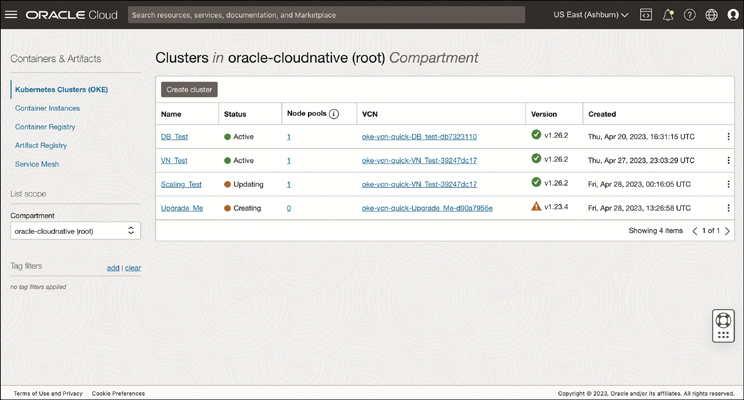
**5**

**Container Engine for Kubernetes in Practice**

Container Engine for Kubernetes in OCI enables you to quickly get started with Kubernetes. However, managing a fleet of clusters or a large cluster with several node pools and multitenant workloads running on it can be a bigger challenge than getting started. Owning a cluster also involves planning for and executing routine activities such as patching and upgrading. The burden imposed by these activities also varies by the node pool type. For instance, when using a managed node pool, you have direct control over the underlying node operating system and are responsible for keeping the operating system up to date. On the other hand, when using virtual nodes, the infrastructure is fully managed by Oracle; the user simply triggers an upgrade process at a desired time. Your runbooks for managing cluster components therefore will vary, depending on the specific configuration of your cluster. You might also need to leverage third-party software and use custom configurations to go beyond the defaults provided by the service. This chapter examines the processes and best practices for managing your clusters, integrating with third-party products and services, and using customized infrastructure configurations for your cluster components.

You can view the clusters created in your tenancy by opening the navigation menu in the console and clicking **Developer Services**. Under **Containers & Artifacts**, click **Kubernetes Clusters (OKE)**. Then choose a compartment. You will see the names, statuses, number of node pools, VCN, Kubernetes version, and creation date of all clusters in that compartment and region, as illustrated in [Figure 5-1](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#ch05fig01). This overview page is useful for keeping track of the high-level status of your clusters. Clusters running Kubernetes versions earlier than the latest available version display a warning indicating that upgrading to the latest version is recommended. From this page, you can jump directly to the Cluster Details page of a specific cluster by clicking the name of the cluster in the **Name** column. Similarly, you can jump directly to the node pool and VCN pages by clicking on the name in the **Node Pool** and **VCN** columns, respectively.



**Figure 5-1** Clusters of Varying Statuses and Kubernetes Versions on the Clusters Page of the OCI Console

**Kubernetes Version Support**

The upstream Kubernetes project is constantly evolving. Changes made to the project are referred to as *enhancements*. The Kubernetes project loosely defines enhancements as a change that a blog would be written about, that needs significant effort or changes Kubernetes in a significant way, or that users will notice or come to rely on. Enhancements are introduced in new minor versions that are typically released every four months. Patches to address bugs are released monthly or, if needed, more frequently, and are cut from the most recent three minor release branches maintained by the upstream Kubernetes project. Kubernetes versions are expressed as *x.y.z*, where *x* is the major version, *y* is the minor version, and *z* is the patch version.

Oracle Container Engine for Kubernetes (OKE) generally supports three versions of Kubernetes at a given time. For a minimum of 30 days after the announcement of support for a new Kubernetes version, OKE continues to support the oldest of the three Kubernetes versions that were previously available. After that time, the oldest Kubernetes version ceases to be supported. When creating a new OKE cluster, it is recommended that you use the most recent Kubernetes version available. When OKE supports a new Kubernetes version, it is recommended that you upgrade existing clusters to use that new Kubernetes version as soon as possible. OKE will not forcibly upgrade clusters if they become unsupported. If your clusters are running an unsupported version, you will be able to upgrade both the control plane and the data plane through unsupported versions to reach a supported one.

Because Kubernetes includes a control plane and a data plane, upgrading a cluster must be done in two parts. The control plane nodes and worker nodes that comprise the cluster can run different versions of Kubernetes, provided that you follow the Kubernetes version skew support policy detailing the maximum supported difference in versions between each component of a cluster. For example, kubectl is supported within one minor version, either older or newer, of the Kubernetes API server. The Kubernetes control plane must be upgraded before the data plane.

**Upgrading the Control Plane**

You upgrade control plane nodes by upgrading the cluster and specifying a more recent Kubernetes version for the cluster. Control plane nodes running older versions of Kubernetes are upgraded. Because Container Engine for Kubernetes distributes the Kubernetes control plane on multiple Oracle-managed control plane nodes to ensure high availability (distributed across different availability domains in multi-AD regions where this is possible), you’re able to upgrade the Kubernetes version running on control plane nodes with zero downtime to the Kubernetes API. The steps for upgrading the control plane are as follows:

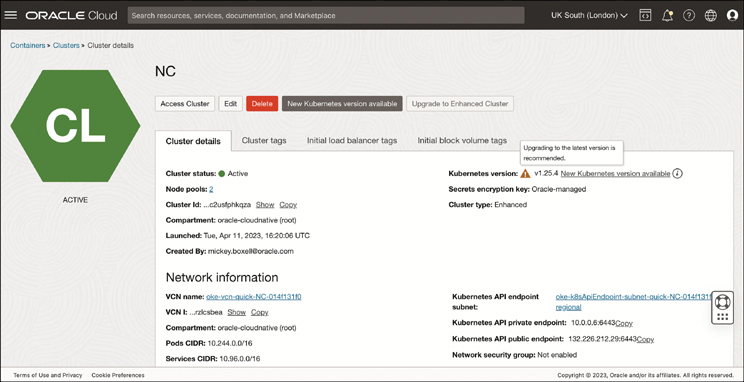
Step 1.In the OCI Console, open the navigation menu and click **Developer Services**.

Step 2.Under Containers & Artifacts, click **Kubernetes Clusters (OKE)**.

Step 3.From the Compartment drop-down menu, choose a compartment that contains a cluster you have permission to upgrade.

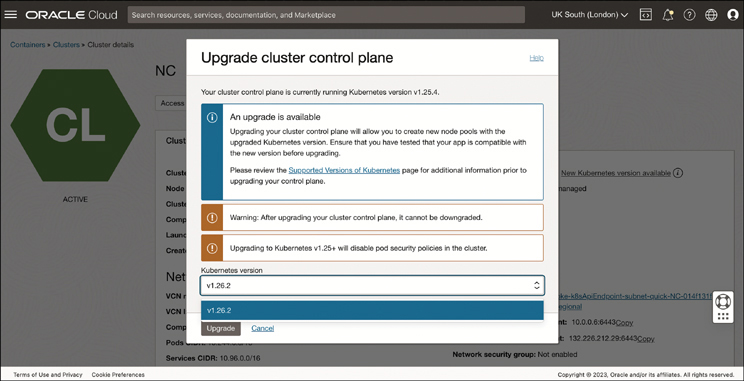
Step 4.The Version column displays the Kubernetes version of each cluster. Clusters running older versions display a warning stating, “Upgrading to the latest version is recommended.” Click the name of the cluster you want to upgrade.

Step 5.On the Cluster Details page, if a newer Kubernetes version than the one running on the control plane nodes in the cluster is available (see [Figure 5-2](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#ch05fig02)), the Upgrade Available button is enabled at the top of the page.



**Figure 5-2** Cluster Details Page with a Recommendation to Upgrade to the Latest Available Kubernetes Version

Step 6.Click **Upgrade Available** and select the Kubernetes version to which you want to upgrade the control plane nodes (see [Figure 5-3](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#ch05fig03)). Keep in mind that, after you upgrade your cluster control plane, it cannot be downgraded.



**Figure 5-3** Upgrade Cluster Control Plane Window That Enables You to Select a New Kubernetes Version for Your Control Plane

Step 7.Click **Upgrade** to apply the update to your cluster.

Step 8.After you click Upgrade, you will see the status of the cluster change to Updating. You can track the status of the upgrade by clicking the **Work Requests** tab and then clicking the work request to open the Work Requests Details page. When the process is complete, the work request status moves from In Progress to Succeeded and the control plane nodes will be running the newer Kubernetes version. From that point on, the new Kubernetes version will appear as an option when defining node pools parameters.

You can update the cluster control plane using the CLI by passing in the OCID of the cluster you want to upgrade and the Kubernetes version you want to upgrade to:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05_images.xhtml#f0169-01a)

oci ce cluster update --cluster-id <cluster-ocid> --kubernetes-version

<kubernetes-version-number>

If you want to upgrade to a version of Kubernetes that is more than one version ahead of the version currently running on the control plane nodes, you *must* upgrade to each intermediate version in sequence, without skipping versions. Skipping minor versions while upgrading a control plane is not supported by OKE or the upstream Kubernetes project.

**Upgrading the Data Plane**

Following the upgrade of your control plane to a new version of Kubernetes, you can create worker nodes running the newer version as well. Alternatively, you can continue to create nodes running older versions of Kubernetes, as long as the older versions are compatible with the control plane Kubernetes version. As long as you follow the Kubernetes version skew support policy described in the Kubernetes documentation, the control plane nodes and worker nodes that comprise the cluster can run different versions of Kubernetes. Each component of a cluster has its own skew policy, and the official Kubernetes documentation should be treated as the ultimate source of truth when it comes to compatibility. Generally, worker nodes are expected to function without issue as long as their kubelet version is within two minor versions of the control plane. For example, the worker node version for a 1.26 control plane should be 1.26, 1.25, or 1.24.

When it comes to upgrading worker nodes, OKE recommends following the approach of the upstream Kubernetes project by treating nodes as “cattle, not pets.” For those unfamiliar, this phrase is used to describe an environment in which you should consider your resources as easily replaceable as opposed to something you would feel distressed about losing. In this context, that means that rather than updating the kubelet on existing worker node hosts, you should terminate the instance and replace it with one running the updated Kubernetes version. This approach applies to the host OS version and other node pool properties as well. Given this recommendation, you can choose one of two paths: upgrading an existing node pool or creating a new node pool.

OKE enables you to define the properties of a node pool, including Kubernetes version, host image, shape, and more, which are then passed down to all the nodes in a pool. This also allows you to update the properties of a node pool. Any node created after the properties are updated will come online with the updated properties. For example, if you change the node pool’s Kubernetes version to the latest available Kubernetes version, only nodes created after the change is made will come online with the new version. Existing nodes will continue to run the previous Kubernetes version. The same is true for the other properties of the node pool. The question becomes, how do you move workloads from nodes running the old properties to nodes running the new ones: little by little or all at once?

**Upgrading an Existing Node Pool**

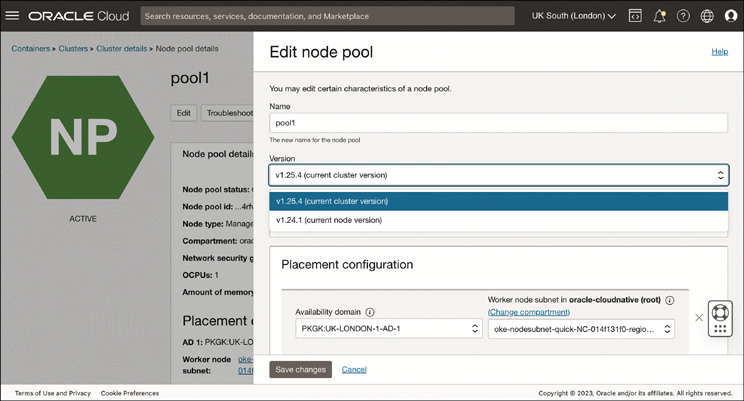
Upgrading an existing node pool provides you with a mechanism for moving workloads little by little. The approach is useful for users who lack extra capacity or prefer to keep costs low. This approach enables you to sequentially replace as few as one node at a time by adding nodes to a node pool or taking existing nodes offline as you incrementally move work to newly created ones that possess new parameters. You begin by specifying a more recent Kubernetes version for the existing node pool. Then you delete each worker node. OKE allows you to select cordon and drain options, which prevents new pods from starting on the target node or nodes and deleting existing pods, respectively. You can specify to have a new worker node be created to take the place of each worker node that you delete, to ensure that the node pool matches the desired node pool size. All new worker nodes starting in the pool will be running the more recent Kubernetes version you specified. Depending on your availability requirements, you can choose to scale the desired size of the node pool before you terminate nodes, to limit service disruptions. A larger capacity of nodes to run your workloads means more space for pods to be scheduled, as well as a lower likelihood of pods left unscheduled. We recommend leveraging pod disruption budgets for your application to ensure that a sufficient number of replica pods is running throughout the deletion operation. Pod disruption budgets limit the number of instances of your application that can be down at the same time because of a voluntary disruption. The steps for upgrading an existing node pool are as follows:

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

Step 2.On the Cluster List page, choose a compartment and click the name of the cluster where you want to change the Kubernetes version of the worker nodes.

Step 3.On the Cluster Details page, select the **Node Pools** tab. This tab shows the current Kubernetes version of the node pool(s). Click the name of the node pool whose nodes you want to upgrade.

Step 4.On the Node Pool Details page, click **Edit** and, in the Version field, specify the Kubernetes version required for your worker nodes (see [Figure 5-4](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#ch05fig04)). Keep in mind that the Kubernetes version you specify must be compatible with the version that is running on the control plane nodes. In the case of OKE images (a worker node image type optimized for use with OKE clusters), under Image, you need to click **Change Image** and, with **OKE Worker Node Images** selected as your Image Source, choose an image that matches the updated Kubernetes version. Click **Select Image**.



**Figure 5-4** Edit Node Pool Panel, Which Enables You to Modify the Properties of an Existing Node Pool—in This Case, by Selecting a New Kubernetes Version

Step 5.Click **Save Changes** to save the change.

Step 6.You must now delete existing worker nodes so that new worker nodes are started with the new Kubernetes version. For the first worker node in the node pool, on the Node Pool page, display the Nodes tab and select **Delete Node** from the Actions menu beside the node you want to delete. Either accept the defaults for advanced options or click **Show Advanced Options** and specify the alternatives, as follows:

* **Eviction Grace Period (Mins):** The length of time allowed for nodes to be cordoned and drained before termination. Either accept the default of 60 minutes or specify an alternative value between 0 and 60. To skip cordoning and draining worker nodes, specify 0 minutes.
* **Force Terminate After Grace Period:** The instruction on whether to terminate worker nodes at the end of the specified eviction grace period, even if they have not been successfully cordoned and drained. Select this option if you want worker nodes terminated at the end of the eviction grace period, regardless of whether they have been successfully cordoned and drained. Deselect this option if you want to preserve worker nodes that have not been successfully cordoned and drained by the end of the eviction grace period. Node pools containing worker nodes that fail to be terminated within the specified eviction grace period have the Needs Attention status. The status of the work request that initiated the termination operation will be set to Failed, and the termination operation will be cancelled.
* **Do Not Decrease Node Pool Size:** The option to maintain the current size of the node pool. By default, the size of the node pool will be decremented by one after a node is terminated, and a new node will not come online. To ensure that a new node running the updated Kubernetes version comes online, select this option.

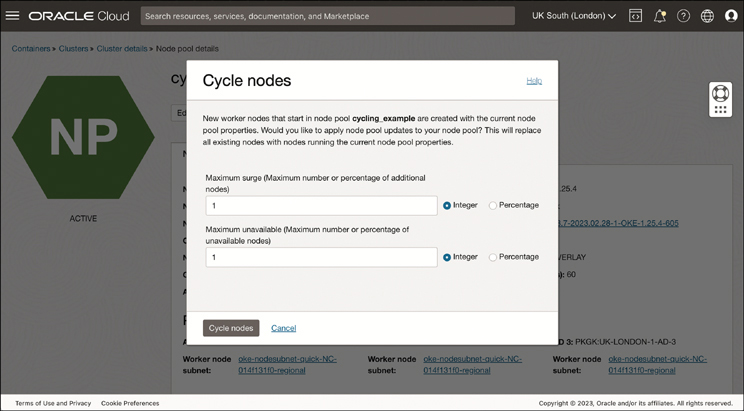
Step 7.Click **Delete** to delete the worker node. A work request is launched to delete the worker node. You can track the status of the work request by navigating to the Work Requests tab of the Node Pool Details page and choosing the appropriate work request from the Work Requests table.

Step 8.Repeat the previous step for each remaining worker node in the node pool until all worker nodes in the node pool are running the desired Kubernetes version.

As an alternative to steps 6–8, you can use the OKE on-demand node cycling feature. This feature provides a one-click operation to replace existing nodes with updated nodes. To minimize service disruption, you can specify a maximum number of additional nodes that can be added to the node pool during an upgrade, referred to as max-surge. Increasing max-surge raises the number of nodes that can be upgraded simultaneously. To account for budget and capacity constraints, you can specify a maximum number of nodes that can become unavailable at a given time during an upgrade. Increasing the max-unavailable value raises the number of nodes that can be upgraded in parallel in a single node pool. When using max-unavailable, nodes will be cordoned/drained based on the node pool’s evictionGracePeriod and terminated before a new node comes online, to avoid extra costs.

Step 6.After you save the changes to the node pool parameters in step 5, navigate to the Node Pool Details page and click **Cycle Nodes**.

Step 7.On the page shown in [Figure 5-5](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#ch05fig05), specify a value for **Maximum Surge**, the number of nodes that can be upgraded simultaneously. This value can be an integer or a percentage. This value cannot be greater than the total number of nodes currently running in the pool.



**Figure 5-5** The Cycle Nodes Window on the Node Pool Details Page Gives You the Ability to Cycle the Nodes in Your Node Pool

Step 8.Specify a value for **Maximum Unavailable**, the number of nodes that can simultaneously become unavailable. This value can be an integer or a percentage. This value cannot be greater than the total number of nodes currently running in the pool.

Step 9.Click **Cycle Nodes** to replace all the nodes in your node pool. A work request is launched to cycle the worker nodes. You can track the status of the work request by navigating to the Work Requests tab of the Node Pool Details page and choosing the appropriate work request from the Work Requests table.

