ref4_7a)

* The API endpoint should be able to egress to the control plane on port 443, usually over a service gateway.
* The API endpoint should expose 6443 to allow clients to access it. This can be restricted to a specific CIDR or opened to all sources.
* The API endpoint needs to communicate with the worker nodes.
  + TCP ports 6443 and 12250 should be reachable from the nodes or the pod network (when using native pod networking).
  + ICMP 3,4 should be open for ingress and egress between the API endpoint and worker nodes, for path discovery.
* The worker nodes should be able to communicate with the other worker nodes and the pod network (when using native pod networking).
* The worker nodes should also be able to communicate with the control plane and the API endpoint.
* The load balancer should be able to communicate with the worker nodes on the node port range. This rule can be added by the service at runtime when services are annotated to have OKE manage the security list rules on behalf of the application.

**Using Compartments to Control Access**

OCI compartments can be used effectively to control access to resources. In the case of OKE, you can use compartments to set up various components in separate compartments, to gain fine-grained control over how these resources are accessed and used. For example, you might want to create your network configuration, such as your VCN, subnets, and more, in a specific compartment—for example, a network to which the network engineers have full access but developers have only read access. You might also want to create your nodes in a separate compartment, such as Kubernetes, to which your developers have full access and network engineers have only read access. You can take this concept up a notch by placing each of your node pools into separate compartments. This type of a configuration is particularly attractive for users who run multitenant Software as a Service (SaaS) applications. You can now isolate the workloads of one tenant into a specific node pool and compartment, and create an IAM policy that restricts access to that given compartment—and, thereby, the node pool. You can ensure that pod scheduling respects these boundaries using node taints and tolerations. OCI also offers guidance and automation on bootstrapping environments with predefined and secure isolation models, such as the CIS OCI Landing Zone, which is based on the independent Center for Internet Security (CIS) Foundations Benchmark v1.2 for OCI.

**Creating Groups with Limited Access**

When starting out with a new service, it is common to rely on superuser privileges. This practice usually gets quicker results because it involves less friction in terms of security controls. However, when teams move from running proof-of-concepts into running production applications, thought should be given to how access control is structured, implemented, and enforced. For instance, consider an organization in which infrastructure management is carried out by a dedicated team and application development is carried out by a development team. When planning access, consider these organizational standards and subdivisions within teams. For example, infrastructure engineers focused on networking require different identity and access than infrastructure engineers focused on image hardening and OS security. Similarly, developers working on the front end of your application don’t need the same level of privileges as developers managing your database.

OCI IAM and its policy-driven access model offer several approaches and a flexible way to implement access control, depending on your organizational needs and structure. Access is always denied by default, and policies have an additive effect, making them simple to craft and audit through OCI audit logs and Cloud Guard.

**Enabling Image Signature Verification**

OKE clusters support image signature verification, which ensures that containers created on the cluster are created from images that have not been tampered with and come unmodified from an authentic source. Images in OCIR can be signed with a master encryption key that you manage. When an image is signed, the signature associates the image to the encryption key used to sign it. You can configure the same encryption key on OKE to verify signatures; during signature verification, OKE uses the key to verify the signature. The signature takes into account the contents of the image and when any of the bits in the image have been manipulated. For example, in a man-in-the-middle attack, signature verification would fail and OKE would refuse to pull the image from OCIR. Image verification includes an additional step of signing the images that you push, but this is a quick and simple operation that can easily be integrated into a build pipeline. Consider the example in [Listing 4-22](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04.xhtml#list4_22).

**Listing 4-22** Example of Using the OCI CLI to Sign and Upload a Container Image to the OCI Registry Service

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04_images.xhtml#f155-01a)

oci artifacts container image-signature sign-upload \

--compartment-id COMPARTMENT\_ID \

--kms-key-id KEY\_OCID \

--kms-key-version-id KEY\_VERSION\_ID \

--signing-algorithm ALGORITHM \ # eg : SHA\_224\_RSA\_PKCS\_PSS

--image-id IMAGE\_OCID \

--description "Image Signing"

This command signs the given image identified by IMAGE\_OCID with a key and a key version identified by KEY\_OCID and KEY\_VERSION\_OCID, using the specified algorithm. The algorithm supported by the key is chosen at the time the key is created, and not all algorithms can support a signature. For example, AES keys are symmetric keys that do not support signatures, whereas RSA and ECDSA keys are asymmetric keys that do support it.

