***Chapter 7*: Building a Cloud-Native Application on Oracle Cloud Infrastructure**

Software development methodologies have changed a lot in the last two decades. Containers and microservices-based software development have taken the front seat. It has not only set foot in the startup world, but also expanded into large enterprises. Enterprises that were previously hesitant to look at this new model of building applications are now beginning to learn this and are changing their method of software delivery. Large IT enterprises are developing their own solutions for a hybrid model; that is, their solution not only caters to the on-premises software model, but is scaling to cloud-based models as well – they are being delivered as containers and being orchestrated using container orchestration services such as Kubernetes.

OCI provides **Oracle Container Engine for Kubernetes**, or **OKE** for short, which is a fully managed service that's highly scalable, highly available, and secure for deploying your microservices-based applications at scale. But this is not enough. OCI provides other methods so that we can have an environment where we can choose to be fully cloud native. In this chapter, we would show the use cases for cloud native applications and their evolution. We will also focus on why we need to adopt containers and container orchestration.

In this chapter, we're going to cover the following main topics:

* Evolution of cloud-native applications
* Storing application images on the OCI registry
* Deploying microservices on Oracle Container Engine for Kubernetes
* Exposing microservices using the OCI API gateway

**Evolution of cloud native applications**

Large enterprises are realizing the power of quickly innovating using their existing investments. To keep up this speed of innovation, they are moving on and performing cloud-native application development, where they can deliver their investments using built-in open source and open standards frameworks. But what are cloud-native applications? Cloud-native applications are built to provide consistent development, automation frameworks, and a seamless experience in a private, public, or hybrid cloud model. These applications achieve these benefits by providing self-serve, on-demand resource provisioning, and automated life cycle operations.

With this agility in mind, every cloud provider has their own methods of providing managed services, all while offering to help these large enterprise companies get started on this journey. Here, having a managed Kubernetes offering is the key to success.

Oracle knows that data is the lifeblood of cloud businesses. It ties them to their customers, their supply chain, their finances, and more. Oracle has built their cloud to support all the functionality and performance that their customers have had in their own data center, but with the benefits of increased agility and being able to eliminate mundane tasks such as managing hardware and facilities, upgrades, patches, and capacity forecasting. Oracle has deep expertise in this and provides cloud-specific automation to make migration possible, without any risks or high costs being involved. They offer tools to connect their cloud to their customer's data center, which makes migrating the workload easier. Everything OCI runs in their cloud is consistent with what customers run in their own data centers, including the Oracle database itself, the surrounding ecosystem of tools, such as **Real Application Cluster** (**RAC**), Data Guard, and Golden Gate, and all the third-party data and management tools their customers use.

Oracle has ensured that customers won't take a step backward in terms of performance when they move to the cloud. They have given customers the ability to run Exadata engineered systems as cloud services, thus offering the highest level of performance and scalability for Oracle workloads, something that is widely used in on-premises environments and not available on any other cloud. OCI has also built a cloud network with a massive interconnect bandwidth and no resource over-subscriptions to ensure that noisy neighbor problems aren't an issue, and also that the high performance they deliver doesn't vary based on external factors.

Once you get to the OCI, then the innovation behind this kicks into higher gear. Customers have a full range of options to deprecate and eliminate their data centers if they choose to, or to keep them running for some workloads, with deep compatibility and connectivity options from the Oracle Cloud. OCI allows customers to expand their curation of data with deep analytics and integration options. This helps them get onto Oracle's new autonomous database cloud services, which eliminate tedious management tasks and represent the future of enterprise data management. Customers can also expand the network of applications that are surrounding their data with cloud native functionality, which allows them to build new innovative approaches to managing and making use of data, including our Kubernetes-based container service, their flexible Oracle function capabilities, as well as a broad ecosystem of third-party options that unlock new value from data.

Let us show you the evolution of the application deployment in organizations during past 3 decades:

* During the 80's and the mid 90's, we saw that customers ran monolithic applications running on top of physical servers. They were developed using waterfall methodologies.
* During the end of the 90's until the millennium, customers started heavily using Unix/Linux operating systems, and we saw the emergence of virtualization (VMware and KVM). During this time, applications started using agile methodologies.
* Since 2010, we have seen a revolution and modernization of applications. Cloud computing has replaced a major chunk of data centers. This has led to applications being built around microservices that have been containerized, while developers and operations work toward an agile DevOps environment.

This is the modern application stack, which enables DevOps and microservices and runs on containers in the cloud. Thus, a new generation of applications are being created in the cloud and will only be running in the cloud. These are known as **cloud-native** applications.

Oracle has introduced a portfolio of products to address each of these building blocks. In this chapter, we will focus on containers. With the introduction of Oracle's managed container services, OCI is well positioned to support these modern approaches to software development and operations.