**Upgrading by Adding a Node Pool**

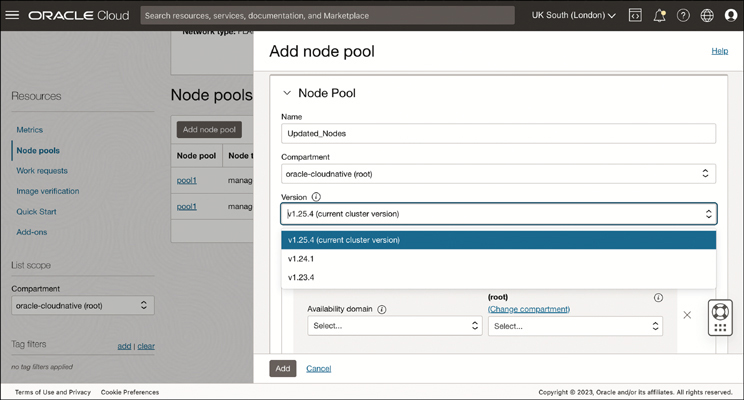
Another option for updating nodes in your cluster is to create an additional node pool and move workloads from your existing pool to the newly created one. This approach requires capacity to create a second node pool of the same size as your original node pool. The steps to do this are as follows:

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

Step 2.On the Cluster List page, choose a compartment and click the name of the cluster where you want to change the Kubernetes version of the worker nodes.

Step 3.On the Cluster Details page, select the **Node Pools** tab. This tab shows the current Kubernetes version of the node pool(s). Click **Add Node Pool** to create a new node pool with your desired parameters.

Step 4.On the Add Node Pool panel (see [Figure 5-6](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#ch05fig06)), in the Version field, specify the Kubernetes version required for your worker nodes. Keep in mind that the Kubernetes version you specify must be compatible with the version that is running on the control plane nodes. In the case of OKE images, a worker node image type optimized for use with OKE clusters, under Image, you need to click **Change Image** and then, with OKE Worker Node Images selected as your image source, choose an image that matches the updated Kubernetes version. Click **Select Image**. Fill out the rest of the node pool parameters with your desired values.



**Figure 5-6** The Add Node Pool Panel, Which Enables You to Add a New Node Pool with Your Chosen Options to Your Existing Cluster

Step 5.Click **Add** to create the new node pool.

Step 6.A work request is launched to delete the worker node. You can track the status of the work request by navigating to the Work Requests tab of the Node Pool Details page and choosing the appropriate work request from the Work Requests table.

Step 7.After the new node pool becomes active and the nodes in the new node pool become ready, you can begin to shift work from one pool to the other. To do so, for the first worker node in the original node pool, prevent new pods from starting on a node by entering **kubectl cordon** ***<node\_name>***. You can do this for multiple nodes in parallel using a label selector. To do so, label the nodes in your node pool and then enter **kubectl drain --selector** ***<your\_node\_pool\_label>***. After cordoning your nodes, you can delete pods running on those nodes by entering **kubectl drain** ***<node\_name>***. This can also be done in parallel by entering: **kubectl drain --selector** ***<your\_node\_pool\_label>***.

Step 8.When you have drained all the worker nodes from the original node pool and pods are running on worker nodes in the new node pool, you can delete the original node pool. On the Cluster Details page, click the **Node Pools** tab and then select **Delete Node Pool** from the Actions menu beside the original node pool.

Step 9.A work request is launched to delete the node pool. You can track the status of the work request by navigating to the Work Requests tab of the Node Pool Details page and choosing the appropriate work request from the Work Requests table.

**Alternative Host OS (Not Kubernetes Version) Upgrade Options**

Although we advocate for an immutable infrastructure approach in which nodes are terminated and re-created instead of being updated in place, it is possible to update existing nodes in place. This approach might be preferred for stateful workloads that are complicated or time-consuming to reschedule.

One option is to connect to your worker nodes using SSH and run a YUM update as you would with any OS to ensure that it is patched. If a reboot is required, drain your pods off that node while rebooting (or just let Kubernetes [k8s] reschedule them—no additional capacity is needed during upgrade).

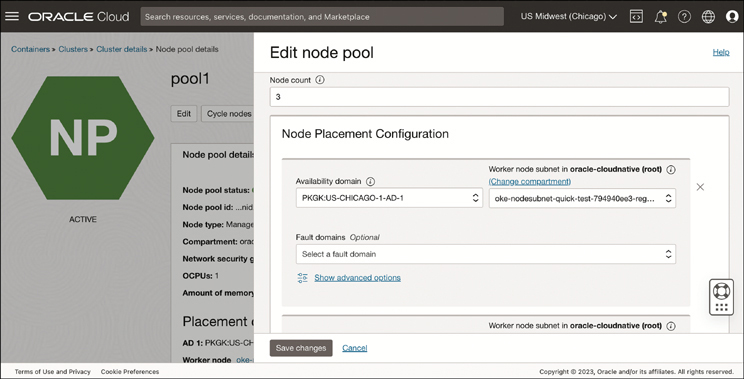
Run the os-updater tool on your worker nodes. If a reboot is required, drain your pods off that node while rebooting (or just let Kubernetes reschedule them). No additional capacity is needed during an upgrade.

**Scaling a Cluster**

Scaling OKE clusters occurs differently for the control plane and the data plane. The OKE platform scales the control plane nodes without the need for you to intervene.

**Manual Scaling**

The number of virtual nodes and managed nodes in a cluster can be scaled manually by specifying a new value for the node count property of the node pool or by adding more node pools with nodes to the cluster. When you increase or decrease the node count in the node pool, the service creates or destroys the required number of nodes to converge onto the new node count set on the node pool. When performing scale-up operations, the service respects the placement configuration provided on the node pool, and the nodes that are created have the most up-to-date configuration of that node pool. For instance, if you updated any of the node pool properties (for example, the cloud-init script), the updated configuration is applied to the new nodes that are created. Manual scaling can be performed using the OCI console, Terraform automation, the OCI CLI, or the APIs directly. You can scale applications manually by updating manifest files. [Figure 5-7](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#ch05fig07) shows the window in the Node Pool Details page that can be used to specify a new desired size for your node pool, as well as to define placement configurations used to control the distribution of nodes across availability domains and fault domains.



**Figure 5-7** Use the Edit Node Pool Page to Specify a New Desired Size for Your Node Pool and Define Placement Configurations Used to Control the Distribution of Nodes Across Availability Domains and Fault Domains

**Autoscaling**

To optimize resources, you can automatically scale at the node and pod levels of your cluster. Autoscaling at the node level is accomplished by deploying the Kubernetes Cluster Autoscaler. Autoscaling at the pod level is accomplished by deploying the Kubernetes Metrics Server to collect resource metrics from each worker node in the cluster and then deploying either the Kubernetes HorizontalPodAutoscaler (HPA), which is used to adjust the number of pods in a deployment, or the Kubernetes Vertical Pod Autoscaler (VPA), which is used to adjust the resource requests and limits for containers running in a deployment’s pods. Kubernetes resource requests are values specified when creating pods to control the resources (for example CPU and memory) guaranteed for a given container. The scheduler takes these values into account when deciding which node to schedule your workloads onto. Kubernetes resource limits are also specified when creating pods and are used to ensure a cap on the resources used by a given container. You will see examples of both resource management tools when we discuss VPA in the “[Vertical Pod Autoscaler](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#ch05lev3sec4)” section of this chapter.

Autoscaling the data plane depends on the type of nodes chosen for your node pools. In the case of virtual nodes, scaling data plane capacity based on workload demands is largely done for you. Infrastructure-level scaling of a virtual node pool is managed by OKE, which means that the underlying infrastructure capacity is scaled automatically and there is no need to use the Kubernetes Cluster Autoscaler. You can use the Kubernetes Cluster Autoscaler and the Kubernetes Vertical Pod Autoscaler with managed node pools only. You can use the Kubernetes Metrics Server and the Kubernetes HorizontalPodAutoscaler with both virtual node pools and managed node pools.

**Cluster Autoscaler**

You can use the Kubernetes Cluster Autoscaler to automatically resize the number of nodes in managed node pools based on workload demands. Resizing enables you to ensure the availability of your application and optimize costs. The Kubernetes Cluster Autoscaler adds worker nodes to a node pool when a pod cannot be scheduled in the cluster because of resource constraints; it removes worker nodes from a node pool when nodes have been underutilized for a given period of time and when pods can be rescheduled on other nodes.

**Note**

The Cluster Autoscaler scales the number of nodes in a node pool based on resource requests instead of the resource utilization of nodes in the node pool.

The Kubernetes Cluster Autoscaler is configured per node pool using a configuration file that specifies the node pools to target for expansion and contraction. The Cluster Autoscaler manages only node pools referenced in the configuration file. The file also enables you to specify the minimum and maximum sizes for each node pool, in addition to other settings. Because the Cluster Autoscaler itself runs on nodes in your cluster, it is important to ensure that at least one node pool in a cluster is not managed by the Kubernetes Cluster Autoscaler, to avoid a situation in which all Cluster Autoscaler nodes are evicted and cannot be rescheduled. This is also a good reason to ensure that you configure the Cluster Autoscaler deployment to have multiple replicas.

Keep in mind that although the Cluster Autoscaler is managing the capacity of your node pools, you should avoid manually changing the size of node pools or managing them using another tool, such as Terraform; the different capacity management systems will conflict as they try to reconcile one another’s changes to reach different desired states.

**Using the Cluster Autoscaler**

Before using the Cluster Autoscaler, you must configure the identity and access management policies to allow the Cluster Autoscaler to add and remove nodes from your node pools. To do so, follow these steps:

Step 1.Create a dynamic group to contain the node pools that you want to manage with the Cluster Autoscaler, and add a rule to the group that includes all instances in the compartment that you plan to manage with the Cluster Autoscaler: ALL {instance.compartment.id = '<compartment-ocid>'}.

Step 2.Next, create a policy to allow worker nodes to manage node pools:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05_images.xhtml#f0178-01a)

Allow dynamic-group acme-oke-cluster-autoscaler-dyn-grp to manage

cluster-node-pools in compartment <compartment-name>

Allow dynamic-group acme-oke-cluster-autoscaler-dyn-grp to manage

instance-family in compartment <compartment-name>

Allow dynamic-group acme-oke-cluster-autoscaler-dyn-grp to use subnets

in compartment <compartment-name>

Allow dynamic-group acme-oke-cluster-autoscaler-dyn-grp to read

virtual-network-family in compartment <compartment-name>

Allow dynamic-group acme-oke-cluster-autoscaler-dyn-grp to use vnics in

compartment <compartment-name>

Allow dynamic-group acme-oke-cluster-autoscaler-dyn-grp to inspect

compartments in compartment <compartment-name>

Replace <dynamic-group-name> with the name of the dynamic group you created earlier, and replace <compartment-name> with the name of the compartment in which the cluster belongs.

Step 3.Copy the Cluster Autoscaler configuration file, and edit it to apply to your node pools.

Step 4.Create a file called cluster-autoscaler.yaml with the following contents:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05_images.xhtml#f0178-02a)

---

apiVersion: v1

kind: ServiceAccount

metadata:

labels:

k8s-addon: cluster-autoscaler.addons.k8s.io

k8s-app: cluster-autoscaler

name: cluster-autoscaler

namespace: kube-system

---

apiVersion: rbac.authorization.k8s.io/v1

kind: ClusterRole

metadata:

name: cluster-autoscaler

labels:

k8s-addon: cluster-autoscaler.addons.k8s.io

k8s-app: cluster-autoscaler

rules:

- apiGroups: [""]

resources: ["events", "endpoints"]

verbs: ["create", "patch"]

- apiGroups: [""]

resources: ["pods/eviction"]

verbs: ["create"]

- apiGroups: [""]

resources: ["pods/status"]

verbs: ["update"]

- apiGroups: [""]

resources: ["endpoints"]

resourceNames: ["cluster-autoscaler"]

verbs: ["get", "update"]

- apiGroups: [""]

resources: ["nodes"]

verbs: ["watch", "list", "get", "patch", "update"]

- apiGroups: [""]

resources:

- "pods"

- "services"

- "replicationcontrollers"

- "persistentvolumeclaims"

- "persistentvolumes"

verbs: ["watch", "list", "get"]

- apiGroups: ["extensions"]

resources: ["replicasets", "daemonsets"]

verbs: ["watch", "list", "get"]

- apiGroups: ["policy"]

resources: ["poddisruptionbudgets"]

verbs: ["watch", "list"]

- apiGroups: ["apps"]

resources: ["statefulsets", "replicasets", "daemonsets"]

verbs: ["watch", "list", "get"]

- apiGroups: ["storage.k8s.io"]

resources: ["storageclasses", "csinodes"]

verbs: ["watch", "list", "get"]

- apiGroups: ["batch", "extensions"]

resources: ["jobs"]

verbs: ["get", "list", "watch", "patch"]

- apiGroups: ["coordination.k8s.io"]

resources: ["leases"]

verbs: ["create"]

- apiGroups: ["coordination.k8s.io"]

resourceNames: ["cluster-autoscaler"]

resources: ["leases"]

verbs: ["get", "update"]

---

apiVersion: rbac.authorization.k8s.io/v1

kind: Role

metadata:

name: cluster-autoscaler

namespace: kube-system

labels:

k8s-addon: cluster-autoscaler.addons.k8s.io

k8s-app: cluster-autoscaler

rules:

- apiGroups: [""]

resources: ["configmaps"]

verbs: ["create","list","watch"]

- apiGroups: [""]

resources: ["configmaps"]

resourceNames: ["cluster-autoscaler-status", "cluster-autoscaler-priority-expander"]

verbs: ["delete", "get", "update", "watch"]

---

apiVersion: rbac.authorization.k8s.io/v1

kind: ClusterRoleBinding

metadata:

name: cluster-autoscaler

labels:

k8s-addon: cluster-autoscaler.addons.k8s.io

k8s-app: cluster-autoscaler

roleRef:

apiGroup: rbac.authorization.k8s.io

kind: ClusterRole

name: cluster-autoscaler

subjects:

- kind: ServiceAccount

name: cluster-autoscaler

namespace: kube-system

---

apiVersion: rbac.authorization.k8s.io/v1

kind: RoleBinding

metadata:

name: cluster-autoscaler

namespace: kube-system

labels:

k8s-addon: cluster-autoscaler.addons.k8s.io

k8s-app: cluster-autoscaler

roleRef:

apiGroup: rbac.authorization.k8s.io

kind: Role

name: cluster-autoscaler

subjects:

- kind: ServiceAccount

name: cluster-autoscaler

namespace: kube-system

---

apiVersion: apps/v1

kind: Deployment

metadata:

name: cluster-autoscaler

namespace: kube-system

labels:

app: cluster-autoscaler

spec:

replicas: 3

selector:

matchLabels:

app: cluster-autoscaler

template:

metadata:

labels:

app: cluster-autoscaler

annotations:

prometheus.io/scrape: 'true'

prometheus.io/port: '8085'

spec:

serviceAccountName: cluster-autoscaler

containers:

- image: iad.ocir.io/oracle/oci-cluster-autoscaler:{{ image tag

}}

name: cluster-autoscaler

resources:

limits:

cpu: 100m

memory: 300Mi

requests:

cpu: 100m

memory: 300Mi

command:

- ./cluster-autoscaler

- --v=4

- --stderrthreshold=info

- --cloud-provider=oci-oke

- --max-node-provision-time=25m

- --nodes=1:5:{{ node pool ocid 1 }}

- --nodes=1:5:{{ node pool ocid 2 }}

- --scale-down-delay-after-add=10m

- --scale-down-unneeded-time=10m

- --unremovable-node-recheck-timeout=5m

- --balance-similar-node-groups

- --balancing-ignore-label=displayName

- --balancing-ignore-label=hostname

- --balancing-ignore-label=internal\_addr

- --balancing-ignore-label=oci.oraclecloud.com/fault-domain

imagePullPolicy: "Always"

env:

- name: OKE\_USE\_INSTANCE\_PRINCIPAL

value: "true"

- name: OCI\_SDK\_APPEND\_USER\_AGENT

value: "oci-oke-cluster-autoscaler"

Step 5.Change the image path of the Kubernetes Cluster Autoscaler image in the cluster-autoscaler.yaml file to an image stored in the OCIR Registry. Images are available in a number of regions:

1. Find the following line: - image: iad.ocir.io/oracle/oci-cluster-autoscaler:{{ image tag }}
2. Update it to the appropriate region and Kubernetes version: - image: phx.ocir.io/oracle/oci-cluster-autoscaler:1.25.0-6. This image is from the Phoenix region and was built to run on Kubernetes 1.25.

Step 6.Specify the node pools that you want the Cluster Autoscaler to manage. You can specify multiple node pools:

1. Find the following line: - --nodes=1:5:{{ node pool ocid 2 }}. This line is formatted as --nodes=<min-nodes>:<max-nodes>:<nodepool-ocid>. <min-nodes> and is used to define the minimum number of nodes allowed in the node pool. The number of nodes will not drop below this number. <max-nodes> is used to define the maximum number of nodes allowed in the node pool. The number of nodes will not increase above this number. <nodepool-ocid> is used to define the node pools you want to manage by the Cluster Autoscaler.
2. You can specify other supported Cluster Autoscaler parameters at this time.
3. Save the cluster-autoscaler.yaml file.

Step 7.Deploy the Kubernetes Cluster Autoscaler to the cluster:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05_images.xhtml#f0183-03a)

kubectl apply -f cluster-autoscaler.yaml

Step 8.View the Kubernetes Cluster Autoscaler logs to confirm a successful deployment. These logs also indicate whether the workload of node pools in the cluster is currently being monitored:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05_images.xhtml#f0183-01a)

kubectl -n kube-system logs -f deployment.apps/cluster-autoscaler

**Metrics Server**

The Kubernetes Metrics Server is a set of Kubernetes resources that you can deploy to your cluster that collects resource metrics from the kubelet processes running on the data plane nodes in your cluster. Metrics Server collects these metrics and exposes them through the Kubernetes API, using custom resources that represent metric readings. To deploy the Kubernetes Metrics Server to a cluster with managed node pools, use the following command:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05_images.xhtml#f0183-02a)

kubectl apply -f https://github.com/kubernetes-sigs/metrics-server/releases/

download/<version-number>/components.yaml

Update <version-number> with the Kubernetes Metrics Server version that you want to deploy (for example, v0.6.3). Because the Kubernetes Metrics Server is being actively developed, the version number will change over time.