With image signature verification enabled, every image that is pulled is then verified for its signature validity. Within your pod spec, you should refer to specific images using their digest value, such as image\_name@sha256:xxxxx, instead of a transient tag, such as image\_name:latest or image\_name:1.0.0. In some emergencies, it might be necessary to “break glass” to deploy an image that would normally fail the signature verification onto a cluster that has the feature enabled. To do this, you use the following policy on the pod spec:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04_images.xhtml#f155-02a)

oracle.image-policy.k8s.io/break-glass: "true"

This allows the cluster to bypass the image verification for this container alone. Needless to say, it is a best practice to have linting enabled for your code repositories and to configure policy check alerts on the audit logs so that the “break glass” function is used only in emergencies and so that alarms are raised when it is.

**Encrypting Kubernetes Secrets**

A *secret* is an object in Kubernetes that is typically used to store and distribute sensitive data such as passwords, tokens, or configuration information. Secrets are similar to ConfigMaps in how you use them, and they decouple the sensitive information from the application or the container image. Although secrets decouple the application from having to bake sensitive data into the application or the container image, you need to take an informed approach in how your cluster is configured to ensure that secrets are handled and managed safely.

Secrets, like ConfigMaps and most other data that Kubernetes stores, are stored in the *etcd* datastore. etcd runs on the control plane. When you run your own control plane, by default, the data in etcd is stored unencrypted. Anyone with the appropriate API permissions or direct access to etcd can thus access and modify secrets.

When you are using OKE, the control plane is managed by Oracle, and you do not have direct access to it. OKE also manages the etcd datastore for you, and it uses the OCI Block Volume Storage service to persist etcd data. These block volumes are always encrypted, by default, so your secrets, ConfigMaps, and all other Kubernetes objects that are stored in etcd are encrypted at rest. Oracle manages and periodically rotates these encryption keys without any action from you, and this default encryption is always on.

In some scenarios, you might want to bring your own encryption key to encrypt the data in etcd and manage the lifecycle and key rotations yourself. This might be required for compliance reasons or simply because you want to ensure that the data is encrypted with a key that you manage and that Oracle does not have access to. OKE supports this model as well: You can choose to bring your existing keys and store them in the OCI Vault, where you can use them to encrypt the secrets in etcd. This option is available only while creating a cluster, using the Custom Create workflow, and using the APIs and Terraform provider. When you create the cluster, you can choose a Vault and a master encryption key that you have imported to the Vault or generated by yourself. In either case, Oracle does not have access to the key, and you can manage this key.

Note that, when you manage your own keys, if you delete the keys, the secrets encrypted by the key become inaccessible. There is no way to recover the secrets at this point.

When you use your own keys to encrypt secrets in etcd, you might also want to implement key rotation to periodically update the key. When you rotate keys, the key’s OCID does not change, but it will have a new value and be considered a new version of the key. Any Kubernetes secret created from that point onward will be encrypted using the new version of the key. Existing secrets will remain encrypted with the old version of the key. This does not introduce any problems, but you have the choice to re-encrypt all the existing secrets as well. For instance, if you suspect that the old key has been compromised, you might want to re-encrypt the existing secrets with the new key. You can re-encrypt the existing secrets in your cluster with the new key using a simple one-line script:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04_images.xhtml#f156-01a)

kubectl get secrets --all-namespaces -o json | kubectl annotate --overwrite -f

- encryption-key-rotation-time="<time>"

**Authorization Using Workload Identity and Instance Principls**

When your workload needs to access an OCI API such as read or write from an object storage bucket, you need a way to provide access and permissions to allow this. This can be achieved using either workload identity or instance principals. These two mechanisms achieve similar goals, but have differences that make them appropriate for specific scenarios. It is important to understand the security implications of choosing one method over the other so that you choose the right approach for any given situation.

**Using Instance Principals**

Instance principals are a powerful feature in OCI that confers identity and permissions to any compute instance in OCI, including OKE worker nodes. When a compute instance that has been given certain privileges is running a workload, the workload is allowed to interact with the OCI APIs using the credentials and permissions allowed for that instance. In the case of OKE, any of the pods running on such a node can call OCI APIs. Although the scheduler can be influenced to ensure that only selective workloads are running on a node, using this feature with OKE leads to a less than ideal governance model. If a running pod has a vulnerability that lets a malicious user gain access to a workload, the attacker can then leverage the instance principals to perform deeper actions and cause further damage. To use instance principals, you need to create a *dynamic group* that selects a set of compute instances based on *matching criteria*. After the dynamic group is created, you can create an OCI IAM policy that grants permissions to this dynamic group. Consider the following example that defines a dynamic group with a matching criterion:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04_images.xhtml#f157-01a)

All {instance.compartment.id='<COMPARTMENT\_ID>', tag.projectA.env.value='prod'}

This dynamic group selects all instances in the compartment with the ID COMPARTMENT\_ID and further narrows the instances selected using a tag. Assume that there is a tag namespace of projectA and a tag key named env. The matching rule in the previous example selects all instances that have this tag with the value set to prod. It can be assumed that all nodes that belong to projectA’s prod environment may carry this tag. Therefore, the dynamic group would select all of projectA’s instances in a given compartment with the projectA.env tag set to prod.