The following diagram shows the current software development and deployment model trend:

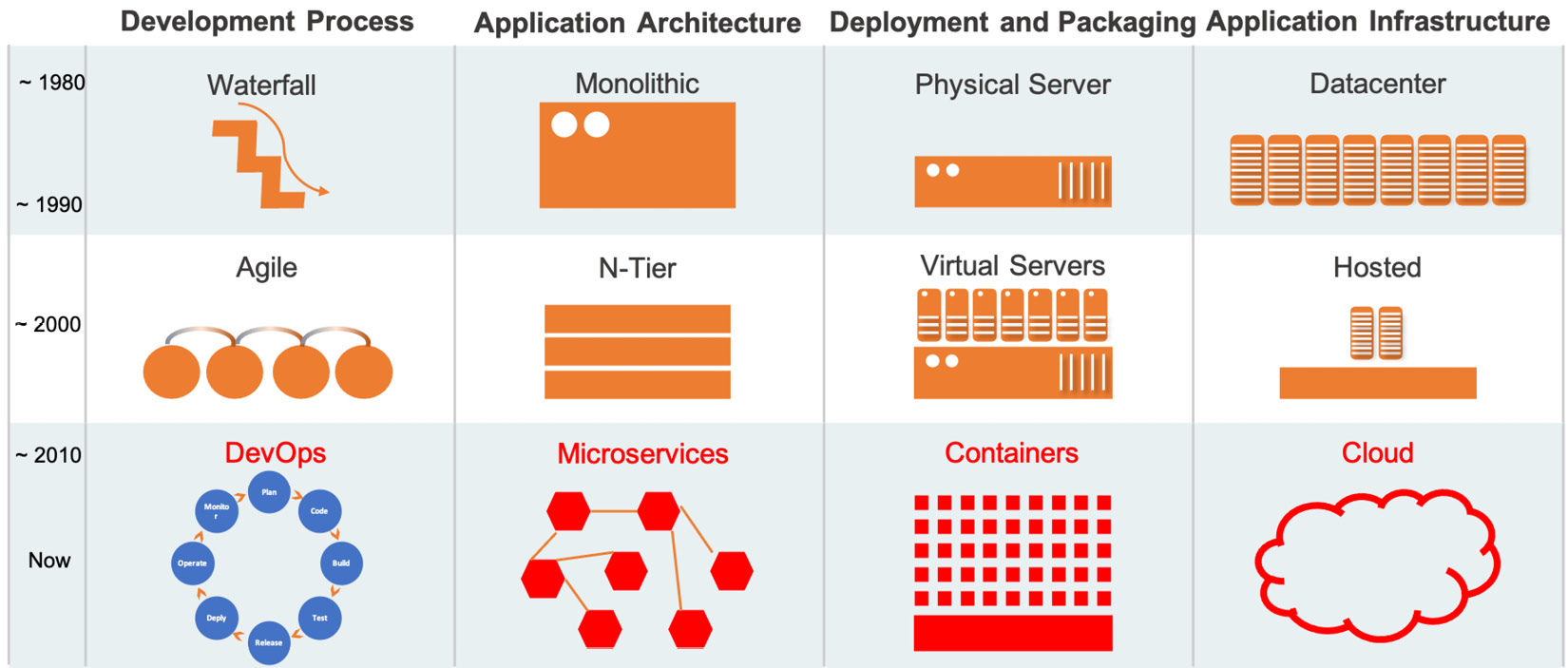


Figure 7.1 – History of software development model

There is a wide range of use cases that containers (we will be using Docker containers as a use case in this chapter) can be applied to, but as soon as their scale increases, orchestration becomes necessary. With Kubernetes, a Docker infrastructure can be made to scale and support much more advanced use cases.

Oracle's strategy for container-based services focuses on the leading technologies for containers and orchestration, which is Docker and Kubernetes. With these technologies, you can create applications at any scale, from simple DevOps setups to global mission-critical enterprise applications. Because the technologies are so widely used, they support a truly hybrid architecture, allowing you to run apps on-premises and in multiple clouds.

Let's look at each of these technology's use cases:

* Docker containers:

- Docker is easy for adopting containers and has good ecosystem tools that help build developer productivity.

- It is the most popular container runtime and supports the Open Container Initiative image format.

* Kubernetes orchestration:

- This is the most popular container orchestration platform. It has widespread adoption and is production-ready.

- Kubernetes is a complex container orchestration platform, but you can run cloud-scale applications on top of it.

- With the maturity that it has, it has a vast feature set, such as autoscaling, rolling upgrades, support for running stateful apps, and many others.

Cloud-native strategies are often pioneered in software development projects by developers who are convinced there must be a better way. Once the projects gain traction, automation is the next logical step. Teams will formalize the hand-offs and workflows to get predictable outcomes. At some point, these approaches become core to the companies that adopt them, and any disruption will significantly impact their business. They will focus on scale, performance, availability, security, access, and more to ensure that these approaches become part of the business infrastructure. This is where managed services from Oracle make adoption easier, more reliable, and more cost-effective.

The following diagram shows how Oracle managed to position their strategy and help companies get into the modern cloud-native application deployment phase:

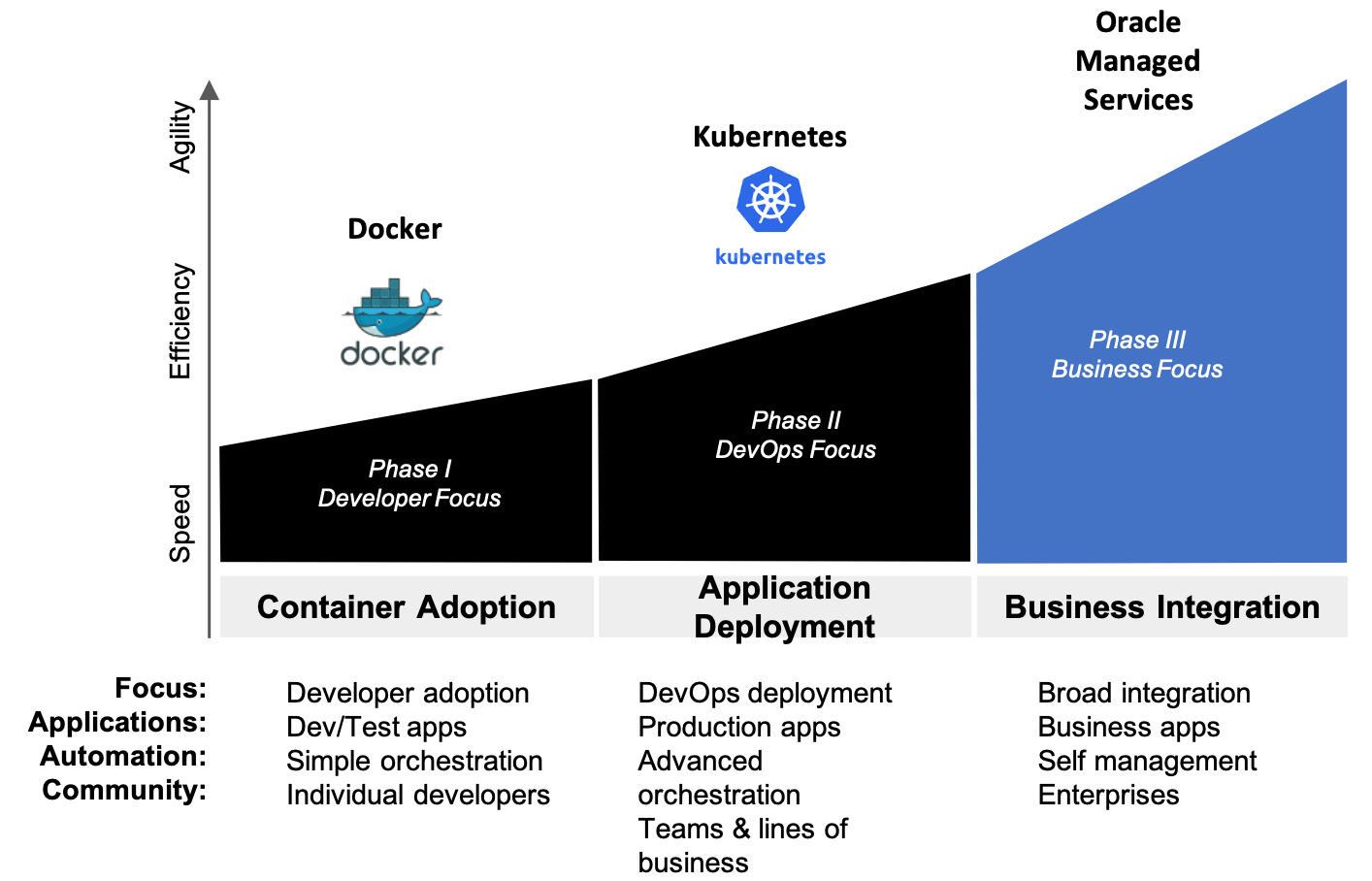


Figure 7.2 – Oracle's strategy to for cloud-native applications

Oracle has taken up a strategy for container-based infrastructure. Let's look at each of its pillars:

* **Complete cloud-native stack**: This strategy is used to help customers deliver tools and services that are complete, integrated, and open:

- Continuous integration and deployment, container registry, orchestration/scheduling, management/operations, analytics/introspection

- Provides an application development platform for serverless and microservices

* **Open source**: This strategy is used to help customers actively participate in community-driven, open source container technologies:

- Investing in Kubernetes, Docker, Fn, CNCF, DevTools, and DevOps by providing engineering resources, code contributions, and sponsorships

- Active support from Oracle's portfolio of open source assets (Java and so on)

* **Managed services**: This strategy is different in terms of the implementation's quality, services, and operational excellence:

- Full, transparent management.

- Open source compatible.

- Standards compliant.

- Deployed to Oracle Cloud Infrastructure.

- It is a purpose-built managed service for meeting enterprise-grade performance, security, HA, and governance.

But the key question here is, why should business leaders care about cloud-native application development? Well, as per the Cloud Native Computing Foundation at <https://www.cncf.io/wp-content/uploads/2020/08/CNCF-Webinar-_-Delivering-Cloud-Native-Application-and-Infrastructure-Management.pdf>:

*"Cloud-native technologies empower organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds, as well as containers, service meshes, microservices, immutable infrastructure, and declarative APIs".*

Cloud-native application development helps us independently invest and evolve features according to our business needs. It breaks monolithic applications into resilient, manageable, observable systems. It improves developer productivity, makes high-impact changes frequently at the same time open source technologies make solutions portable across clouds, and avoids lock-in. So, as you can see, you can use cloud-native technologies to break monolithic applications into small container-based distributed applications. This can be seen in the following diagram:

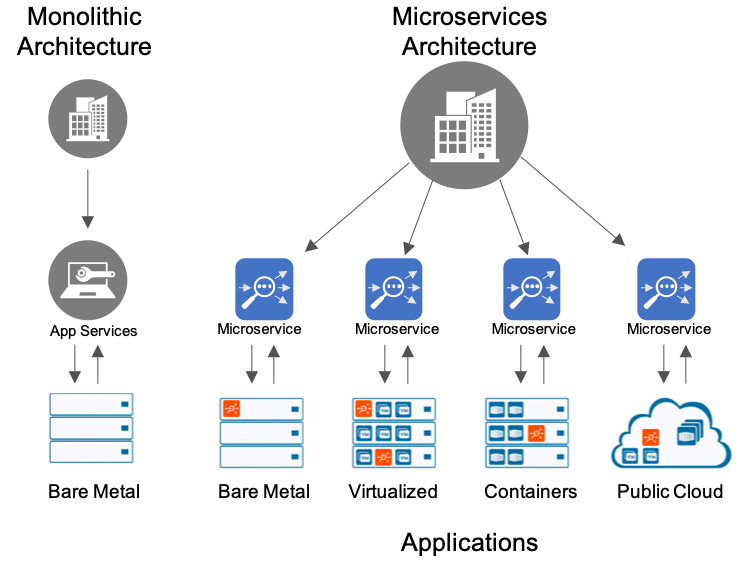


Figure 7.3 – Monolithic architecture for microservices

So far, you've learned about the evolution of OCI and how it provides services that help you with your journey of modern cloud-native application development and deployment. In the next section, we will talk about how OCI provides a free managed service for storing application container images.

**Storing application images on the OCI registry**

The OCI registry is a highly available Docker v2 container registry service that provides either a private or public service for storing and sharing container images. It is a regional service, which means you have different endpoints for different regions. It is a fully managed service and Oracle doesn't charge you for it. You can use the Docker CLI to push and pull images from the registry, and you can also use Kubernetes to download the images and run them on a fully managed Kubernetes environment.

Developers need to store these container images on a registry to maintain their state. You can use free and open source registries as well, but there are issues with doing this, such as access rights. The OCI registry is fully integrated with **OKE** and runs on the same OCI backend infrastructure. The following diagram shows these two service's integration and the customer's responsibilities:

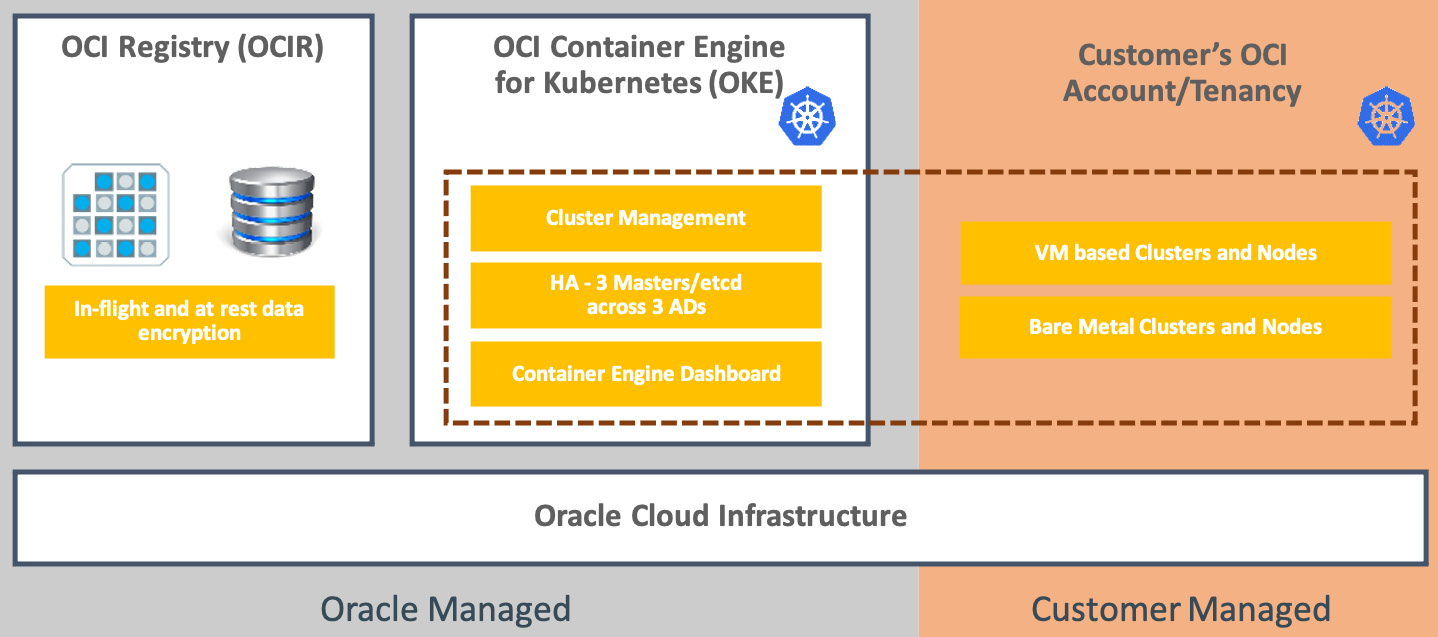


Figure 7.4 – OCIR and OKE integration

The gray shaded area designates the functions that Oracle manages for the customers, including an integrated registry and image storage and the managed Kubernetes. Users *do not* pay for any of Oracle's managed container infrastructure (the gray area). This is the control plane and enables you to configure these services and maintain operations, versions, their availability, and so on. The user pays regular fees for the compute, storage, and networking services that are used in the data plane (the customer managed area), which is where the applications run, data is stored, and so on.