To deploy the Kubernetes Metrics Server to a cluster with virtual node pools, you need to first disable the liveness and readiness checks for the metric server because virtual nodes do not expose these URLs for the metric server. To make this change, download the manifest file components.yaml to a local directory from https://github.com/kubernetes-sigs/metrics-server/releases/download/<version-number>/components.yaml. Ensure that the URL is updated with the latest version number, as in the previous example.

Open the components.yaml file in a text editor of your choice, and remove the livenessProbe and readinessProbe sections (see [Listings 5-1](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#list5_1) and [5-2](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#list5_2)) from the manifest of the metrics-server deployment.

**Listing 5-1** livenessProbe Section

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05_images.xhtml#f0184-01a)

livenessProbe:

failureThreshold: 3

httpGet:

path: /livez

port: https

scheme: HTTPS

periodSeconds: 10

**Listing 5-2** readinessProbe Section

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05_images.xhtml#f0184-02a)

readinessProbe:

failureThreshold: 3

httpGet:

path: /readyz

port: https

scheme: HTTPS

initialDelaySeconds: 20

periodSeconds: 10

Deploy the Kubernetes Metrics Server by entering this line:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05_images.xhtml#f0184-03a)

kubectl apply -f <local-location>/components.yaml

Update with the local directory containing the components.yaml file that you just modified.

Confirm that the Kubernetes Metrics Server has been deployed successfully and is available by entering this line:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05_images.xhtml#f0184-04a)

kubectl get deployment metrics-server -n kube-system

**HorizontalPodAutoscaler**

You can use the Kubernetes HorizontalPodAutoscaler to automatically scale a workload resource, such as the number of pods in a deployment, a replica set, or a stateful set, based on that resource’s CPU or memory utilization. The HorizontalPodAutoscaler can help applications scale out to meet increased demand or scale when demand decreases. You can set a target metric percentage to meet when scaling applications.

The HorizontalPodAutoscaler does not need to be manually installed into a cluster because it is a standard API resource in Kubernetes. However, it does require the installation of a metrics source, such as the Kubernetes Metrics Server. The Metrics Server is used to collect resource metrics directly from the kubelets running in your cluster. It exposes those metrics through the Kubernetes API. You can use the Kubernetes HorizontalPodAutoscaler with both managed node pools and virtual node pools.

To use the HorizontalPodAutoscaler, if you have not already done so, follow the steps to deploy the Kubernetes Metric Server.

Next, create a HorizontalPodAutoscaler Resource. In this example, the resource will maintain a minimum of one and a maximum of five replicas and will aim for an average CPU utilization of 50%:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05_images.xhtml#f0185-01a)

kubectl autoscale deployment example --cpu-percent=50 --min=1 --max=5

This will maintain a minimum of one and a maximum of five replicas of the pods controlled by the example deployment. It will also increase and decrease the number of replicas of the deployment to maintain an average CPU utilization of 50% across all pods. If the average CPU utilization falls below 50%, the HorizontalPodAutoscaler will reduce the number of pods in the deployment to the minimum of 1 that you specified. If the average CPU utilization goes above 50%, the HorizontalPodAutoscaler will increase the number of pods in the deployment to the maximum of 5 that you specified.

You can confirm the current status of the HorizontalPodAutoscaler by entering this line:

kubectl get hpa

The output from the kubectl get hpa command shows the current status, including the deployments, target percentages, minimum and maximum pods, replicas, and age of the resource:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05_images.xhtml#f0185-02a)

NAME REFERENCE TARGET MINPODS MAXPODS REPLICAS AGE

php-apache Deployment/php-apache/scale 0% / 50% 1 10 1 30s

You can try out the HorizontalPodAutoscaler by deploying a sample application and then generating load against the application in reaction to the load. This sample application includes a container called hpa-example that is exposed using a service:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05_images.xhtml#f0185-03a)

apiVersion: apps/v1

kind: Deployment

metadata:

name: php-apache

spec:

selector:

matchLabels:

run: php-apache

template:

metadata:

labels:

run: php-apache

spec:

containers:

- name: php-apache

image: registry.k8s.io/hpa-example

ports:

- containerPort: 80

resources:

limits:

cpu: 500m

requests:

cpu: 200m

---

apiVersion: v1

kind: Service

metadata:

name: php-apache

labels:

run: php-apache

spec:

ports:

- port: 80

selector:

run: php-apache

To deploy the sample application, follow these steps:

Step 1.Enter this line:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05_images.xhtml#f0186-01a)

kubectl apply -f https://k8s.io/examples/application/php-apache.yaml

Step 2.Create an autoscaler, which can be accomplished using the kubectl autoscale command. To match a target CPU utilization across all pods, HorizontalPodAutoscaler dynamically increases and decreases the number of replicas of a given deployment. In this case, you set a target CPU utilization of 50%. You can also set a floor and a ceiling for the number of replicas, to ensure that the number of replicas does not drop below a minimum or increase beyond a maximum. In this case, you set a minimum of 1 and a maximum of 10 replicas. To create the HorizontalPodAutoscaler for the php-apache application, enter this line:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05_images.xhtml#f0187-01a)

kubectl autoscale deployment php-apache --cpu-percent=50 --min=1

--max=10

Step 3.To check the current status of HorizontalPodAutoscaler, enter the following: kubectl get hpa

You will see an output similar to this:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05_images.xhtml#f0187-02a)

NAME REFERENCE TARGET MINPODS MAXPODS REPLICAS AGE

php-apache Deployment/php-apache/scale 0% / 50% 1 10 1 30s

In this example, you can see in the TARGET column that the current CPU utilization is 0%. This is because there is currently no load on the application.

Step 4.To see HorizontalPodAutoscaler in action, you can generate load against the application. To do so, you create an additional pod to send a loop of requests to the php-apache application. In your terminal, enter the following:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05_images.xhtml#f0187-03a)

kubectl get hpa php-apache --watch

Open a second terminal to ensure that load generation continues, and enter the following:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05_images.xhtml#f0187-04a)

kubectl run -i --tty load-generator --rm --image=busybox:1.28

--restart=Never -- /bin/sh -c "while sleep 0.01; do wget -q -O-

http://php-apache; done"

After allowing time for the load generation pod to come online, you will begin to see a higher CPU utilization—for example:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05_images.xhtml#f0187-05a)

NAME REFERENCE TARGET MINPODS MAXPODS REPLICAS AGE

php-apache Deployment/php-apache/scale 300% / 50% 1 10 1 2m

Next, you will see additional replicas created to reach the target of 50% utilization across all pods:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05_images.xhtml#f0187-06a)

NAME REFERENCE TARGET MINPODS MAXPODS REPLICAS AGE

php-apache Deployment/php-apache/scale 300% / 50% 1 10 7 4m

In this example, CPU utilization increased to 300% of the original value specified in the resource request. Consequently, the HorizontalPodAutoscaler resized the deployment to seven replicas. If you generate even more load against the application, you will see HorizontalPodAutoscaler scale up to the maximum number of replicas, which you set as 10.

Step 5.You can see the resized replica count in the deployment itself by entering this line:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05_images.xhtml#f0188-01a)

kubectl get deployment php-apache

You will see an output similar to this:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05_images.xhtml#f0188-02a)

NAME READY UP-TO-DATE AVAILABLE AGE

php-apache 7/7 7 7 30s

After a few minutes, you will see the CPU utilization approximately reach the target utilization value:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05_images.xhtml#f0188-03a)

NAME REFERENCE TARGET MINPODS MAXPODS REPLICAS AGE

php-apache Deployment/php-apache/scale 45% / 50% 1 10 7 8m

Step 6.To stop generating load, navigate to the terminal where you created the busybox pod and enter the following:

<Ctrl> + C

After a few minutes, you will see the utilization drop down to 0% and HorizontalPodAutoscaler will scale down the replicas to the minimum of one:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05_images.xhtml#f0188-04a)

NAME REFERENCE TARGET MINPODS MAXPODS REPLICAS AGE

php-apache Deployment/php-apache/scale 0% / 50% 1 10 1 11m

Step 7.To delete the HorizontalPodAutoscaler, enter the following:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05_images.xhtml#f0188-05a)

kubectl delete horizontalpodautoscaler.autoscaling/php-apache

To remove the sample application, enter this:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05_images.xhtml#f0188-06a)

kubectl delete deployment.apps/php-apache service/php-apache

**Vertical Pod Autoscaler**

The Kubernetes Vertical Pod Autoscaler (VPA) can improve cluster resource utilization by automatically adjusting the resource requests and limits for containers running in a deployment’s pods. The Vertical Pod Autoscaler can update resource requests automatically based on usage to right-size the resources available for each pod while maintaining ratios between limits and requests specified in the initial container’s configuration. This applies to pods that are over-requesting resources and under-requesting resources based on their usage over time. The Vertical Pod Autoscaler has three components:

* The **Recommender** monitors resource consumption and provides recommended CPU and memory request values for a container.
* The **Admission plug-in** configures new pods to use the recommended resource requests on new pods that are created or re-created because of changes made by the Updater.
* The **Updater** checks for pods with incorrect resources and terminates them so that they can be re-created with the updated request values.

Unlike the HorizontalPodAutoscaler, which is already present on clusters, the Vertical Pod Autoscaler must be deployed to your cluster. As with the HorizontalPodAutoscaler, the Vertical Pod Autoscaler requires the installation of a metrics source, such as the Kubernetes Metrics Server, in the cluster.

To use the HorizontalPodAutoscaler, if you have not already done so, first follow the steps to deploy the Kubernetes Metric Server. Then deploy the Vertical Pod Autoscaler as follows:

Step 1.Download the Vertical Pod Autoscaler source code from GitHub:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05_images.xhtml#f0189-01a)

git clone https://github.com/kubernetes/autoscaler.git

Step 2.Change to the vertical-pod-autoscaler directory:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05_images.xhtml#f0189-02a)

cd autoscaler/vertical-pod-autoscaler

Step 3.Deploy the Vertical Pod Autoscaler:

./hack/vpa-up.sh

Step 4.Verify that the Vertical Pod Autoscaler pods have been created successfully:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05_images.xhtml#f0189-03a)

kubectl get pods -n kube-system

The output from this command shows the vpa-admission-controller, vpa-recommender, and vpa-updater pods:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05_images.xhtml#f0189-04a)

vpa-admission-controller-7c7666f6cd-lcjzn 1/1 Running 0 8s vpa-

recommender-786476d7cc-7qk7k 1/1 Running 0 11s vpa-

updater-79d74db98b-f2zv7 1/1 Running 0 13s

After deploying the Vertical Pod Autoscaler pods, you can use them to recommend and set resource requests for resources in your cluster. To do so, you must create a VPA config for the resource for which you want to receive recommendations or to autoscale. When it comes to the manner in which resources are managed, VPA offers the choice of these modes:

* In **Auto** mode, VPA assigns resource requests at pod creation time and updates resource requests of existing pods.
* In **Recreate** mode, VPA assigns resource requests at pod creation time and updates resource requests of existing pods. Recreate differs from Auto because, with Recreate, pods are evicted when updates to requested resources differ significantly from the original recommendation. This is used when you must ensure that pods are restarted when resource requests change.
* In **Initial** mode, VPA assigns resource requests only at pod creation time and does not update resource requests at any other time.
* In **Off** mode, VPA does not automatically assign resource requests for pods. Recommendations are still calculated and are available for you to review in the VPA object. This mode essentially allows you to audit the values that VPA recommends instead of having them actively applied.

**Note**

In the example in [Listing 5-3](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#list5_3), the update mode is not specified. By default, the mode is set to Auto.

You can try out the Vertical Pod Autoscaler by deploying a sample application and then generating load against the application in reaction to the load. This sample application includes a container called hamster and a VPA config, as demonstrated in [Listing 5-3](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#list5_3).

**Listing 5-3** Sample Application and VPA Configuration to Scale the Application

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05_images.xhtml#f0190-01a)

# This config creates a deployment with two pods, each requesting 100 millicores

# and trying to utilize slightly above 500 millicores (repeatedly using CPU for

# 0.5s and sleeping 0.5s).

# It also creates a corresponding Vertical Pod Autoscaler that adjusts the

# requests.

# Note that the update mode is left unset, so it defaults to "Auto" mode.

---

apiVersion: "autoscaling.k8s.io/v1"

kind: VerticalPodAutoscaler

metadata:

name: hamster-vpa

spec:

# recommenders field can be unset when using the default recommender.

# When using an alternative recommender, the alternative recommender's name

# can be specified as the following in a list.

# recommenders:

# - name: 'alternative'

targetRef:

apiVersion: "apps/v1"

kind: Deployment

name: hamster

resourcePolicy:

containerPolicies:

- containerName: '\*'

minAllowed:

cpu: 100m

memory: 50Mi

maxAllowed:

cpu: 1

memory: 500Mi

controlledResources: ["cpu", "memory"]

---

apiVersion: apps/v1

kind: Deployment

metadata:

name: hamster

spec:

selector:

matchLabels:

app: hamster

replicas: 2

template:

metadata:

labels:

app: hamster

spec:

securityContext:

runAsNonRoot: true

runAsUser: 65534 # nobody

containers:

- name: hamster

image: registry.k8s.io/ubuntu-slim:0.1

resources:

requests:

cpu: 100m

memory: 50Mi

command: ["/bin/sh"]

args:

- "-c"

- "while true; do timeout 0.5s yes >/dev/null; sleep 0.5s; done"

To create the deployment and config, enter this line:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05_images.xhtml#f0191-01a)

kubectl create -f examples/hamster.yaml

Deploying the hamster application creates a deployment with two pods and a Vertical Pod Autoscaler pointing at the deployment. Similar to HorizontalPodAutoscaler, VPA enables you to specify minimum and maximum values for the target resources using minAllowed and maxAllowed. You can verify that the application is deployed using the following command:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05_images.xhtml#f0192-01a)

$ kubectl get pods -l app=hamster

NAME READY STATUS RESTARTS AGE

hamster-65cd4dd797-8vglx 1/1 Running 0 43s

hamster-65cd4dd797-pcwpf 1/1 Running 0 44s

Describing one of the pods in the application shows the resource requests for that pod. Replace the pod name with one of the pods running in your cluster, and navigate to the requests section of the output to see the current request values, as demonstrated in [Listing 5-4](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#list5_4).

**Listing 5-4** Current Requested CPU and Memory Values from the Sample Application Pod

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05_images.xhtml#f0192-02a)

kubectl describe pod hamster-65cd4dd797-pcwpf

[… output truncated…]

requests:

cpu: 100m

memory: 50Mi

[… output truncated…]

Each of the pods in the hamster application runs a container that tries to utilize more cores and memory than requested. VPA watches these pods and, after a few minutes, updates the CPU and memory request to match the needs of the application. To see this happen in real time, watch the pods running in the application and wait for VPA to start a new pod with updated request values:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05_images.xhtml#f0192-03a)

kubectl get --watch pods -l app=hamster

When you see a new pod come online, describe the pod and navigate to the requests section of the output to see the current request values, as demonstrated in [Listing 5-5](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#list5_5).

**Listing 5-5** Updated Requested CPU and Memory Values from the Sample Application Pod

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05_images.xhtml#f0192-04a)

kubectl describe pod hamster-7cbfd64f57-wmg4

[… output truncated…]

requests:

cpu: 587m

memory: 262144k

[… output truncated…]

You can also view the recommendations made by VPA for the hamster application. To do so, enter the following:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05_images.xhtml#f0193-01a)

kubectl describe vpa/hamster-vpa

The Recommendation section of the output shows the recommendations:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05_images.xhtml#f0193-02a)

Name: hamster-vpa

Namespace: default

Labels: <none>

Annotations: <none>

API Version: autoscaling.k8s.io/v1

Kind: VerticalPodAutoscaler

Metadata:

Creation Timestamp: 2023-06-07T22:56:14Z

Generation: 4

Resource Version: 25877689

UID: 07e060c6-b46d-407c-961a-df92bcb6c6b6

Spec:

Resource Policy:

Container Policies:

Container Name: \*

Controlled Resources:

cpu

memory

Max Allowed:

Cpu: 1

Memory: 500Mi

Min Allowed:

Cpu: 100m

Memory: 50Mi

Target Ref:

API Version: apps/v1

Kind: Deployment

Name: hamster

Update Policy:

Update Mode: Auto

Status:

Conditions:

Last Transition Time: 2023-06-07T22:57:03Z

Status: True

Type: RecommendationProvided

Recommendation:

Container Recommendations:

Container Name: hamster

Lower Bound:

Cpu: 203m

Memory: 262144k

Target:

Cpu: 587m

Memory: 262144k

Uncapped Target:

Cpu: 587m

Memory: 262144k

Upper Bound:

Cpu: 1

Memory: 500Mi

Events: <none>

To remove the sample application, use the following command:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05_images.xhtml#f0194-01a)

kubectl delete -f examples/hamster.yaml

To remove the Vertical Pod Autoscaler, navigate to the directory into which it was downloaded, and use the following command:

./hack/vpa-down.sh

**Scaling Workloads and Infrastructure Together**

The Cluster Autoscaler scales nodes based on the resource requests made by pods, not the actual utilization of resources or performance characteristics of your workload. In most situations, you need a way to automatically detect degraded performance from your application and to automatically take actions to mitigate this. For instance, if your application receives more than an average amount of traffic, you might need to scale out to better serve the requests. This workload-level scaling is typically accomplished by using the HorizontalPodAutoscaler or the Vertical Pod Autoscaler.

Therefore, it is common practice to combine the Cluster Autoscaler and the HorizontalPodAutoscaler to automate the process of workload and infrastructure scaling, based on metrics. Continuing the example, if your pods are constantly hitting a high CPU utilization rate, the HorizontalPodAutoscaler can create new pods to spread the load, in order. This can create new resource requests from the new pods that are created, which might not be satisfied by the existing nodes. Once the existing resources in the cluster are exhausted, these pods are unschedulable because there are not enough resources to be allocated; then the Cluster Autoscaler kicks into action and creates new nodes to satisfy the updated resource requests. In this manner, you can scale both your workload and the infrastructure required to run the workload, based on the operational characteristics and metrics from your workload.