With the dynamic group created, all that remains to be done is to create an IAM policy that gives this dynamic group access to the required OCI resources. Consider the following example:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04_images.xhtml#f157-02a)

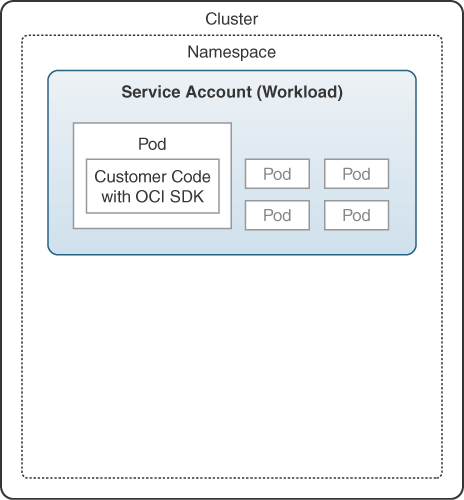
Allow dynamic-group projectA-prod to use buckets in compartment ProjectA

This policy allows members of the dynamic group named projectA-prod, which are the instances selected by the matching criteria of the dynamic group, to use object storage buckets that are in the compartment called projectA.

This approach is valid when all the workloads or applications that may run on these instances are trusted. For Kubernetes nodes, this might not be a good general-purpose solution because the applications that are running on this node are not entirely predictable. Take care to ensure that untrusted workloads or applications are not started on nodes because this could abuse the privileges conferred on the nodes. Of course, you can control node scheduling with features such as node affinity, taints and tolerations, or custom schedulers; however, those use cases are generally the exception, not the norm.

**Using Workload Identity**

For most workloads that need to access OCI APIs, such as to read/write to an object storage bucket, as in the previous example, the preferred approach is to implement workload identity. Workload identity is a feature that enables you to provide access and permissions to your workloads running as pods on OKE. Pods with a specific service account, in a specific cluster, and in a specific Kubernetes namespace are selected and can be given permissions to access OCI APIs. OCI IAM defines a workload resource with this combination of cluster, namespace, and service account. You can write policy statements using these variables to allow a workload access to OCI resources. Applications can then use the OCI SDK and the new OKE authentication provider that authenticates the API calls simply based on the workload’s identity, which is defined by service account, namespace, and cluster. [Figure 4-15](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04.xhtml#ch04fig15) illustrates the different components of the workload identity architecture, including adding the OCI SDK to customer code that is then deployed to the cluster as a pod.



**Figure 4-15** Workload Identity Architecture Showing Where a Customer Uses the OCI SDK to Enable Workloads to Authenticate to OCI IAM

This approach considerably narrows the access and ensures that principles of least access are upheld. If Kubernetes moves the pod around multiple nodes in the cluster, access is uninterrupted because the identity rests with the workload (identified by service account, namespace, and cluster), not the instance where the pod runs. This also enables you to control what applications have access to what OCI APIs, unlike with instance principals, which are inherited by any workload that is running on the given instance. Using OCI IAM policies to manage access also ensures a consistent way of managing and auditing access throughout OCI. API requests made using the workload identity are tracked in OCI audit and can be used to satisfy compliance requirements or to monitor for access violations and suspicious activity.

To use workload identity, you need the typical Kubernetes resources, such as namespaces and service accounts.

Create a namespace for your application if one already does not exist. This creates a boundary for your application, and you deploy your pods and other resources within this namespace. Consider the following example that creates a namespace called my-app:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04_images.xhtml#f159-01a)

kubectl create namespace my-app

Create a service account that the pods of the application can use. Pods that are part of your workload will use this service account to run. These service accounts can also be used to enable Kubernetes RBAC for the pod. Consider the following example, which creates a service account named my-app-sa in the namespace my-app:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04_images.xhtml#f159-02a)

kubectl create serviceaccount my-app-sa \

--namespace my-app

The application pods deployed to the my-app namespace should now use this my-app-sa service account when running. This can be set in the pod spec, as demonstrated in [Listing 4-23](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04.xhtml#list4_23).