Oracle manages the **etcd** and master nodes of the Kubernetes instance, in a high availability setup for the customer. Upgrading to new versions of Kubernetes will also be supported in the OKE dashboard, within the OCI console. The customer will manage the clusters/worker nodes that have been set up by the managed service for that instance in their own OCI account/tenancy. This is shown as **customer managed** in the preceding diagram.

Let's learn how to create the prerequisites and then create a repository before we push Docker images and pull them after that from the same repository.

**Preparing for pushing and pulling images from the registry**

To access the OCI registry, you need to know the URL endpoints for every region. As we've already mentioned, OCI registry is a regional service and an endpoint OS defined for each region. The following link states the OCIR endpoints for each region: <https://docs.oracle.com/en-us/iaas/Content/Registry/Concepts/registryprerequisites.htm>. This list is ever growing as Oracle is consistently adding regions at a rapid pace. So, it is always a good idea to check this portal to see the latest available region and its corresponding OCIR link.

Secondly, you need to generate an auth token so that you can authenticate yourself and access the repository. Follow these steps to generate an auth token and keep it somewhere safe. We will use it to authenticate from Cloud Shell later on:

1. Sign into the Oracle Cloud Infrastructure console.
2. Open the **Profile** menu and click on **User Settings**.
3. Click on **Auth Tokens**.
4. Click on **Generate Token**.
5. Provide a description and click on **Generate Token**, as shown in the following screenshot:

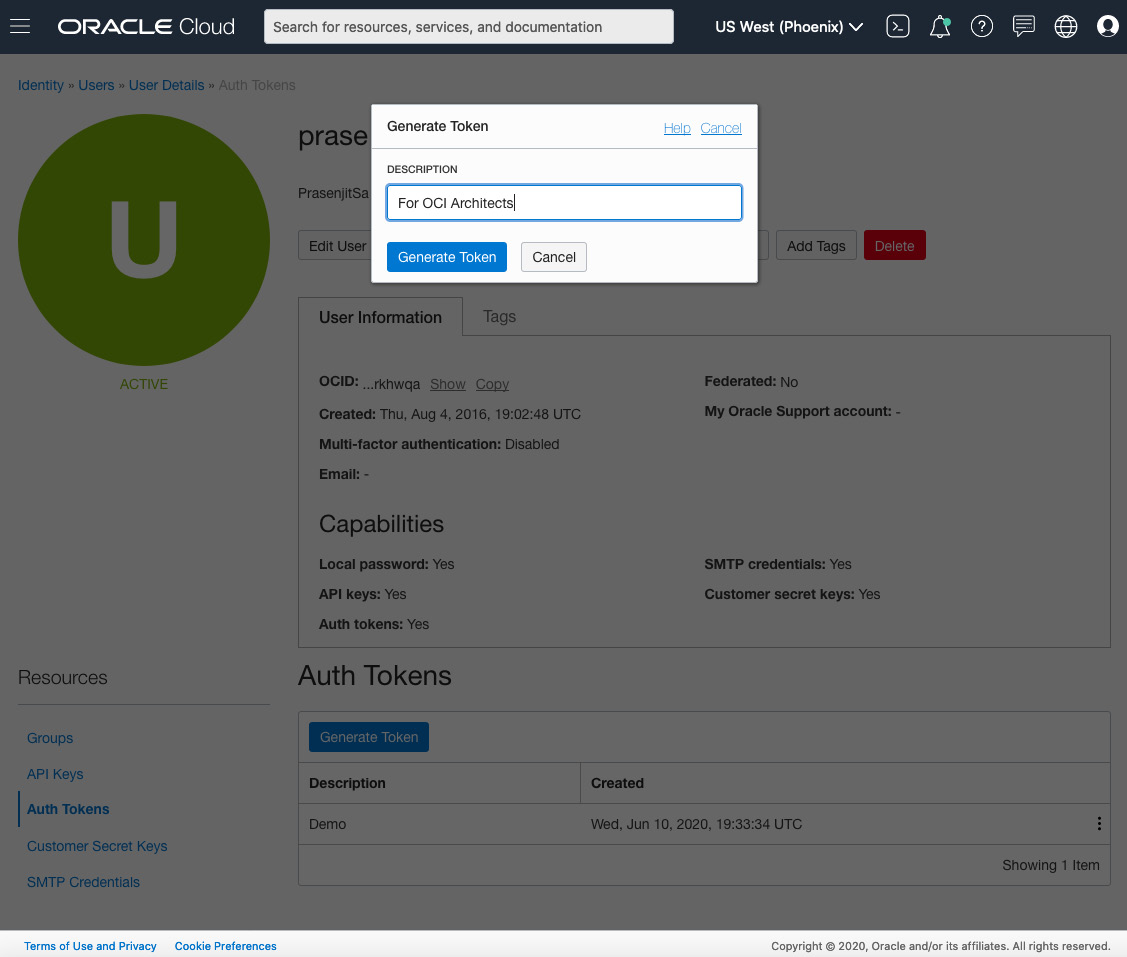


Figure 7.5 – Auth token generation

1. Once it's generated, click on the **Show** button to check the token and copy it to a safe place, as shown here:

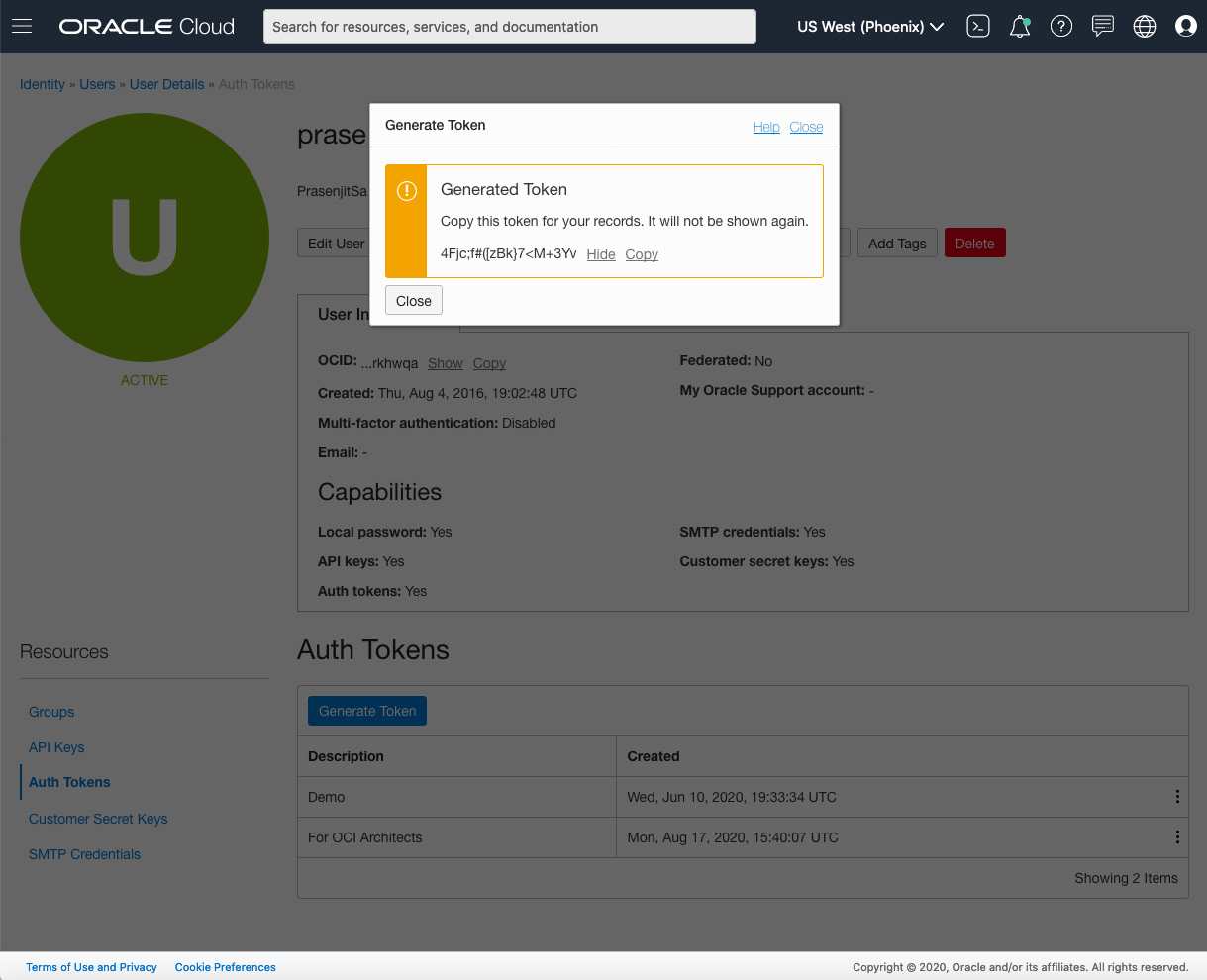


Figure 7.6 – Token value

1. Click on **Close**.
2. Lastly, to use the registry service, the users need to be either a part of the admin group or part of a group that the policy grants the appropriate permissions to, as follows:

- **Allow group dev-viewers to inspect repositories in tenancy**: This policy defines the ability to see all the repositories in OCI registry that belong to the tenancy.

- **Allow group dev-managers to manage repositories in tenancy**: This policy defines the ability to perform any operation on any repository in OCIR that belongs to the tenancy (pull an image, push an image, create/delete repositories, and so on). We looked at the policy statement in detail in [*Chapter 2*](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9781800566460/B16798_02_Final_NM_ePub.xhtml#_idTextAnchor028), *Understanding Identity and Access Management*, so you already know how to write a statement and at what level.

*NOTE*

*Repositories are tenancy-level resources, so if you want to control the access of these repositories, then you must create policies at the root compartment. This is because this is the top tier compartment of the tenancy.*

In this section, we learned how to access the registry. The token that you just generated is required to access the registry. We'll create a registry in the next section.

**Creating a repository**

OCIR stores the container images in repositories. You can either create the repository in advance or have it ready to use. However, when you push an image to the OCIR, if there is no existing repository with the name provided, then the OCIR will create a private repository automatically for you.

Let's create a repository:

1. Sign into the Oracle Cloud Infrastructure console.
2. Select the region where you want to create this repository.
3. Click **Create Repository**.
4. Specify the following details for the new repository:

- **Repository Name**: Provide a repository name of your choice.