**Autoscaler Best Practices**

It is a best practice to design applications running on node pools managed by the Cluster Autoscaler to be disruption tolerant—for example, by means of pod disruption budgets. The Cluster Autoscaler automatically and dynamically adds and removes worker nodes. In the process, pods are moved from one worker node to another, which can cause disruptions if not properly accounted for. The Kubernetes Cluster Autoscaler respects pod scheduling and eviction rules, including the eviction grace period configured for your node pool. These rules might prevent the Cluster Autoscaler from being able to terminate a worker node.

When deploying workloads across availability domains, we recommend that you create one node pool per availability domain. For instance, when working with a region that has three availability domains, you can create three node pools: one for each availability domain. Now you can configure the Cluster Autoscaler to scale each of those node pools independently. Node pools in Container Engine for Kubernetes use a placement configuration to determine the spread of nodes across availability domains and fault domains. The node pool always tries to balance the number of nodes across its placement configuration. Consider this scenario when you have hit your soft resource limit for the specific CPU type or compute shape that you are using for your nodes in one of the availability domains. The shapes are available in the other availability domains; however, the single node pool with a placement configuration that spreads the nodes across all the three availability domains might not create new nodes because the spread of nodes would no longer be balanced. On the other hand, instead of creating a single node pool, if you configure separate node pools for each of the availability domains, you do not encounter this imbalance. This is because each node pool is restricted to a single availability domain and can scale independently of each other. Even if you hit a soft resource limit in a single availability domain, only that node pool is prevented from scaling, whereas the others can still provide new capacity for your workloads.

Because the Cluster Autoscaler itself is a set of pods that you deploy onto your cluster, you should ensure that you deploy multiple replicas for the autoscaler into your cluster. If the autoscaler itself is evicted for a higher-priority workload during a scaling event, the cluster no longer has the capability to scale. You should also ensure that multiple replicas of the autoscaler are deployed across multiple nodes.

**Caution**

When you use the Cluster Autoscaler to manage a node pool, you should *not* manually update or scale this node pool. The autoscaler will notice this change and could override your manual changes with the autoscaler’s configuration.

The Cluster Autoscaler and the HorizontalPodAutoscaler are often used in conjunction with each other. It is therefore important to understand the interactions between these two mechanisms and know how to control their behavior. For instance, setting an appropriate stabilization window in the HorizontalPodAutoscaler for either the scale-up or the scale-down events can control how aggressively new pods are created or destroyed. Transitively, this also impacts the frequency with which the Cluster Autoscaler creates and destroys new nodes. Misconfigurations with these interactions could potentially lead to race conditions or aggressive node scale-up or scale-down events that might not be productive.

**Cluster Access and Token Generation**

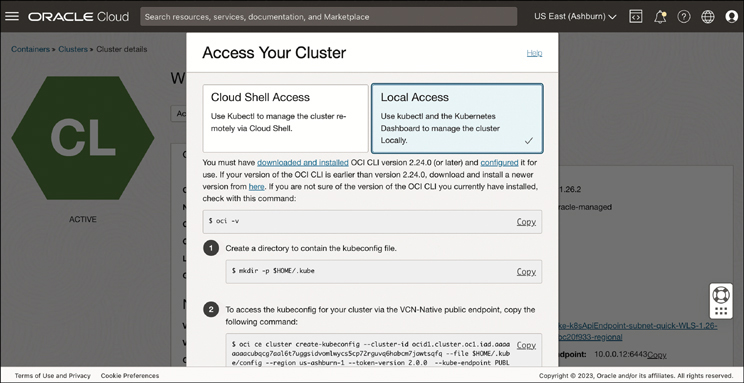
You can use the Kubernetes command-line tool, kubectl, to perform operations on clusters created with Container Engine for Kubernetes. kubectl is used to communicate with a Kubernetes cluster’s control plane using the Kubernetes API, for example, to create, get, describe, and delete resources in your cluster. Your version of kubectl must be compatible with the Kubernetes version of your OKE control plane. To be compatible, Kubectl must be either the same version as your control plane or one version ahead or behind your control plane. kubectl uses a configuration file, or kubeconfig, stored in the $HOME/.kube directory.

To access a cluster configured with a private Kubernetes API endpoint, you must configure your virtual cloud network for access or configure a bastion using the Oracle Cloud Infrastructure Bastion service.

Step 1.Install kubectl following the upstream Kubernetes documentation, if you have not already done so.

Step 2.Set up the kubeconfig file.

1. Generate an API signing key pair. Navigate to the Profile menu in the console, and click **User Settings**. Click the **API Keys** tab and then click **Add API Key**. Either use OCI to generate an API key pair or upload a PEM format key pair generated yourself, paste the contents of the public key, and click **Add**.
2. If you have not already done so, install and configure the Oracle Cloud Infrastructure CLI.
3. Set up the kubeconfig file by navigating to the Cluster Details page of the Kubernetes cluster you want to access and clicking **Access Cluster**, as shown in [Figure 5-8](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#ch05fig08). These instructions include the capability to connect using Cloud Shell, the command line built directly into the OCI console, or using local access.



**Figure 5-8** Steps for Configuring Cluster Access on the Cluster Details Page of the OCI Console

Clicking the **Local Access** option opens the steps for how to create a /.kube directory to store the kubeconfig file and how to access the kubeconfig for your cluster via the VCN-Native public or private endpoint. Choose the appropriate options, and then set your kubeconfig variable with export KUBECONFIG=$HOME/.kube/config. If a kubeconfig file already exists in the specified location, details for the new cluster are added as a new context to the existing kubeconfig file, and the current context element in the kubeconfig file is updated to point to the newly created cluster context.

In your terminal, enter kubectl, followed by the command for the operation you want to perform on the cluster—for example, kubectl get pods.

A single kubeconfig file can include the details for multiple Kubernetes clusters. Each cluster is referred to as a *context*. The cluster specified by the current context in the kubeconfig file is the cluster on which operations will be performed. The kubeconfig file generated by OKE includes an OCI CLI command that dynamically generates an authentication token and inserts it when you run a kubectl command. For this to function correctly, the OCI CLI must also be available on your shell’s executable path. The authentication tokens generated by the OCI CLI command in the kubeconfig file are short lived and specific to individual users, which means they cannot be shared between users to access a Kubernetes cluster.

**Service Account Authentication**

In addition to authenticating access to a Kubernetes cluster by means of an automatically generated OCI CLI command, users can authenticate by means of a Kubernetes service account. In some situations, automatically generated authentication tokens might be impractical. For example, you might be leveraging tools such as continuous integration and continuous delivery (CI/CD) pipelines that require long-lived authentication tokens. One solution is to use a Kubernetes service account:

Step 1.Begin by creating a service account: kubectl -n kube-system create serviceaccount <service account name>.

Step 2.Create a new clusterrolebinding with permissions appropriate for your use case, and bind it to the service account you just created. For example, this is how to create a clusterrolebinding with administrative access: kubectl create clusterrolebinding --clusterrole=cluster-admin --serviceaccount=kube-system: <service account name>.

**Note**

It is important to properly scope the permissions of roles in your cluster. The cluster admin role has every permission for every resource in the cluster, which is most likely too wide a scope for any one user in your organization.

Step 3.Next, create a Kubernetes secret that stores the authentication token for the service account.

1. To do so, create a kubeconfig-secret.yaml file with the following content:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05_images.xhtml#f0198-02a)

apiVersion: v1

kind: Secret

metadata:

name: kubeconfig-secret

namespace: kube-system

annotations:

kubernetes.io/service-account.name: kubeconfig-sa

type: kubernetes.io/service-account-token

1. Create the token with kubect apply -f kubeconfig-secret.yaml.
2. View the details of the secret by describing the secret you just created: kubectl describe secrets kubeconfig-secret -n kube-system.
3. The output from the preceding command includes a Base64-encoded authentication token as a value of the token element. Obtain the value of the service account authentication token, and assign its value to an environment variable after decoding it from Base64: TOKEN=kubectl -n kube-system get secret oke-kubeconfig-sa-token -o jsonpath=’{.data.token}’ | base64 --decode.

Step 4.You can then add the service account and the associated service account authentication token as a user in the kubeconfig file. Add the service account (and its authentication token) as a new user definition in the kubeconfig file by entering the following kubectl command: kubectl config set-credentials --token=$TOKEN.

Step 5.Set the user in the kubeconfig file for the current cluster context to be the new service account user you created: kubectl config set-context --current --user=<service-account-name>.

Step 6.After doing so, other tools may use the service account authentication token when accessing the cluster.

This kubeconfig file can be used across processes and tools to access the cluster.

**Configuring DNS**

Kubernetes uses DNS records pervasively for services and pods so that they can be discovered and communicated with using DNS names instead of IP addresses. When Kubernetes creates pods and services, it publishes information that the kubelet uses to configure DNS entries for them. OKE clusters use CoreDNS, a general-purpose DNS server that also supports plug-ins, as its DNS server, which you will see running as a pod in the kube-system namespace of your cluster.

CoreDNS maintains its configuration properties in a configuration file referred to as a Corefile. When deployed on Kubernetes, the Corefile is maintained as a Kubernetes ConfigMap that is provided to the CoreDNS pods when they are launched. You can view the default CoreDNS settings by using the command demonstrated in [Listing 5-6](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#list5_6).

**Listing 5-6** Default CoreDNS Settings of an OKE Cluster

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05_images.xhtml#f0199-01a)

kubectl get cm coredns -n kube-system -o yaml

apiVersion: v1

data:

Corefile: |-

.:53 {

errors

health {

lameduck 5s

}

ready

kubernetes cluster.local in-addr.arpa ip6.arpa {

pods insecure

fallthrough in-addr.arpa ip6.arpa

}

prometheus :9153

forward . /etc/resolv.conf

cache 30

loop

reload

loadbalance

}

import custom/\*.server

kind: ConfigMap

metadata:

name: coredns

namespace: kube-system

...TRUNCATED...

The default configuration includes several plug-ins, such as loadbalance and health, as in the example in [Listing 5-6](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#list5_6). Of these, the import plug-in is noteworthy and provides the mechanism for you to safely extend and customize the configuration. The default Corefile’s import plug-in is set up to import other configurations by looking for and importing config files that have the extension .server from the custom directory. The location of the custom directory is relative to the location of the Corefile.

When you need to customize CoreDNS behavior, such as when specifying a forwarding server for your network traffic, enabling logging for debugging DNS queries, or configuring your environment’s custom domains and upstream nameservers, you can override the default configuration by creating your own Corefile as a ConfigMap named coredns-custom. Because the default Corefile imports everything with a .server extension, all you have to do is make sure that the configmap you create has a key that ends in .server. Consider the example in [Listing 5-7](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#list5_7).

**Listing 5-7** Corefile, a configmap Used to Customize CoreDNS

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05_images.xhtml#f0200-01a)

apiVersion: v1

kind: ConfigMap

metadata:

name: coredns-custom

namespace: kube-system

data:

custom.server: |

corp.local {

cache

forward . \_IP\_ADDRESS\_OF\_YOUR\_RESOLVER\_

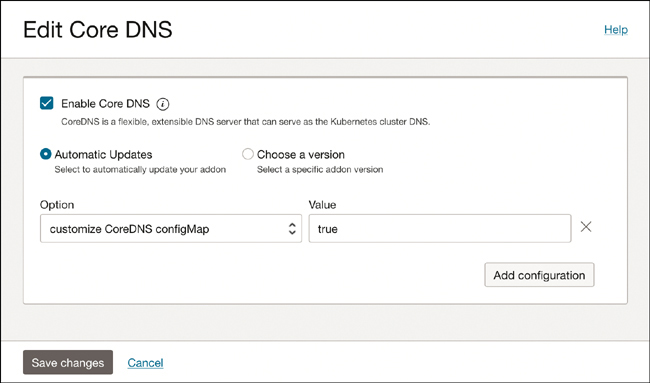
}

This ConfigMap defines a custom Corefile named custom.server that sends all requests within the *corp.local* domain to the nameserver whose IP address is provided to the forward plug-in. When the custom Corefile is created as a ConfigMap, as in [Listing 5-7](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#list5_7), the CoreDNS pods can be restarted to load the new config. Because CoreDNS runs as a DaemonSet, you can simply delete the pods to have Kubernetes re-create them with the updated configuration. The default pod definition for the CoreDNS pods loads the default Corefile from the ConfigMap and mounts it at /etc/coredns within the coreDNS container. The default pod definition also mounts an optional config volume from a ConfigMap named coredns-custom at /etc/coredns/custom. The customized Corefile, custom.server in [Listing 5-7](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#list5_7) is visible to the pod at /etc/coredns/custom/custom.server. The import plug-in from the default Corefile loads and merges this custom configuration because it meets the import pattern of ending in .server and is located in a directory named custom that is relative to the location of the default Corefile.

OKE also offers a way to customize and control the default Corefile contained in the configmap coredns, which is useful when you want to update the configuration and, say, remove plug-ins from the default plug-in chain. This can be done using the Add-Ons feature. CoreDNS is deployed as a cluster add-on, so it can be configured through the add-on setup process as well. To do this, follow these steps:

Step 1.Navigate to your cluster and the Add-ons section under Resources.

Step 2.Choose the CoreDNS add-on (see [Figure 5-9](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#ch05fig09)).



**Figure 5-9** Option to Customize CoreDNS by Passing in a Custom ConfigMap

Step 3.In the Options drop-down, choose **Customize CoreDNS ConfigMap**.

Step 4.In the value section, provide the value **true**.

With the CoreDNS add-on customized, you can provide your own default ConfigMap named coredns. During upgrades, OKE will not replace this customized configuration.

**Configuring Node Local DNS Cache**

Typically, pods that run with the dnsPolicy: ClusterFirst perform DNS queries using the kube-dns service. The service might be running on another node, and this can introduce latency for DNS lookups. The kube-dns service’s clusterIP is translated to the DNS server endpoint using IP tables rules. This can involve connection tracking, and when there are a lot of UDP DNS lookups, the conntrack table sometimes can fill up (usually because UDP entries have to time out in the conntrack table, unlike with TCP). To avoid these issues and improve performance, NodeLocal DNSCache runs a DNS caching agent on cluster nodes as a DaemonSet. Pods reach out to the DNS caching agent running on the same node always (because it is a DaemonSet) and thus can avoid iptables DNAT rules and connection tracking. If the local cache experiences a cache miss, it still queries kube-dns; however, the number of these queries that have to go through iptables rules DNAT and conntrack is significantly reduced.

**Note**

CoreDNS also runs its service under the name kube-dns to ensure compatibility with applications and systems that rely on that common name without having a dependency on the DNS server implementation.

The steps to set up the NodeLocal DNS cache are described in Kubernetes documentation[**1**](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#ref5_1a); however, this process involves setting a kubelet flag named --cluster-dns to override the default. To add this kubelet flag in OKE, you can use the Custom Cloud-Init feature described in [Chapter 4](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04.xhtml#ch04), “[Understanding Container Engine for Kubernetes](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04.xhtml#ch04),” which allows you to pass in these kubelet flags. Consider the following example cloud-init script:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05_images.xhtml#f0202-01a)

#!/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 --cluster-dns "CLUSTER\_DNS"

This simple script does not actually modify the cloud-init script itself. Instead, it downloads the default startup script and runs it with the --cluster-dns flag. The value of CLUSTER\_DNS should be set to something that does not collide with anything else on the cluster. For this reason, it is recommended that you use an address in the link local range of 169.254.0.0/16, such as 169.254.0.10.

**Configuring ExternalDNS**

When building public applications and services, you often expose your application using a Kubernetes service of type load balancer. A public load balancer is allocated a public IP address as well, but you then need to update the DNS entries for your application so that the domain name, such as [https://api.my-app.com](https://api.my-app.com/), can now point at your newly created load balancer’s IP address, making the application available over it. ExternalDNS is an add-on to Kubernetes that can create these DNS records for services in ExternalDNS providers, including OCI DNS. It eliminates the manual work of setting up DNS records in your DNS provider and makes Kubernetes services seamlessly discoverable. Note that ExternalDNS is not a DNS server such as CoreDNS. It does not perform the functions of DNS; instead, it automates the task of creating and updating DNS configuration in an ExternalDNS provider. The ExternalDNS provider supports OCI DNS, among several other DNS providers. The process of setting up and configuring ExternalDNS depends on the provider you want to use and is documented on the project’s GitHub page.[**2**](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#ref5_2a)

**Cluster Add-ons**

In the context of Kubernetes, the term *add-ons* refers to operational tools and features used to support and extend the functionality of Kubernetes clusters. This area includes software essential to the proper functioning of a cluster, such as CoreDNS, kube-proxy, and a container network interface (CNI) such as Flannel or Oracle Cloud Infrastructure (OCI) native pod networking. It also includes a growing portfolio of optional add-on software used to extend core Kubernetes functionality and improve cluster manageability and performance, such as the Kubernetes Dashboard, Oracle Database Operator, and WebLogic Kubernetes Operator. In a more concrete sense, add-ons include the software deployed to the kube-system namespace that is present by default when you create a cluster.

The OKE add-on feature gives you the capability to control add-ons deployed to your clusters. You can choose to disable or opt out of using a specific add-on altogether. For example, you can choose to disable the OCI native pod networking CNI and bring your own alternative CNI, such as Calico. If you have specific compliance or audit requirements, you can choose to pin to an add-on version and control when to update the add-on. Alternatively, you can choose to have Oracle fully manage your add-ons, including enabling automatic updates.

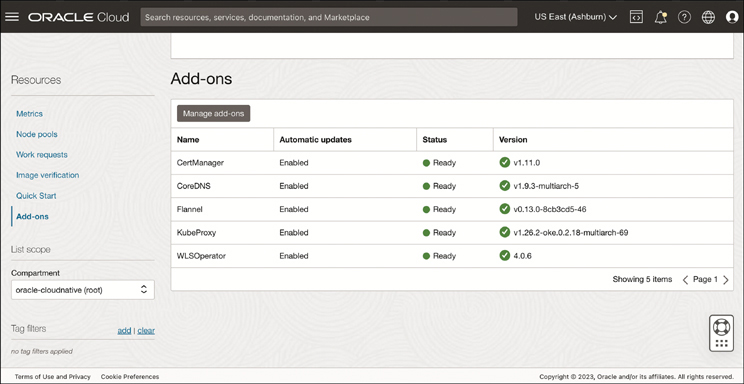
Each add-on comes with a set of customizable options. For example, CoreDNS, a general-purpose authoritative DNS server commonly found in Kubernetes environments, comes with the option to bring your own Kubernetes ConfigMap, with a Corefile section that defines CoreDNS behavior. This Corefile configuration includes several CoreDNS plug-ins with different DNS functions to extend the basic functionality. These supported customizations enable you to tailor your add-ons to your specific use cases while still benefitting from lifecycle management by Oracle.