**Listing 4-23** Manifest Used to Create a Pod That Uses the my-app-sa Service Account

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04_images.xhtml#f159-03a)

apiVersion: v1

kind: Pod

metadata:

name: my-application

spec:

serviceAccountName: my-app-sa

automountServiceAccountToken: true

containers:

- name: my-application

image: my-application:1.0.0

ports:

- containerPort: 8080

Define an IAM policy to grant the workload resource access to other OCI resources. As with other IAM policies, the policy defines what permissions are granted to the principal. Consider the example in [Listing 4-24](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04.xhtml#list4_24), which shows a policy that checks the incoming request to ensure that it meets the criteria for the workload identity, the combination of service account, namespace, and cluster ID. Processes that are running in pods with the service account my-app-sa, that are running within the my-app namespace, and that are in a cluster with a specified CLUSTER\_OCID are allowed to use object storage buckets in the compartment named ProjectA.

**Listing 4-24** Example OCI IAM Policy Used to Grant the Target Workload Access to OCI Resources

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04_images.xhtml#f160-01a)

Allow any-user to use buckets in compartment ProjectA where all {

request.principal.type = 'workload',

request.principal.namespace = 'my-app',

request.principal.service\_account = 'my-app-sa',

request.principal.cluster\_id = 'CLUSTER\_OCID'}

With the service account and the IAM policy in place, for the workload to access OCI (for example, to use the object storage APIs as indicated in the policy), it needs to use the OKE workload identity provider. Consider the example in [Listing 4-25](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04.xhtml#list4_25) using the OCI Java SDK.

**Listing 4-25** Example of Using the OCI Java SDK to Enable a Workload to Access OCI IAM

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04_images.xhtml#f160-02a)

[...]

final OkeWorkloadIdentityAuthenticationDetailsProvider provider = new OkeWorkloadIdentityAuthenticationDetailsProvider

.OkeWorkloadIdentityAuthenticationDetailsProviderBuilder()

.build();

/\* create the client using the workload identity provider \*/

ObjectStorage osClient =

ObjectStorageClient.builder().region(Region.US\_PHOENIX\_1).

build(provider);

[...]

At runtime, when the API call is made, the OKE authentication provider validates the workload identity (service account, namespace, and cluster) and issues a short-lived resource token from the OCI identity service. For most workloads, this approach offers a secure method of accessing OCI APIs without having to grant access too widely or having to manage credentials as secrets that need to be rotated and managed.

**Securing Access to the Cluster**

Clients interact with Kubernetes through the APIs exposed by the API server. This includes interactions for basic actions such as creating a pod, or more complex ones such as deploying an operator. When a client interacts with the API server, the client is authenticated using the credentials that it provides. On OKE, the default method for a client such as kubectl to authenticate with the API server is to use a short-lived token.

Once authenticated, the client credentials are subjected to authorization checks using role-based access control (RBAC). OKE clusters have RBAC enabled by default. RBAC in Kubernetes lets you implement fine-grained access control using the standard Kubernetes roles and RoleBindings. Roles and RoleBindings are bound to namespaces, while ClusterRoles and ClusterRoleBindings are valid across the cluster. A role represents a set of permissions that allow actions on a set of resources and API groups—for example, allowing the get and list verbs on the pods resource in the core API group. A set of default Kubernetes ClusterRole and ClusterRoleBinding objects, such as the cluster-admin ClusterRole, is created along with the cluster. Many of the default ClusterRoles and ClusterRoleBindings are prefixed with system:, indicating that these are created and managed by the cluster control plane.

**OCI IAM and Kubernetes RBAC**

OKE comes preconfigured with an authorizer that integrates with OCI IAM. For example, OKE considers anyone in the OCI IAM Administrators group to be a Kubernetes cluster-admin as well. However, you can—and should—create more fine-grained access policies using Roles and RoleBindings as a best practice. This way, you can maintain and manage appropriate access while maintaining principles of least access. Because the OKE Authorizer integrates Kubernetes RBAC with OCI IAM, it becomes easy to create and manage RoleBindings that associate OCI IAM groups and users to Kubernetes roles.