**Configuring Add-ons**

You can configure add-ons either during the cluster creation process or after you create your cluster. To configure add-ons during cluster creation in the Console, click **Show Advanced Options** on the first page of the custom cluster creation flow, scroll down to the Configure Cluster Add-ons panel, and click the add-on you want to customize. For example, to use the optional Kubernetes dashboard add-on, you can click on the Kubernetes dashboard add-on to open a panel that enables you to choose whether you want to enable or disable the add-on. Enabling the add-on deploys the Kubernetes dashboard as a pod to your cluster. The same panel also allows you to choose to have Oracle automatically manage the lifecycle of the add-on, including updating the add-on as new versions are released over time, or to pin your add-on version and meet internal security and compliance requirements.

Each add-on comes with several configuration options. Some options are common to all add-ons; others are add-on specific. In the case of the Kubernetes dashboard, you can specify the number of replicas you want to make of the Kubernetes dashboard pod and then use node selectors and tolerations to control onto which nodes Kubernetes schedules a given add-on. For the complete list of key/value pairs used to pass on add-on specific arguments to the cluster, consult the Oracle documentation.

After you create your clusters, you can view your deployed add-ons by navigating to the Add-ons tab on the Clusters Details page. Here, you can see a list of deployed add-ons, details on whether they’re automatically updated, the status, and the add-on version (see [Figure 5-10](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#ch05fig10)).



**Figure 5-10** Displaying a List of the Add-ons Currently Deployed to Your Cluster

You can also configure add-ons after you create a cluster. To do so, click **Manage Add-ons**. This selection opens a panel showing all available add-ons. Clicking into an add-on gives you the capability to update the configuration. To apply the updates, click **Save Changes**. To track the changes in real time, navigate to the Work Requests tab and click the appropriate work request.

Not all add-ons can be deployed to all node types. For example, the Kubernetes dashboard runs on managed nodes but not virtual nodes.

**Disabling Add-ons**

Two options are available if you want to remove an add-on from your cluster: You can remove it, which actively deletes it from your cluster, or you can disable it, which leaves the pods running in your cluster. If you disable an essential cluster add-on, a warning indicates that you have taken responsibility for deploying and configuring an alternative add-on to provide equivalent functionality.

**Observability: Prometheus and Grafana**

Prometheus and Grafana are some of the most commonly used tools for metrics and monitoring with Kubernetes. The kube-prometheus project offers a “batteries included” experience to get started with Prometheus and Grafana on Kubernetes. It is built on top of the Prometheus Operator for Kubernetes that implements the operator pattern to manage Prometheus deployments on Kubernetes. The project also includes prebuilt Grafana dashboards and Prometheus rules to create an end-to-end solution for monitoring Kubernetes clusters. This project offers a good starting point for most Kubernetes users.

The kube-prometheus project can be deployed in two ways: directly using the manifests provided or using the Helm chart. Deploying from the manifest files is the simplest way to get started, but the kube-prometheus project uses Jsonnet to customize the manifests, if customization is desired. If you are new to Jsonnet, the Helm chart offers a simpler and more widely used method to customize the manifests before deployment. In this document, we use the Helm chart to deploy the kube-prometheus-stack.

**Monitoring Stack Components**

Several CNCF projects are used in combination to create this monitoring stack. These include the Prometheus Monitoring system and time series database; Alertmanager, which can deliver rule base alerts in response to events; and Grafana, for visualizing the monitoring data and interacting with it. The next sections look at each of these in detail.

**Installing the kube-prometheus-stack**

You can install the kube-prometheus-stack using Helm. The default installation does not use persistent storage, so the collected metrics will be lost if the pods are restarted. To overcome this, you can customize the values.yaml passed to the Helm chart, with added storage configuration. Create a new file named values-oci.yaml (see [Listing 5-8](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#list5_8)).

**Listing 5-8** File Used to Configure Default Values for the Prometheus Stack

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05_images.xhtml#f0206-01a)

cat <<EOF > values-oci.yaml

prometheus:

prometheusSpec:

storageSpec:

volumeClaimTemplate:

spec:

storageClassName: oci-bv

accessModes: ["ReadWriteOnce"]

resources:

requests:

storage: 50Gi

EOF

**Note**

This is a minimal configuration of the default values, and the charts offer a lot more configuration options. Notably, this setup does not cover the creation of an ingress resource for Grafana or Prometheus, so these services will be accessed later using a port-forward. If you want to configure a LoadBalancer or an ingress for the components, you can customize this through the Helm chart.

Now add the Helm repo and update the charts:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05_images.xhtml#f0206-02a)

helm repo add prometheus-community https://prometheus-community.github.io/

helm-charts

helm repo update

Install the chart, providing the values-oci.yaml with overrides for the default chart values.

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05_images.xhtml#f0206-03a)

helm install kube-prometheus-stack \

--namespace monitoring \

--create-namespace \

-f values-oci.yaml \

prometheus-community/kube-prometheus-stack

In a few moments, the Kubernetes resources are provisioned and the monitoring stack then is operational. To track the progress of the deployment, use the following command and ensure that all resources are ready:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05_images.xhtml#f0206-04a)

kubectl get statefulsets,deploy,svc,po -n monitoring

**Note**

You might have to run this command repeatedly, or you could prepend the command with watch -n5 to execute the command every 5 seconds until all resources are ready. This requires you to have the watch utility installed.

When all resources are ready, you can get the default password for the Grafana UI from the Kubernetes secret that was created during the deployment:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05_images.xhtml#f0207-01a)

kubectl get secret kube-prometheus-stack-grafana -o jsonpath="{.data.admin-

password}" -n monitoring | base64 --decode;echo

To log into the Grafana UI, you need to expose the service for Grafana. You can use a port-forward for this. The service that is created for Grafana is listening on port 80, by default. To create a port-forward, you can use the following command, which forwards port 3000 on the host to the service’s port 80:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05_images.xhtml#f0207-02a)

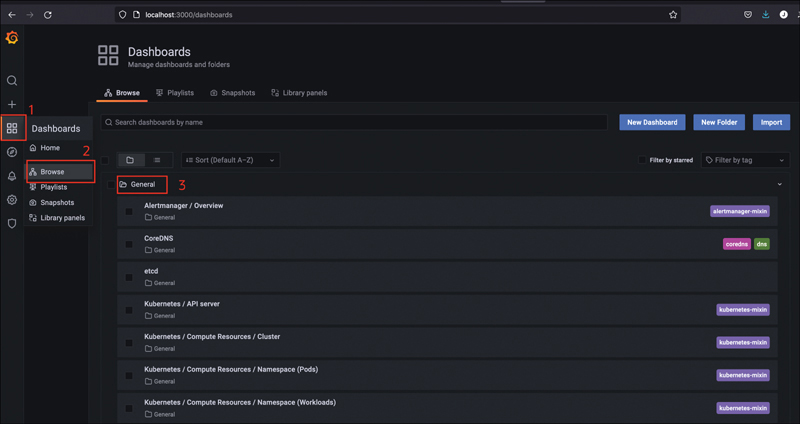
kubectl --namespace monitoring port-forward svc/kube-prometheus-stack-grafana

3000:80

You can now access the Grafana UI at http://localhost:3000.

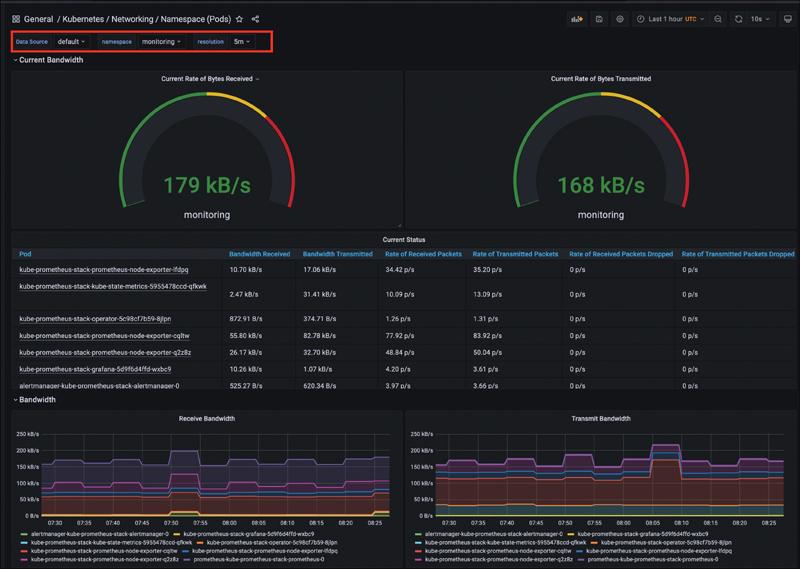
The kube-prometheus-stack [<https://github.com/prometheus-community/helm-charts/tree/main/charts/kube-prometheus-stack#kube-prometheus-stack>] comes bundled with a set of dashboards. These dashboards provide commonly used metrics and serve as examples of creating your own dashboards.

The bundled dashboards are grouped into the General folder. You can navigate to the list of dashboards through **Dashboards > Browse > General**, as shown in [Figure 5-11](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#ch05fig11).



**Figure 5-11** Set of Default Dashboards Available Through Grafana

From here, clicking any of the dashboards opens it and displays the metrics, as illustrated in [Figure 5-12](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#ch05fig12). Most dashboards are parametrized, meaning that you can specify the scope of the data displayed by narrowing the data to a specific cluster, namespace, and even resources within a namespace.



**Figure 5-12** Example Grafana Dashboard Monitoring Pod Bandwidth

**Operators and OCI Service Operator for Kubernetes**

Operators in Kubernetes are a way to package, run, and manage the entire lifecycle of a Kubernetes-native application, including actions such as upgrades. Any application that is built to rely on and take advantage of the Kubernetes features and tools for its operation and management can be called a Kubernetes-native application.

By itself, Kubernetes provides many features and capabilities to manage applications as pods, make them resilient to failures, and scale them up or down, based on need. However it also typically relies on the assumption that pods are arbitrarily replaceable. This model works well for stateless applications, and modern distributed application design practices promote this approach of preferring stateless services that inherently have properties such as better scalability. However, most applications require you to manage state; this state can be pushed down and consolidated further down the stack, but this still requires a workload-specific approach for handling replication of the state and for managing failover scenarios. For instance, how a sharded database is managed differs from how MySQL is managed, and Kubernetes would not be expected to know these differences. Similarly, some software components, such as a distributed cache, often have their own notion of what a “cluster” means, how to join and maintain membership in the cluster, and elect a leader among members of those specific components. Complex software systems can also have dependencies among their components, startup ordering requirements, specialized initialization and termination handling across dependent components, and more.

When the requirements of specialized stateful workloads go beyond the level of capabilities that Kubernetes offers as a general-purpose platform for all types of workloads, you need a way to manage that in a Kubernetes environment. When these applications operate outside Kubernetes, they often require the help of human actors to perform the required orchestration and to effectively *operate* the stack. If you move away from these manual operations and apply modern principles such as building software to automate the management of systems to this problem of managing complex and customized workflows for stateful applications in Kubernetes, you approach the notion of *operators*. For example, the Oracle database operator for Kubernetes knows the details of how to manage multiple types of databases, such as a sharded database on Kubernetes. In addition to making the process of setting up the workload easier, an operator also continuously monitors the workload and performs actions such as routine upgrades, data backups, and failover. Operators and the operator framework make use of the standard Kubernetes extension mechanisms. This means that operators work with the existing tooling for Kubernetes. Operators bring a custom resource definition (CRD) and the operator code that monitors for changes to the CRD and takes appropriate action.

Although operators make it easy to manage complex applications on top of Kubernetes, writing an operator can be challenging. The Operator SDK is a framework that uses the Kubernetes controller-runtime library to make writing operators easier. The Operator SDK provides high-level APIs and tools such as the operator-sdk CLI to make it easier to develop and work with operators.

Another component of the operator framework is the Operator Lifecycle Manager (OLM). The OLM provides an easy way for users to manage operators themselves. The process of installing, upgrading, and keeping operators up-to-date is made easier by using the OLM. For those developing operators, the OLM offers a model to package operators with declarative dependencies and offers discoverability for operators.

**Getting Started with Operators on OKE**

To install operators and work with them, the first step is to install the operator framework and tooling. The installation for the operator framework is documented in the official Operator Framework page.[**3**](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#ref5_3a) The following command showcases the installation on macOS:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05_images.xhtml#f0209-01a)

brew install operator-sdk

This installs the operator-sdk CLI tool, which offers a streamlined way to install the OLM. The OLM is installed into a target Kubernetes cluster, such as an OKE cluster:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05_images.xhtml#f0210-01a)

$ operator-sdk olm install --version 0.20.0

...

...

INFO[0079] Successfully installed OLM version "latest"

This installs the OLM and its required components into your cluster. It also creates a dedicated namespace named olm for these components. When OLM is installed on the cluster, you can use it to install and manage community operators. To list the operators available, use the following command:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05_images.xhtml#f0210-02a)

kubectl get packagemanifest -n olm

**Operators for OCI, Oracle Database, and Oracle WebLogic**

Oracle offers several operators that can help you manage and operate Oracle products and services directly from your Kubernetes cluster using familiar Kubernetes tooling. These range from operators that help you deploy and manage a sharded Oracle database on your Kubernetes cluster to the OCI Service Operator for Kubernetes (OSOK), which helps you create and manage OCI services such as the MySQL database or Object Storage buckets. It also includes operators that can manage WebLogic server clusters deployed on top of Kubernetes.

The OCI Service Operator for Kubernetes (OSOK)[**4**](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#ref5_4a) makes it easy to create, manage, and connect to OCI resources from a Kubernetes environment. OSOK supports the following services at the time of writing:

* Autonomous Database Service
* Oracle Streaming Service
* MySQL DB System Service
* Service Mesh Service

The operator enables you to manage these services as if they were part of your application. When the operator is installed,[**5**](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#ref5_5a) it sets up the custom resources that represent these resources in OCI. The operator also installs the controllers that react to these resource definitions by invoking the OCI APIs on your behalf to manage the service. This effectively allows you to manage OCI services using Kubernetes tooling and a Kubernetes resource definition in YAML. Consider the example in [Listing 5-9](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#list5_9).

**Listing 5-9** Managing OCI Services

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05_images.xhtml#f0210-03a)

apiVersion: oci.oracle.com/v1beta1

kind: MySqlDbSystem

metadata:

name: mysql\_db

spec:

compartmentId: ...compartment.ocid...

displayName: ApplicationDatabase

shapeName: MySQL.VM.Standard.E4.8.128GB

subnetId: ...subnet...

configuration:

id: MySQL.VM.Standard.E4.8.128GB.HA

availabilityDomain: ...avaiability.domain...

adminUsername:

secret:

secretName: ...kubernetes.secret...

adminPassword:

secret:

secretName: ...kubernetes.secret...

The manifest represents an object of kind: MySqlDbSystem, which is not a standard Kubernetes object, but a custom resource that is managed by the OSOK operator. When this manifest is deployed alongside the standard Kubernetes pods and services that make up an application, you are effectively creating the application database in OCI. The standard Kubernetes controllers take on the task of creating the pods for the application; the controller that the OSOK operator installed knows how to interpret the resource definition for the MySqlDbSystem and make the appropriate calls to OCI to create this database instance of OCI if it does not exist.

Another operator that is available from Oracle is the Oracle Database Operator. The Oracle Database Operator supports multiple database deployment models that you can directly manage from your Kubernetes cluster. These offer a wide range of configuration options from dedicated autonomous databases on the OCI infrastructure to multitenant databases and containerized sharded databases within the cluster.

The operator itself can be installed by following the documentation, or you can enable it as an add-on for your cluster. To install the operator through OKE add-ons, follow these steps:

Step 1.Navigate to the add-ons section for the cluster.

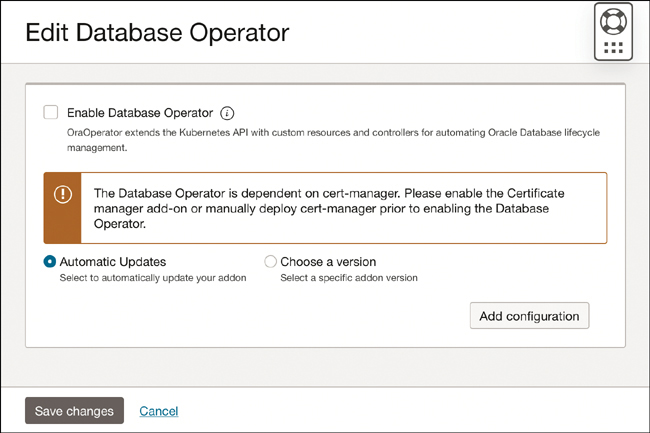
Step 2.Select **Oracle Database Operator**.

Step 3.Select a version or choose to keep the operator automatically updated.

Step 4.Select the check box to enable the operator add-on.

Step 5.Save your changes.

[Figure 5-13](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#ch05fig13) shows configuring the database operator as an add-on.



**Figure 5-13** Oracle Database Operator as a Cluster Add-on

As with any operator, you can use the CRDs that are enabled to describe the database and configuration you need; the operator then can create and help you operate the database. [Listing 5-10](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#list5_10) showcases the CRD for an Oracle sharded database.

**Listing 5-10** CRD for an Oracle Sharded Database

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05_images.xhtml#f0212-01a)

apiVersion: database.oracle.com/v1alpha1

kind: ShardingDatabase

metadata:

name: shardingdatabase-sample

namespace: shns

spec:

shard:

- name: shard1

storageSizeInGb: 50

- name: shard2

storageSizeInGb: 50

catalog:

- name: catalog

storageSizeInGb: 50

...TRUNCATED ...

Oracle sharding distributes segments of a data set across many databases (shards), which can be distributed across multiple systems or locations. When combined together, the individual shards make up a single logical database. This greatly improves the scalability of the database while still maintaining Oracle database features, such as powerful SQL and strong consistency guarantees. The Sharding Database controller that is installed to your cluster with the Oracle Database Operator deploys Oracle sharding topology as a stateful set when it encounters a CRD such as the one shown in [Listing 5-10](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#list5_10). The Sharding Database controller also manages the typical lifecycle of Oracle sharding topology in the Kubernetes cluster.

Additionally, Oracle offers a WebLogic Kubernetes Operator that can be used to create and manage WebLogic clusters and applications on Kubernetes. The WebLogic Kubernetes Operator is highly flexible and can be configured in a multitude of ways. These include the capability to configure WebLogic domains as Kubernetes resources (using CRDs), manage multiple WebLogic domains across namespaces, scale domains by adding or removing managed servers, integrate the operations with HorizontalPodAutoscaler, and more. Consider the partial example in [Listing 5-11](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#list5_11).

**Listing 5-11** Manifest Used to Create the WebLogic Kubernetes Operator

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05_images.xhtml#f0213-01a)

apiVersion: "weblogic.oracle/v1"

kind: Cluster

metadata:

name: sample-domain1-cluster-1

namespace: sample-domain1-ns

labels:

weblogic.domainUID: sample-domain1

spec:

replicas: 2

clusterName: cluster-1

---

apiVersion: "weblogic.oracle/v9"

kind: Domain

metadata:

name: sample-domain1

namespace: sample-domain1-ns

labels:

weblogic.domainUID: sample-domain1

spec:

configuration:

model:

auxiliaryImages:

- image: "phx.ocir.io/weblogick8s/quick-start-aux-image:v1"

domainHomeSourceType: FromModel

domainHome: /u01/domains/sample-domain1

image: "container-registry.oracle.com/middleware/weblogic:12.2.1.4"

serverStartPolicy: IfNeeded

serverPod:

resources:

requests:

cpu: "250m"

memory: "768Mi"

replicas: 1

clusters:

- name: sample-domain1-cluster-1

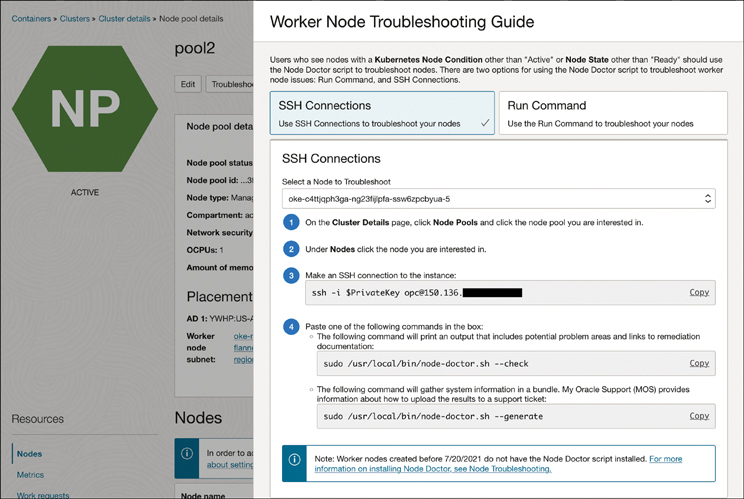
...TRUNCATED ...

In the partial example in [Listing 5-11](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#list5_11), the CRDs describe both a Cluster resource and a Domain resource. The Domain resource is a way to provide the WebLogic domain configuration, a WebLogic install, and other components and configurations to run the domain. A Cluster resource models a WebLogic cluster within a given WebLogic domain. Because WebLogic has its own notion of what a “cluster” is and how cluster operations such as scaling function, the Cluster resource bridges Kubernetes to the WebLogic notion of a cluster. This makes operations such as scaling a WebLogic cluster possible through Kubernetes tooling and integrates it with typical Kubernetes scaling processes such as the HorizontalPodAutoscaler. The Domain resource and Cluster resource do not replace the traditional WebLogic configuration files; instead, they cooperate with those files to describe the Kubernetes artifacts of the corresponding domain.

**Troubleshooting Nodes with Node Doctor**

As with most software systems, you will occasionally encounter issues with your cluster that you need to troubleshoot. With a managed Kubernetes service such as OKE, the control plane is fully managed by the cloud provider and the data plane is managed by the user. This means that most issues that require you to gather data and analyze will be related to the data plane. These issues can range from infrastructure and OS-level issues with the data plane nodes, to problems with components such as the kubelet, the CNI, or other system pods and DaemonSets that run on the data plane. Troubleshooting these components can usually be done with some common diagnostic commands; however, a deep knowledge of Kubernetes is usually required to do this. When using OKE, users have access to additional tooling provided by OCI, called Node Doctor. Node Doctor helps users gather diagnostic data, provides suggestions, and troubleshoots data plane–related issues without requiring users to have a deep knowledge of Kubernetes- or Linux-based systems. The tool can also create support bundles that users can provide to Oracle support, avoiding time-consuming back-and-forth communications as support engineers ask for diagnostic data.

Node Doctor is a script that is preinstalled on the data plane nodes for OKE clusters; it is available in the location /usr/local/bin/node-doctor.sh. You can run Node Doctor by executing the script after logging into the data plane node using SSH, or you can execute it using the OCI Run-Command feature. The OCI console also includes a useful step-by-step guide that walks you through the process, regardless of how you want to run the Node Doctor tool. The guide can be accessed by navigating to the Node Pool Details page and then clicking the **Troubleshoot Nodes** button. [Figure 5-14](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#ch05fig14) shows the guide that provides you with specific instructions for the selected cluster and node pool.



**Figure 5-14** Steps to Run Node Doctor Available from the Console

When you run Node Doctor, you can choose to use the --check flag to get a health report for the node, with a summary of the checks performed and the issues identified, along with suggested actions to resolve the issues (see [Listing 5-12](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#list5_12)).

**Listing 5-12** Getting a Health Report for a Node

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05_images.xhtml#f0215-01a)

$ sudo /usr/local/bin/node-doctor.sh --check

INFO: /usr/local/bin/oke-node-doctor.tar.gz already exists and MD5 match.

pip requires Python '>=3.7' but the running Python is 3.6.8

Running node doctor...

PASS node health...

PASS DNS lookup...

PASS kubelet cert rotation flag...

PASS kubelet logs...

PASS iscsi health...

PASS service health...

PASS instance metadata...

WARN cloud-init version...

SKIP cloud-init status...

SKIP chef onboard status...

PASS image and instance info...

Command line error: one of the following arguments is required: --save --add-repo

--dump --dump-variables --set-enabled --enable --set-disabled --disable

SKIP yum status...

PASS flannel status...

PASS coredns status...

PASS proxymux-client status...

PASS kube-proxy status...

PASS pods in ImagePullBackOff...

PASS pods failed mounting volume...

FAIL proxymux client registration status...

PASS runc version...

PASS pod usage...

PASS br\_netfilter module availability...

NODE DOCTOR REPORT

------------------

17/19 checks passed

2 Signal(s) generated

Signal 1: CLOUD\_INIT\_CUSTOMIZED

Description:

Instance user\_data is different from OKE native cloud init

Signal 2: PROXYMUX\_CLIENT\_REGISTRATION\_FAILURE

Description:

Proxymux client is not able to register with proxymux server

Resolution 1: CHECK\_VCN\_K8S\_ENDPOINT

Description: Network related failures have been detected. Please validate the

network settings. Most likely, port 12250 in the security list of the k8s api

endpoint VCN is misconfigured.

Useful links: ['https://docs.oracle.com/en-us/iaas/Content/ContEng/Concepts/

contengnetworkconfig.htm', 'https://docs.oracle.com/en-us/iaas/Content/ContEng/

Concepts/contengnetworkconfigexample.htm']

Related Signals:

PROXYMUX\_CLIENT\_REGISTRATION\_FAILURE: Proxymux client is not able to register with

proxymux server

Node doctor scan is complete. Report has been saved at /var/log/oke-node-doctor/

oke-node-doctor-2092328.log

Alternatively, if you are working with support, you can use the --generate flag to generate a TAR file that collects multiple log files, diagnostic command output, and other data that helps you or the support team quickly analyze the state of the node from multiple fronts. Contained in the data collected as a bundle is information and output from diagnostic commands, including VNIC details, iptables rules, and storage information. The tools try to limit the data collected, with a goal to keep the bundle’s size less than 10MB; however, this might mean that log files with critical data must be truncated. To prevent this, you can combine the --generate flag with the --large flag to avoid a size goal for the generated bundle or combine the --since and --until flags to restrict the data collected within a time boundary. [Listing 5-13](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#list5_13) shows Node Doctor usage for generating a support bundle and some of the logs and diagnostics that are included in the bundle.

**Listing 5-13** Generating a Support Bundle Using Node Doctor

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05_images.xhtml#f0218-01a)

[opc@oke-c4ttjqph3ga-ng23fijlpfa-ssw6zpcbyua-5 ~]$ sudo /usr/local/bin/node-

doctor.sh --generate --large

--- TRUNCATED ---

Generating node doctor bundle...

Generated /tmp/oke-support-bundle-2023-04-19T19-17-08.tar

[opc@oke-c4ttjqph3ga-ng23fijlpfa-ssw6zpcbyua-5 ~]$ tar -tf /tmp/oke-support-

bundle-2023-04-19T19-17-08.tar

--- TRUNCATED ---

home/opc/TEMP\_DIR/tmpypxxadi7/system/iptables\_filter

home/opc/TEMP\_DIR/tmpypxxadi7/system/iptables\_nat

home/opc/TEMP\_DIR/tmpypxxadi7/system/crictl\_pods

home/opc/TEMP\_DIR/tmpypxxadi7/system/crictl\_images

home/opc/TEMP\_DIR/tmpypxxadi7/logs/

home/opc/TEMP\_DIR/tmpypxxadi7/logs/kubelet.gz

home/opc/TEMP\_DIR/tmpypxxadi7/logs/containers/kube-flannel-ds-zb9zm\_kube-system\_

install-cni.log.gz

home/opc/TEMP\_DIR/tmpypxxadi7/logs/containers/proxymux-client-dgbcn\_kube-system\_

proxymux-client.log.gz

home/opc/TEMP\_DIR/tmpypxxadi7/logs/containers/csi-oci-node-sncjp\_kube-system\_csi-

node-driver.log.gz

--- TRUNCATED ---

**Configuring SR-IOV Interfaces for Pods on OKE Using Multus**

When highly network-oriented workloads require setting up secondary network interfaces within pods, you can use a meta CNI such as Multus[**6**](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#ref5_6a) to achieve this. The secondary network interfaces that are usually attached in these cases have specialized networking capabilities or properties, such as single root IO virtualization (SR-IOV).

SR-IOV is a specification that allows a single PCIe device to appear to be multiple separate physical PCIe devices. SR-IOV works by introducing the idea of physical functions (PFs) and virtual functions (VFs). A PF is used by the host and usually represents a single NIC port. VF is a lightweight version of that PF. With appropriate support, SR-IOV presents a way for the physical hardware (such as a SmartNIC) to present itself as several distinct (network interface) devices. With containers, you can then move one of these interfaces (a VF) from the host into the network namespace for a container or a pod so that the container can now directly access the interface. The advantage this offers is that you get none of the overhead with virt-io and you get native device performance.

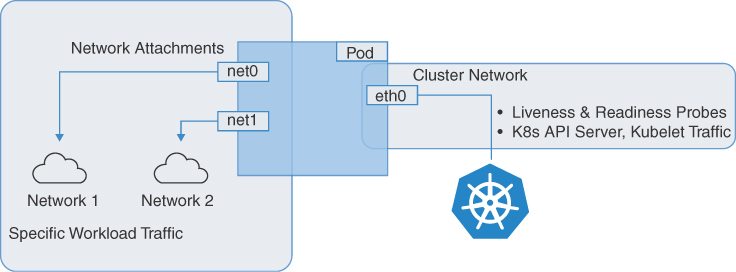
Significant differences exist between how the interfaces are created and managed when using bare metal nodes (you have full control over the hardware) and VM-based nodes (a hypervisor abstracts your access to the underlying hardware and you do not have as much control over it). Specifically, when using VMs, you do not typically have access to the PF. In both cases, however, Multus is used to provide additional network interfaces to pods. The sections that follow look at how these secondary network interfaces are created and examine the different plug-ins used to manage them.

**Using Bare Metal Nodes**

When running on bare metal nodes, you can leverage the SR-IOV CNI plug-in to manage SR-IOV virtual functions as resources that can be allocated on a node and use the Multus meta CNI to add network interfaces to pods. The approach has several layers and components. At its crux, a Kubernetes device plug-in manages a set of virtual functions and publishes it as an allocatable resource on the node. When a pod requests such a resource, the pod can be assigned to a node where the resource is available and an SR-IOV CNI can plumb the virtual function into the pod’s network namespace. A CNI meta plug-in such as Multus handles multiple network attachments to the pod so that the pod can communicate over both the SR-IOV and the overlay networks.

You first set up a number of VFs on the SR-IOV-capable smartNICs, which then present themselves as individual NICs. You then configure these VFs with MAC addresses that OCI recognizes. These VFs are created outside Multus, either manually (as described in this tutorial) or using a script that can be invoked at node creation time. At this point, you have a pool of VFs, each identified by the host as a separate NIC, and an OCI MAC address. The Kubernetes network plumbing working group maintains a special-purpose network device plug-in that discovers and publishes VFs as allocatable node resources. The SR-IOV CNI (also from the Kubernetes network plumbing working group) works alongside the device plug-in and manages the assignment of these virtual functions to the pod based on the pod lifecycle.

Now you have one or more nodes with a pool of VFs that are recognized and managed by the SR-IOV device plug-in as allocatable node resources. These can be requested by pods. The SR-IOV CNI plumbs (moves) the VF into the pod’s network namespace upon pod creation and releases the VF (moves it back to the root namespace) upon pod deletion. This makes the VF available to be allocated for another pod. A meta plug-in such as Multus can provide the VF information to the CNI and manage multiple network attachments on the pod. [Figure 5-15](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#ch05fig15) illustrates a pod with three network interfaces attached, with the first interface eth0 being used for liveness and readiness probes, as well as kubelet and Kubernetes API server communications. The two other interfaces are connected to two separate networks, and the workload is assumed to be able to make use of these interfaces to communicate with them individually.



**Figure 5-15** Pod with Multiple Network Interfaces Providing Connectivity to Multiple Networks

The first task is to set up the bare metal hosts by creating the VFs on the PCIe device. This can be done by setting the number of desired VFs in /sys/class/net/${PHYSICAL\_DEVICE}/device/sriov\_numvfs. The ${PHYSICAL\_DEVICE} can be identified by running ip addr show and looking for the primary interface. These steps can be condensed into a simple script, as provided in [Listing 5-14](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#list5_14).

**Listing 5-14** Script to Create and Verify the VFs

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05_images.xhtml#f0220-01a)

# Gets the physical device. Alternatively, just run 'ip addr show' and look at the

primary iface to set $PHYSDEV

URL=http://169.254.169.254/opc/v1/vnics/

baseAddress='curl -s ${URL} | jq -r '.[0] | .privateIp''

PHYSDEV='ip -o -4 addr show | grep ${baseAddress} | awk -F: '{gsub(/^[ \t]|[

\t]$/,"",$2);split($2,out,/[ \t]+/);print out[1]}''

# Add two VFs

echo "2" > /sys/class/net/${PHYSDEV}/device/sriov\_numvfs

# Verify the VFs

ip link show ${PHYSDEV}

Next, you need to assign OCI MAC addresses to the VFs. These VFs that were just created have autogenerated MAC addresses to begin with (or 000). For the traffic from these VFs to be permissible on the OCI network, you need to set MAC addresses that OCI provides. These can be generated by creating VNIC attachments or using the API for it. From the OCI console, create the same number of VNIC attachments on the host as the number of VFs created. Note the MAC addresses of each VNIC attachment. Now these MAC addresses that are recognized by OCI can be assigned to each of these VFs that was created earlier. This completes the host setup. At this point, you have a bare metal instance with multiple VFs that have OCI-generated MAC addresses, as confirmed in [Listing 5-15](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#list5_15).

**Listing 5-15** Setting the MAC Addresses and Assigning Them to the Previously Created VFs

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05_images.xhtml#f0220-02a)

'''

# For each MAC address from the VNIC attachments

ip link set ${PHYSDEV} vf <n= 0..numVFs> mac <MAC Address from VNIC attachment>

spoofchk off

# verify all VFs have Mac addresses from OCI

ip link show ${PHYSDEV}

'''

With the host setup completed, the next step is to install the SR-IOV CNI and device plug-in. The SR-IOV CNI enables the configuration and use of SR-IOV VF networks from within pods; the device plug-in discovers and advertises the SR-IOV capable network devices on the node. This makes these SR-IOV devices allocatable resources on the node that pods can request, just as a pod requests CPU and memory.

This SR-IOV CNI can be installed on any Kubernetes cluster that is running Kubernetes version 1.16 or later. The CNI runs on the cluster as a daemon set. Because it is common for only some nodes in the cluster to have SR-IOV, nodes without SR-IOV devices are handled gracefully by the device plug-in itself:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05_images.xhtml#f221-01a)

git clone https://github.com/k8snetworkplumbingwg/sriov-cni.git && cd

sriov-cni

kubectl apply -f images/k8s-v1.16/sriov-cni-daemonset.yaml && cd..

The primary purpose of the device plug-in is to discover, advertise, and track the usage of the SR-IOV-capable network devices on the node, so a configuration (expressed as a ConfigMap) is required to enable it to create the device plug-in endpoints. The configuration is specific to the NIC hardware and identifies the devices and the drivers used. To create this configuration, you need to know the vendor ID, device ID, and driver used by the device. This can be done with standard tools such as lspci. [Listing 5-16](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#list5_16) shows how to find the vendor ID and device ID.