As an example, consider two users, Bob and Bill. Bob is a member of the Administrators group in OCI IAM, and Bill is a member of the Developers group in OCI IAM. Bill wants to get full privileges to create and manage Kubernetes resources, but only in the ProjectA namespace. By default, Bob has full access to the cluster, whereas Bill has no access. The first step that you might consider for giving Bill access to the cluster would be to add the group Developers to an IAM policy that lets the group members use OKE clusters:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04_images.xhtml#f161-01a)

Allow group Developers to use clusters in compartment <compartment\_name>

This grants the members of the Developers group, including Bill, the CLUSTER\_USE permission. Next, you can create a RoleBinding in the ProjectA namespace that binds the cluster-admin ClusterRole to the Developers group. This can be done with a RoleBinding, as demonstrated in [Listing 4-26](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04.xhtml#list4_26).

**Listing 4-26** Example Manifest Used to Create a Cluster Role Binding That Maps an OCI IAM Group to a Kubernetes Role

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04_images.xhtml#f161-02a)

apiVersion: rbac.authorization.k8s.io/v1

kind: RoleBinding

metadata:

name: namespace-admin

namespace: projectA

subjects:

- kind: Group

name: [OCID\_for\_the\_Developers\_IAM\_group]

apiGroup: rbac.authorization.k8s.io

roleRef:

kind: ClusterRole

name: cluster-admin

apiGroup: rbac.authorization.k8s.io

By creating this RoleBinding, you have associated or granted the cluster-admin ClusterRole to the Developers IAM group by providing the group’s OCID. Because you created a RoleBinding and not a ClusterRoleBinding, it grants the cluster-admin privileges to the Developers group only in the ProjectA namespace, not across the cluster. With the RoleBinding in place, and because Bill is a member of the Developers group, when he tries to perform an operation such as creating a pod, he is able to do so because the authorizer will allow this request based on his IAM group memberships. Similar to the example here, you can also create a ClusterRoleBinding instead of a RoleBinding, and you can replace the subjects.kind: Group with a subjects.kind: User to bind the role to a single IAM user instead of a group.

**Federation with an IDP**

The integration between OCI IAM and Kubernetes RBAC can extend to federated identities and groups as well. OCI IAM supports federation with identity providers that adhere to the Security Assertion Markup Language (SAML) 2.0 protocol. Federation enables enterprises to integrate their existing identity provider (IdP) with OCI. Once configured, users can log into OCI and use their existing enterprise usernames and passwords through their familiar single sign-on (SSO) login page. When administrators set up federation in OCI IAM with an IdP, they map the existing groups from the IdP to OCI IAM groups. When creating RoleBinding and ClusterRoleBindings, you can continue to use the OCID for the OCI IAM group to which the IdP group has been mapped. When changes to group memberships occur on the IdP—for example, Bob moved from the Developers group to the SiteReliability group—these changes are synchronized to the mapped groups as well and will correctly reflect the access given to the users by virtue of their group memberships.

**Summary**

This chapter introduced Container Engine for Kubernetes (OKE). You looked at the basic terminology for the service, including what constitutes the control plane and data plane, and you examined ways to create a new cluster and automate that process. This chapter also introduced the various Kubernetes networking models that are available for use with OKE and storage features that are integrated with various OCI storage services through the Kubernetes persistent volume mechanism. This was followed by a discussion on load balancer support and various configurations to support a wide array of workload types. Additionally, you examined the cluster topology configurations and security considerations, including setting up security rules for various methods of access control. Finally, the chapter discussed ways to secure access to the cluster, including integrating Kubernetes RBAC with OCI IAM and configuring identity federation with an external identity provider (IdP).

**References**

[1](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04.xhtml#ref4_1) All Image Families: <https://docs.oracle.com/en-us/iaas/images/>

[2](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04.xhtml#ref4_2) oci\_containerengine\_cluster: <https://registry.terraform.io/providers/oracle/oci/latest/docs/resources/containerengine_cluster>

[3](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04.xhtml#ref4_3) oci\_containerengine\_node\_pool: <https://registry.terraform.io/providers/oracle/oci/latest/docs/resources/containerengine_node_pool>

[4](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04.xhtml#ref4_4) Data source: oci\_containerengine\_clusters: <https://registry.terraform.io/providers/oracle/oci/latest/docs/data-sources/containerengine_clusters>

[5](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04.xhtml#ref4_5) Terraform OKE for Oracle Cloud Infrastructure: <https://registry.terraform.io/modules/oracle-terraform-modules/oke/oci/latest>

[6](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04.xhtml#ref4_6) Load balancer annotations: <https://github.com/oracle/oci-cloud-controller-manager/blob/master/docs/load-balancer-annotations.md>

[7](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04.xhtml#ref4_7) Security Rule Configuration in Network Security Groups and/or Security Lists: <https://docs.oracle.com/en-us/iaas/Content/ContEng/Concepts/contengnetworkconfig.htm#securitylistconfig>