**Listing 5-16** How to Find the Vendor and Device IDs

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05_images.xhtml#f221-02a)

lspci -nn|grep Virtual

31:02.0 Ethernet controller [0200]: Broadcom Inc. and subsidiaries NetXtreme-E

Ethernet Virtual Function [14e4:16dc]

31:02.1 Ethernet controller [0200]: Broadcom Inc. and subsidiaries NetXtreme-E

Ethernet Virtual Function [14e4:16dc]

The example in [Listing 5-16](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#list5_16) shows two VFs because we filtered the output of lspci with the keyword Virtual. lspci reads through the sysfs entries and presents the information in an easy-to-understand (and easy-to-parse) manner. In the example here, 31:02.0 represents the bus number (31), device number (02), and function (0). lspci uses libpci, which uses the PCI identification data in /usr/share/hwdata/pci.ids to decode information such as vendor and device numbers; it then uses that information here to identify the device class (0200) as an Ethernet controller and present the vendor information (Broadcom). The last bit of information provides the vendor ID (14e4) and device ID (16dc). You can cross-check this with the hwdata that lspci uses. The output shows that it is indeed an Ethernet virtual function from the vendor:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05_images.xhtml#f221-03a)

cat /usr/share/hwdata/pci.ids|grep 16dc

16dc NetXtreme-E Ethernet Virtual Function

With the device ID and vendor ID identified, you now need to find the drivers used. You can do this by searching /sys for the driver user on the PCI bus number, device number, and function. In the example here, the PCI bus, device, and function can be seen from the previous lspci output: 31:02.0. Searching sysfs for this device reveals the driver name to be bnxt\_en, as shown in [Listing 5-17](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#list5_17).

**Listing 5-17** How to Find the Drivers Used

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05_images.xhtml#f222-01a)

# filtering based on the PCIe slots.

Find /sys | grep drivers.\*31:02.0|awk -F/ '{print $6}'

bnxt\_en

Now you have the information to set up the configuration for the SR-IOV device plug-in. Create a configMap; it should be named sriovdp-config and should have a key config.json. [Listing 5-18](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#list5_18) shows an example of how this configMap should look.

**Listing 5-18** Example configMap

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05_images.xhtml#f222-02a)

cat << EOF > sriovdp-config.yaml

apiVersion: v1

kind: ConfigMap

metadata:

name: sriovdp-config

namespace: kube-system

data:

config.json: |

{

"resourceList": [

{

"resourceName": "netxtreme\_sriov\_rdma",

"resourcePrefix": "broadcom.com",

"selectors": {

"vendors": ["14e4"],

"devices": ["16dc"],

"drivers": ["bnxt\_en"],

"isRdma": false

}

}

]

}

EOF

kubectl create -f sriovdp-config.yaml

This configuration lets the device plug-in look for PCIe devices that match the selectors in the configuration and advertise that the node has this type of resource. Pods then can request these by the resourceName in the configuration. An example of how a pod requests these resources is presented later in the chapter, in [Listing 5-21](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#list5_21).

With the config map created, the device plug-in can be installed as a DaemonSet:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05_images.xhtml#f0223-01a)

git clone https://github.com/k8snetworkplumbingwg/sriov-network-device-plugin.git

&& cd sriov-network-device-plugin

kubectl create -f deployments/k8s-v1.16/sriovdp-daemonset.yaml && cd ..

With the DaemonSets deployed, you can check the container logs for troubleshooting. After a successful deployment, the node should list the virtual functions as allocatable resources. In this example, because you created two VFs and configured them with resourceName: netxtreme\_sriov\_rdma and resourcePrefix: broadcom.com, you see that the node now has two of these broadcom.com/netxtreme\_sriov\_rdma resources that can be requested by pods, along with the CPU, memory, and other resources on the node (see [Listing 5-19](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#list5_19)).

**Listing 5-19** Output of Running kubectl get nodes to See VFs as Allocatable Node Resources

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05_images.xhtml#f0223-02a)

'''

kubectl get node <node\_name> -o json | jq '.status.allocatable'

{

"broadcom.com/netxtreme\_sriov\_rdma": "2",

"cpu": "128",

"ephemeral-storage": "37070025462",

"hugepages-1Gi": "0",

"hugepages-2Mi": "0",

"memory": "527632840Ki",

"pods": "110"

}

'''

With the SR-IOV CNI and device plug-in set up, pods can now request these resources. However, you still need a way to plumb multiple network interfaces to a pod. This can be done by Multus, so the next task is to install Multus.

Multus is a meta plug-in that can chain multiple CNI plug-ins such as the SR-IOV CNI plug-in and the Flannel CNI plug-in, to support “multi-homed” pods or pods with multiple network interfaces. Installing Multus is done by simply applying the Multus DaemonSet:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05_images.xhtml#f0223-03a)

git clone https://github.com/k8snetworkplumbingwg/multus-cni.git && cd multus-cni

kubectl apply -f images/multus-daemonset.yml && cd ..

To attach additional interfaces to the pods, you need a configuration for the interface to be attached. This is encapsulated in the custom resource of kind: NetworkAttachmentDefinition. This CRD is created when Multus is installed. This configuration is essentially a CNI configuration packaged as a custom resource. [Listing 5-20](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#list5_20) shows a NetworkAttachmentDefinition that uses the VFs created earlier.

**Listing 5-20** Example Manifest Used to Create a NetworkAttachmentDefinition Custom Resource

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05_images.xhtml#f0224-01a)

cat << EOF > sriov-net1.yaml

apiVersion: k8s.cni.cncf.io/v1

kind: NetworkAttachmentDefinition

metadata:

name: sriov-net1

annotations:

k8s.v1.cni.cncf.io/resourceName: broadcom.com/netxtreme\_sriov\_rdma

spec:

config: '{

"type": "sriov",

"cniVersion": "0.3.1",

"name": "sriov-network",

"ipam": {

"type": "host-local",

"subnet": "10.20.30.0/25",

"routes": [{

"dst": "0.0.0.0/0"

}],

"gateway": "10.20.10.1"

}

}'

EOF

kubectl apply -f sriov-net1.yaml

Pods can now request additional interfaces using an annotation and resource request. The resource request helps the scheduler assign the pod based on VF availability on nodes, and the annotation lets the meta plug-in (Multus) know which NetworkAttachmentDefinition (CNI Config) to use. [Listing 5-21](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#list5_21) shows an example with a test pod.

**Listing 5-21** Example Manifest Used to Create Pods That Will Be Scheduled Based on the Availability of VFs on Nodes

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05_images.xhtml#f0224-02a)

## Create the first pod

cat << EOF | kubectl create -f -

apiVersion: v1

kind: Pod

metadata:

name: testpod1

annotations:

k8s.v1.cni.cncf.io/networks: sriov-net1

spec:

containers:

- name: appcntr1

image: centos/tools

imagePullPolicy: IfNotPresent

command: [ "/bin/bash", "-c", "--" ]

args: [ "while true; do sleep 300000; done;" ]

resources:

requests:

broadcom.com/netxtreme\_sriov\_rdma: '1'

limits:

broadcom.com/netxtreme\_sriov\_rdma: '1'

EOF

## Create a second pod

cat << EOF | kubectl create -f -

apiVersion: v1

kind: Pod

metadata:

name: testpod2

annotations:

k8s.v1.cni.cncf.io/networks: sriov-net1

spec:

containers:

- name: appcntr1

image: centos/tools

imagePullPolicy: IfNotPresent

command: [ "/bin/bash", "-c", "--" ]

args: [ "while true; do sleep 300000; done;" ]

resources:

requests:

broadcom.com/netxtreme\_sriov\_rdma: '1'

limits:

broadcom.com/netxtreme\_sriov\_rdma: '1'

EOF

With two pods created, you should be able to see that they are both running. Each pod is annotated with the k8s.v1.cni.cncf.io/networks: sriov-net1 annotation, which tells Multus that this pod needs to be attached to the network whose configuration is defined in the NetworkAttachmentDefinition named sriov-net1. Of course, this is the CNI configuration, and it establishes the default routes and IP address management. Additionally, the pod is making a resource request for the resource broadcom.com/netxtreme\_sriov\_rdma and is requesting a count of 1 of these resources. This effectively tells the Kubernetes scheduler that this pod needs to be allocated to a node that has at least one of these VFs available. When the network attachment has been made, the device plug-in updates the node and decrements the number of available VFs by one. When this pod is terminated, the VF that it has been using is released, and the device plug-in updates the node and increments the number of available VFs by one.

After the pods have been deployed, you can check that they both have multiple interfaces. You also can check the communication between the pods over the SR-IOV devices, as demonstrated in [Listing 5-22](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#list5_22).

**Listing 5-22** Verifying That the Deployed Pods Have Multiple Interfaces and Can Communicate over SR-IOV

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05_images.xhtml#f0226-01a)

## Verify that both pods have two interfaces. An 'eth0' on the overlay and a

## 'net1' which is the VF.

kubectl exec -it testpod1 -- ip addr show

kubectl exec -it testpod2 -- ip addr show

## Checkout the routes

kubectl exec -it testpod1 -- route -n

kubectl exec -it testpod2 -- route -n

## test communication

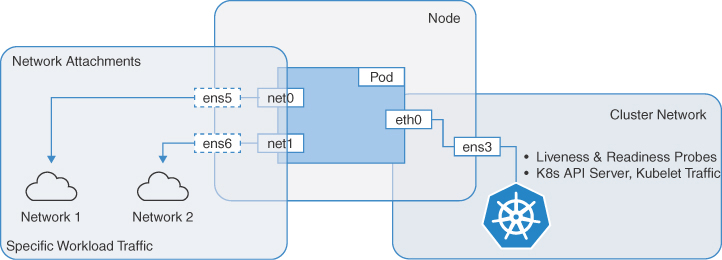
kubectl exec -it testpod1 -- ping <IP for net1 on pod2>

**Using Virtual Machine Nodes**

Significant differences exist in how the interfaces are created and managed between bare metal and virtual machines. On a VM, you do not have access to the physical functions (PFs) on a PCIe device, so you must instead use the cloud provider APIs to interact with the PCIe device in order to create and manage the SR-IOV virtual functions (VFs) on them.

On VMs, you still use Multus to provide multiple interfaces to a pod; however, the SR-IOV CNI and the associated device plug-in are not used. This is because the SR-IOV CNI requires direct access to the underlying hardware. To overcome this challenge, you can use the OCI networking APIs for VNICs, to create a VF on the PF as in the bare metal scenario and give the VM direct and unobstructed access to this VF. These VFs now can be attached to a compute instance, including OKE nodes, as network interfaces. These interfaces/VFs can be moved to the network namespaces for pods, which allow the pod to use the VF directly and exclusively as a network interface. From the perspective of the pods, they are not able to distinguish between the two and, in both cases, have access to a VF that they can directly use.

To give a VM direct access to a VF, you need to launch the VM with the VFIO network attachment mode instead of the default *paravirtualized* mode. This choice is controlled by the launch mode for the compute instance. When the network attachment mode is set as VFIO, you can create network attachments using the OCI APIs, which creates VFs on the underlying PF and provides the VF directly to the VM. The OS on the host recognizes these as network interfaces. When the VF is available to the VM, it can be moved to the pod namespace. In this model, the VFs are created using OCI APIs instead of system commands in the bare-metal scenario. [Figure 5-16](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#ch05fig16) shows the VFs (ens5 and ens6) being moved into the pod namespace as net0 and net1.



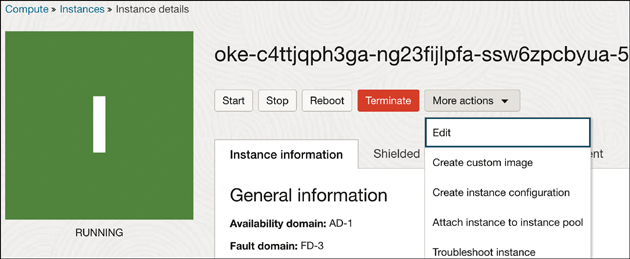
**Figure 5-16** A Pod with Multiple Network Interfaces—Additional Interfaces Are Made Available to the Pod by Moving Them from the Host’s Network Namespace to the Pod’s Network Namespace

The first task in setting up SR-IOV-based secondary interfaces for pods is to prepare the nodes in a manner similar to the bare metal servers. In the case of VMs, each node that requires access to SR-IOV interfaces must be prepared for hardware-assisted network attachments before it can be used by pods. This is done by editing the nodes and updating their network attachment type.

**Note**

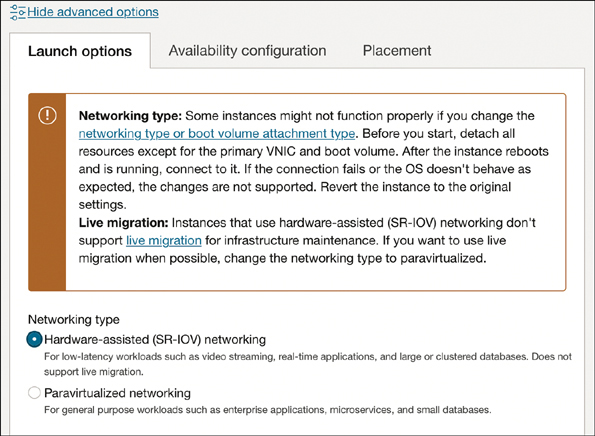
The method described here can be performed through the OCI console, which makes it easier to comprehend; however, this method is limited to clusters that operate using the Flannel CNI. The goal of updating the launch options is to essentially launch these nodes in the hardware-assisted (SR-IOV) mode, which creates a VF on the underlying PF and provides that to the VM when a network attachment is made. An alternate way to accomplish this, which also is applicable for clusters using the OCI native CNI, is to create a custom image based on the standard OKE image; edit the image capabilities and set the image to launch instances using the hardware-assisted (SR-IOV) mode; and then use this custom image for your node pools.

To update the nodes to use the hardware-assisted (SR-IOV) mode, edit the instance properties of the node as shown in [Figure 5-17](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#ch05fig17).



**Figure 5-17** How to Edit the Properties of an Existing Compute Instance

On the instance properties, click **Show Advanced Options** to view the additional properties. On the **Launch Options** tab, choose **Hardware-Assisted (SR-IOV) Networking** for the networking type, as illustrated in [Figure 5-18](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#ch05fig18).

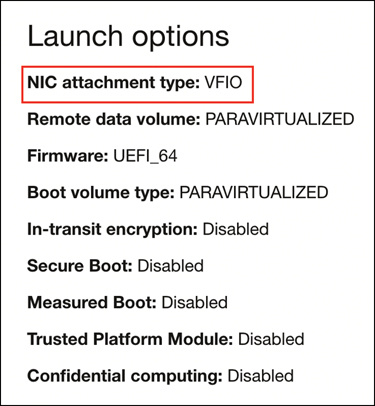


**Figure 5-18** Updating the Networking Type of an Instance to Use Hardware-Assisted (SR-IOV) Networking

**Note**

After an instance has been switched from *paravirtualized* network attachment to hardware-assisted (SR-IOV or VFIO) mode, it is no longer eligible for live migration for infrastructure maintenance.

The update workflow prompts you to reboot the instance. After the reboot, the instance has VFIO network attachments. This can be verified on the console, as [Figure 5-19](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#ch05fig19) illustrates.



**Figure 5-19** Current Launch Options for the Instance, Including the VFIO Network Attachment

Additionally, you can verify that your instances are using SR-IOV network attachments to connect to a node using SSH and use lspci to list the PCI devices on the VM. You should be able to see the underlying virtual function directly on the VM instead of a device using a virtio driver (such as the storage controller in [Listing 5-23](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#list5_23).

**Listing 5-23** Using lspci to Verify That the Virtual Function Is Directly Visible from the VM

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05_images.xhtml#f0229-01a)

$ lspci

00:00.0 Host bridge: Intel Corporation 440FX - 82441FX PM [Natoma] (rev 02)

00:01.0 ISA bridge: Intel Corporation 82371SB PIIX3 ISA [Natoma/Triton II]

00:01.1 IDE interface: Intel Corporation 82371SB PIIX3 IDE [Natoma/Triton II]

00:01.2 USB controller: Intel Corporation 82371SB PIIX3 USB [Natoma/Triton II]

(rev 01)

C00:01.3 Bridge: Intel Corporation 82371AB/EB/MB PIIX4 ACPI (rev 03)

00:02.0 VGA compatible controller: Device 1234:1111 (rev 02)

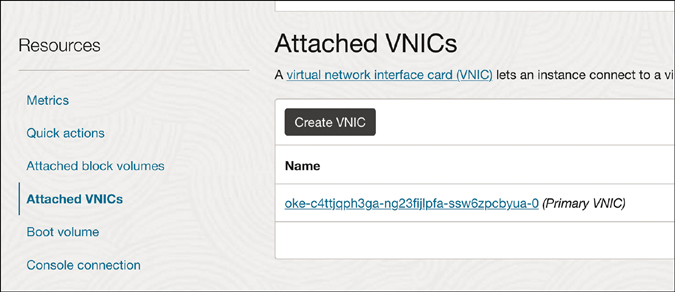
00:03.0. Ethernet controller: Mellanox Technologies MT28800 Family [Connect-5 Ex

Virtual Function]

00:04.0 SCSI storage controller: Red Hat, Inc. Virtio SCSI

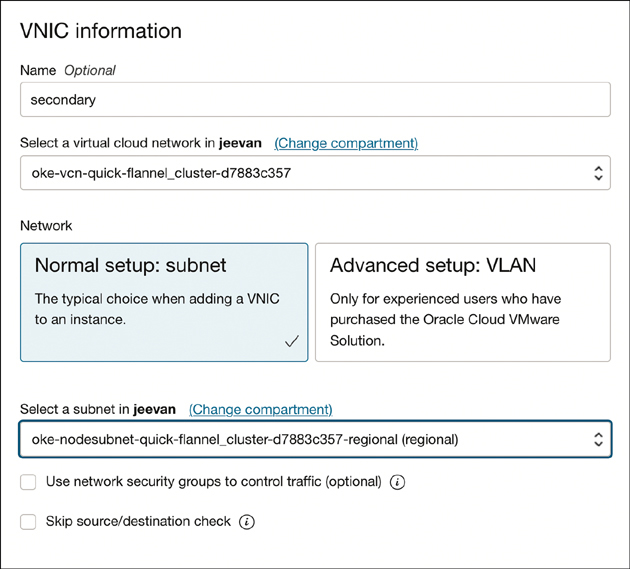
At this point, the node has a single VNIC attachment, which is the primary VNIC used for all communications to the node. Because the instance is using hardware-assisted network attachments, the network attachment is visible to the node as a virtual function on the underlying hardware. For pods to have exclusive use of a virtual function (VF), you need additional VFs on the VM. This can be provided using the console or API to add VNIC attachments to the instance. These VNIC attachments are VFs on the underlying PF. They can be verified with lspci.

To add VNIC attachments, from the instance page, choose **Attached VNICs** and click **Create VNIC**, as shown in [Figure 5-20](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#ch05fig20).



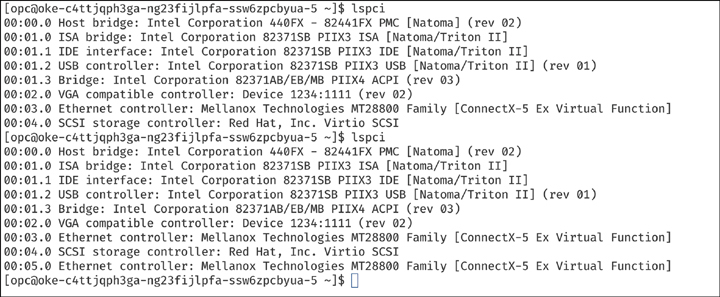
**Figure 5-20** Creating a VNIC in the Console

On this page, you can configure the VNIC using the VCN and subnet that is needed, as demonstrated in [Figure 5-21](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#ch05fig21).



**Figure 5-21** Selecting the VCN and Subnet for Use with Your VNIC

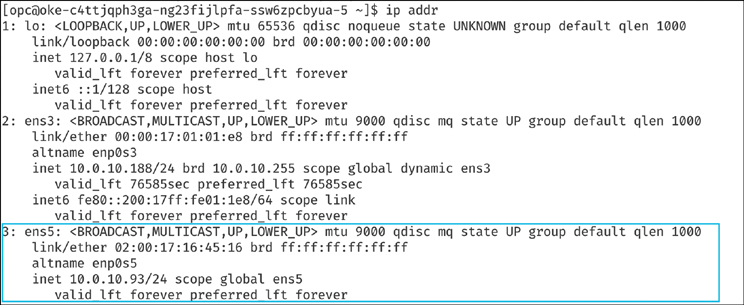
When this is configured, you should verify that the VNIC can be seen on the host as a virtual function (as before) by connecting to the node using SSH and running lspci, as demonstrated in [Figure 5-22](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#ch05fig22).



**Figure 5-22** Displaying the VNIC on the Host as a VF

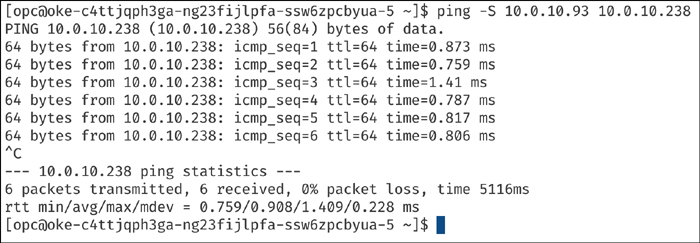
When you add a secondary VNIC to a Linux VM instance, a new interface (that is, an Ethernet device) is added to the instance and automatically recognized by the OS. However, DHCP is not active for the secondary VNIC, and you must configure the interface with the static IP address and default route. The next step is to configure the OS for secondary VNICs. OCI provides documentation and a script for configuring the OS for secondary VNICs. To configure the secondary VNIC, download the script on the node and run it based on the instructions provided in the OCI documentation.

After the script for configuring the secondary VNICs has been run, you should verify that the interface is now connected, with its IP address and default route. To check that this has been configured, use the command ip addr as shown in [Figure 5-23](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#ch05fig23) or a similar tool like nmcli.

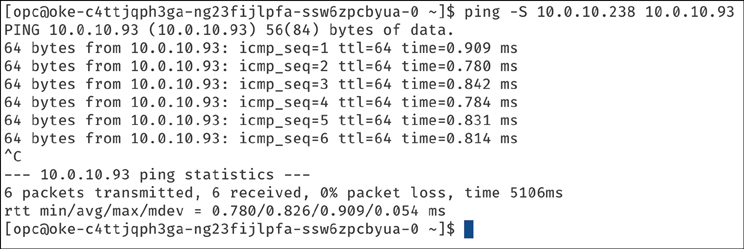


**Figure 5-23** Verifying Whether the Interface Is Now Connected

Optionally, it would be a good practice to verify the routing using a ping to reach the secondary IP addresses from each other. In [Figure 5-24](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#ch05fig24) and [Figure 5-25](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#ch05fig25), 10.0.10.238 is the secondary IP on a second node in the cluster. This completes the host setup for the nodes.



**Figure 5-24** Verifying Connectivity Between the Secondary VNICs



**Figure 5-25** Verifying Connectivity Between the Secondary VNICs from the Other Direction

With the host setup completed, you can now install Multus on the cluster. The installation of Multus follows the exact same steps as for the bare metal nodes. This is because Multus is just software that runs on the cluster and does not care about the node types.

With Multus installed on the cluster, you are ready to attach multiple interfaces to pods. To do this, you need a configuration for the interface to be attached, which is expressed as a NetworkAttachmentDefinition just as before, for bare metal nodes. This configuration is essentially a CNI configuration packaged as a custom resource. When using VMs, there is no access to the underlying hardware, and the VM is directly given access to one or more virtual functions on the physical NIC. The goal for the NetworkAttachmentDefinition is to provide an SR-IOV virtual function that has already been created for the exclusive use of a single pod so that the pod can take advantage of the capabilities without interference of any layers in between. To grant a pod exclusive access to the VF, you can leverage the host device plug-in that enables you to move the interface from the default or the root namespace into the pod’s namespace so that it has exclusive access to it.

The examples in [Listing 5-24](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#list5_24) show NetworkAttachmentDefinition objects that configure the secondary ens5 interface that was added to the nodes. The ipam plug-in configuration determines how IP addresses are managed for these interfaces. In this example, because you want to use the same IP addresses that were assigned to the secondary interfaces by OCI, you use the static ipam configuration with the appropriate routes. ipam configuration also supports other methods, such as host-local or dhcp, for more flexible configurations.

**Listing 5-24** Creating the Objects Used to Configure the Secondary ens5 Interface Added to the Nodes

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05_images.xhtml#f0233-01a)

## network attachment for the first node. Note the IPaddress assignment in the

## 'ipam' configuration.

cat << EOF | kubectl create -f -

apiVersion: "k8s.cni.cncf.io/v1"

kind: NetworkAttachmentDefinition

metadata:

name: sriov-vnic-1

spec:

config: '{

"cniVersion": "0.3.1",

"type": "host-device",

"device": "ens5",

"ipam": {

"type": "static",

"addresses": [

{

"address": "10.0.10.93/24",

"gateway": "0.0.0.0"

}

],

"routes": [

{ "dst": "10.0.10.0/24", "gw": "0.0.0.0" }

]

}

}'

EOF

## network attachment for the second node. Note the IPaddress assignment in the

## 'ipam' configuration.

cat << EOF | kubectl create -f -

apiVersion: "k8s.cni.cncf.io/v1"

kind: NetworkAttachmentDefinition

metadata:

name: sriov-vnic-2

spec:

config: '{

"cniVersion": "0.3.1",

"type": "host-device",

"device": "ens5",

"ipam": {

"type": "static",

"addresses": [

{

"address": "10.0.10.238/24",

"gateway": "0.0.0.0"

}

],

"routes": [

{ "dst": "10.0.10.0/24", "gw": "0.0.0.0" }

]

}

}'

EOF

With Multus configured with these additional network attachment definitions, pods can now request additional interfaces using an annotation. The annotation lets the meta plug-in (Multus) know what NetworkAttachmentDefinition (CNI Config) to use to provide additional interfaces when the pod is created.

**Note**

When using a static configuration like the one shown in [Listing 5-24](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#list5_24), the pods need to have node affinity set so that the pod is scheduled on the node where the desired host device is available. This differs from the approach when using bare metal nodes: In that case, you can use the SR-IOV device plug-in that keeps track of the VFs that are available on the node.

[Listing 5-25](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#list5_25) shows an example with a test pod.

**Listing 5-25** Creating a Test Pod That Requests Additional Interfaces

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05_images.xhtml#f0234-01a)

## Create the first pod

cat << EOF | kubectl create -f -

apiVersion: v1

kind: Pod

metadata:

name: testpod1

annotations:

k8s.v1.cni.cncf.io/networks: sriov-vnic-1

spec:

containers:

- name: appcntr1

image: centos/tools

imagePullPolicy: IfNotPresent

command: [ "/bin/bash", "-c", "--" ]

args: [ "while true; do sleep 300000; done;" ]

EOF

## Create a second pod

cat << EOF | kubectl create -f -

apiVersion: v1

kind: Pod

metadata:

name: testpod2

annotations:

k8s.v1.cni.cncf.io/networks: sriov-vnic-2

spec:

containers:

- name: appcntr1

image: centos/tools

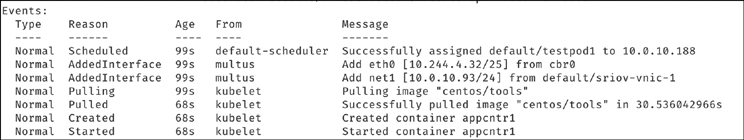
imagePullPolicy: IfNotPresent

command: [ "/bin/bash", "-c", "--" ]

args: [ "while true; do sleep 300000; done;" ]

EOF

With two pods created and in the running state, you should be able to see that additional network interfaces were created during the creation of the pods. Multus provides the eth0 interface that is backed by the default CNI (Flannel, in this example) and an additional net1 interface that is the SR-IOV virtual function. You can describe the pods and observe the Events section of the output to see the various events, including the interfaces attached to the pod (see [Figure 5-26](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#ch05fig26)).



**Figure 5-26** Events Associated with Creating Additional Network Interfaces When the Pods Were Started

After the pods have started, you can perform a quick test (see [Listing 5-26](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#list5_26)) to verify that the pods have multiple network interfaces attached to them.

**Listing 5-26** Verifying That the Pods Have Multiple Network Interfaces Attached to Them

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05_images.xhtml#f0236-01a)

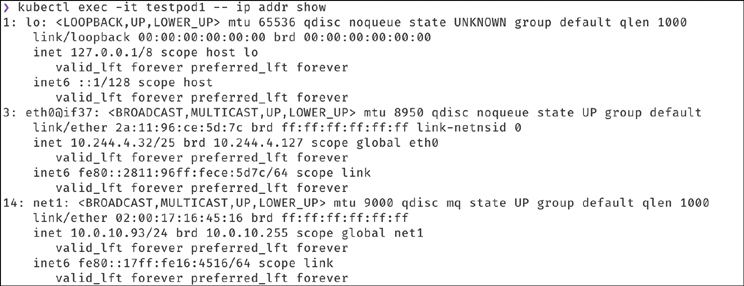
## Verify that both pods have two interfaces. An 'eth0' on the overlay and a

## 'net1' which is the VF, along with the IP address for the secondary VNIC.

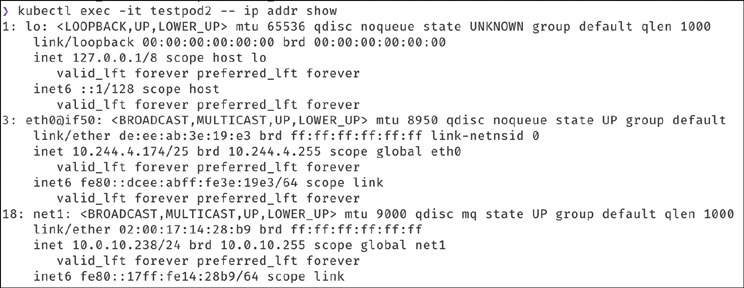
kubectl exec -it testpod1 -- ip addr show

kubectl exec -it testpod2 -- ip addr show

The output should be similar to [Figure 5-27](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#ch05fig27) and [Figure 5-28](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#ch05fig28).



**Figure 5-27** Output for the First Pod of the Test to Verify That Pods Have Multiple Network Interfaces Attached to Them



**Figure 5-28** Output for the Second Pod of the Test to Verify That Pods Have Multiple Network Interfaces Attached to Them

After you have verified that the pods have the SR-IOV interfaces attached to them in addition to the primary interface, you can verify the communication between the two pods over these secondary (SR-IOV) interfaces using the commands in [Listing 5-27](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#list5_27).

**Listing 5-27** Testing Connectivity Between the Two Pods over the Secondary Interfaces

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05_images.xhtml#f0237-01a)

## test communication

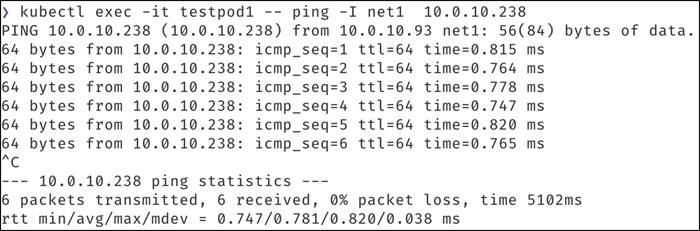
kubectl exec -it testpod1 -- ping -I net1 <ip address for secondary vnic on the

other pod/node>

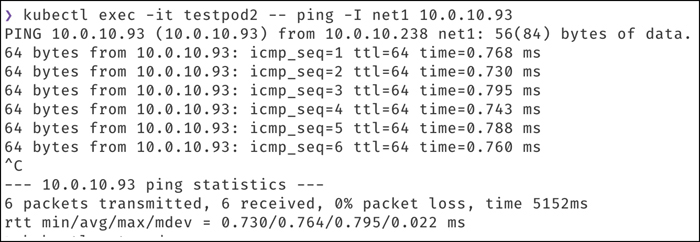
kubectl exec -it testpod2 -- ping -I net1 <ip address for secondary vnic on the

other pod/node>

The output should be similar to [Figure 5-29](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#ch05fig29) and [Figure 5-30](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#ch05fig30).



**Figure 5-29** Output of the Connectivity Test for the First Pod



**Figure 5-30** Output of the Connectivity Test for the Second Pod

Optionally, you can validate that the pods are routable using their network attachments by trying to reach them from the VMs or any other source within the VCN using the commands in [Listing 5-28](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#list5_28).

**Listing 5-28** Verifying Whether the Pods Are Routable from Another Source Within the VCN, Such as a VM

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05_images.xhtml#f0238-01a)

## Test that the pod is routable from outside Kubernetes. This is executed from

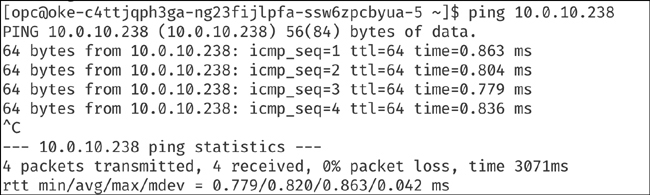
node1.

ping 10.0.10.238

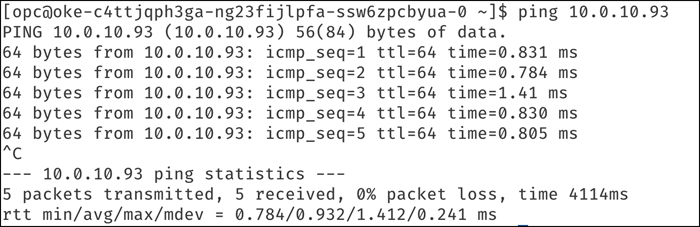
## similarly, from node 2

ping 10.0.10.93

The output should resemble [Figure 5-31](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#ch05fig31) and [Figure 5-32](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#ch05fig32).



**Figure 5-31** Output of the Routability Test for the First Pod



**Figure 5-32** Output of the Routability Test for the Second Pod

**Summary**

This chapter examined what it is like to own and operate a Kubernetes cluster with Container Engine for Kubernetes. The chapter started with the most common and most frequent task that Kubernetes operators perform: upgrades. You examined upgrades for your control plane and looked at several strategies for upgrading your data plane or nodes. Then you delved into strategies for scaling a cluster to meet the demands of workloads—after all, workloads and their characteristics change. The chapter discussed various autoscaling mechanisms, including the HorizontalPodAutoscaler (HPA), the Vertical Pod Autoscaler (VPA,) and the Kubernetes Cluster Autoscaler, which scales the cluster infrastructure. You also looked at some autoscaler best practices and the interactions between the HPA and the Cluster Autoscaler, which are commonly used together to orchestrate and automate cluster scaling operations and to optimize cost. Additionally, you examined other day 2 operational concerns, such as service account authentication, client token generation, and DNS configuration both within the cluster and using ExternalDNS.

The chapter moved on to look at extending cluster functionality with cluster add-ons and did a deep dive on setting up a monitoring stack based on Prometheus and Grafana. You also looked at operators for Kubernetes, including the operators for OCI, Oracle Database, and Oracle WebLogic. A critical aspect of managing your own cluster is troubleshooting it when you encounter issues. You saw how Node Doctor from OCI helps you diagnose and troubleshoot issues with your cluster nodes.

Finally, you looked at an advanced configuration that showcases several techniques for cluster configuration and customization by deploying a meta CNI and additional device plug-ins on both bare metal and virtual machine nodes, to grant additional networking capabilities for your pods.

**References**

[1](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#ref5_1) Setting up a NodeLocal DNS cache: <https://kubernetes.io/docs/tasks/administer-cluster/nodelocaldns/#configuration>

[2](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#ref5_2) Setting up ExternalDNS for Oracle Cloud Infrastructure (OCI): <https://github.com/kubernetes-sigs/external-dns/blob/master/docs/tutorials/oracle.md>

[3](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#ref5_3) OKE Operator Framework page: <https://sdk.operatorframework.io/docs/installation/>

[4](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#ref5_4) OCI Service Operator for Kubernetes (OSOK): <https://github.com/oracle/oci-service-operator>

[5](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#ref5_5) Installing the Operator SDK: <https://github.com/oracle/oci-service-operator/blob/main/docs/installation.md#install-operator-sdk>

[6](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#ref5_6) Multus CNI: <https://github.com/k8snetworkplumbingwg/multus-cni>