- **Public**: Choose whether this repository will be private or public.

Click **Submit**. You can see an example of this in the following image:

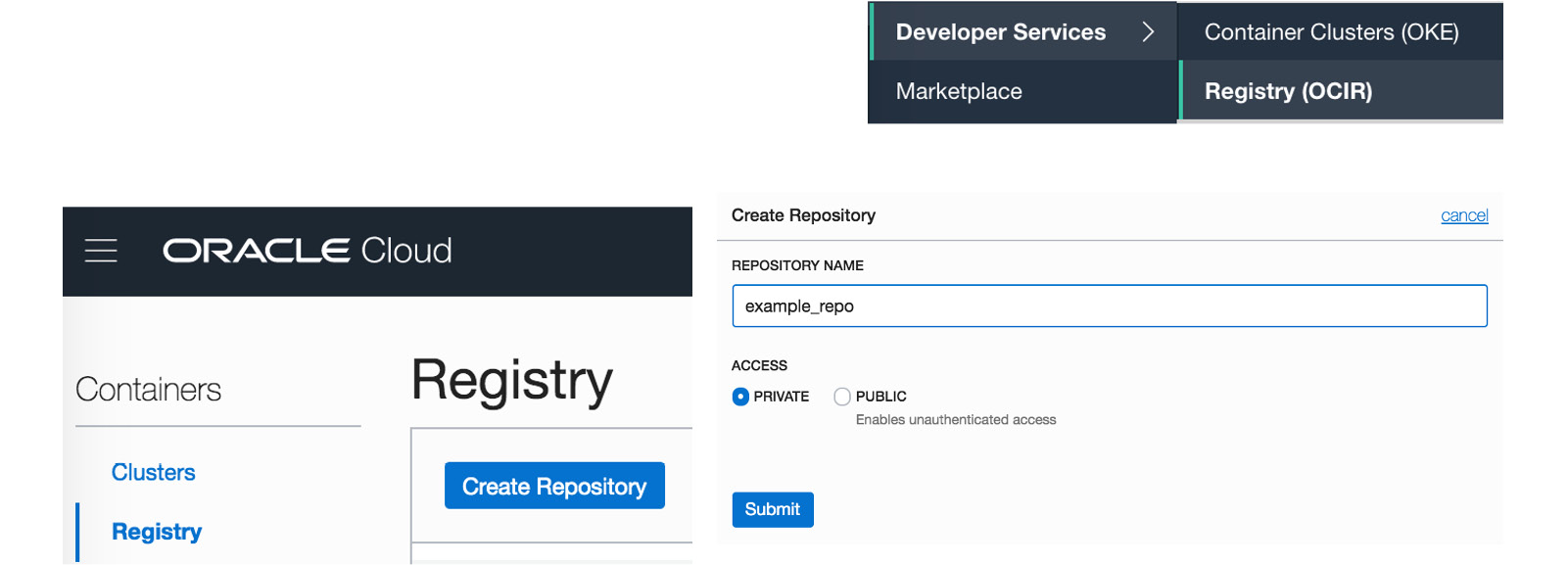


Figure 7.7 – Creating an OCIR repository

With that, you've learned how to create a repository. Now, we will create a Docker container image and push it to the repository.

**Creating a Docker container image**

Let's build a Docker image using a sample Python flask application and push it to our OCIR repository. We will use the OCI Cloud Shell to do this:

1. In the top-right corner of the console, open **Cloud Shell**.
2. Run **git clone**<https://github.com/stretchcloud/flask-rate-limiter-cors-auth>.
3. Go inside the cloned directory by typing **cd flask-rate-limiter-cors-auth/**. You will see a **Dockerfile** inside this directory.
4. Now, build the Docker image by typing **docker build -t flaskapp:latest**. This can be seen in the following screenshot:

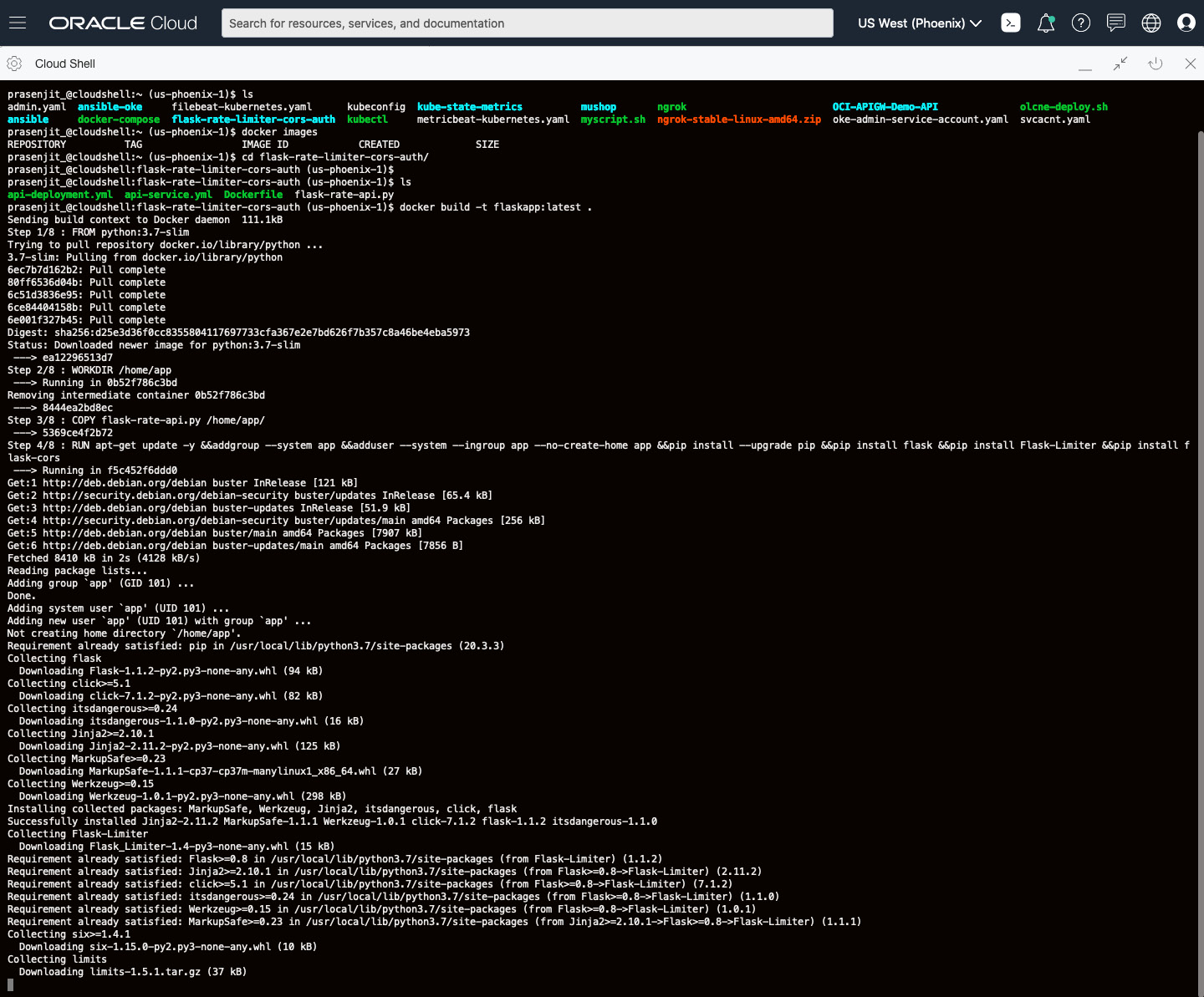


Figure 7.8 – Creating a Docker container image

1. Check the Docker image once it's been built by typing **docker images**. Copy the Docker image's ID.

With that, you've created a Docker container. Now, let's push it to the registry.

**Pushing and pulling a Docker container image**

Once you have the image ready, you can log into OCIR to push this image and then pull it anywhere you like:

1. We need to log into OCIR and to do that enter **docker login <region-key>.ocir.io**. You need to replace the **region-key** according to the OCIR region where you are logging in.
2. It will ask for the username at first and the format for the username is **tenancy-namespace/username**. You need to replace **tenancy-namespace** with your tenancy namespace, and you will get that from the **Tenancy Details** page. If you are using federated identity with Oracle Identity Cloud Service, then you need to use **tenancy-namespace>/oracleidentitycloudservice/username** as your format.
3. When prompted, enter the auth token you copied from the *Preparing for pushing and pulling images from the registry* section.
4. Tag to the image that you're going to push to the OCIR, like so:

**docker tag image-id target-tag**

For example, you might enter the following:

**docker tag 35255459d043 phx.ocir.io/intprasenjits/demoproject/flaskapp:latest**

1. If you want to check whether the image has been tagged correctly, run the Docker images and verify that the list of images includes the tag you specified. The following screenshot shows an example of this:

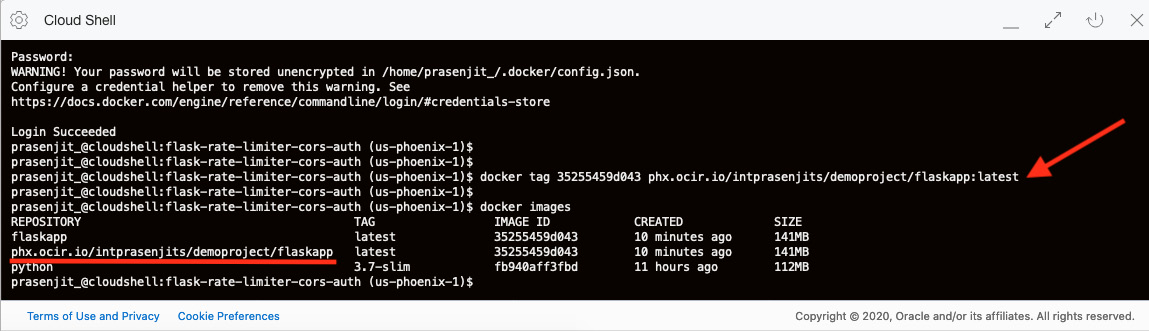


Figure 7.9 – Tagging a Docker image

1. Push the Docker image from the client machine to the OCIR by entering the following:

**docker push target-tag**

The following is an example of this:

**docker push phx.ocir.io/intprasenjits/demoproject/flaskapp:latest**

1. From the navigation menu, navigate to **Developer Services** and click on **Container Registry**.
2. Check the image that you pushed and click on the tag. You will see the image's layers, the size of the image, how many times it has been pulled, and so on:

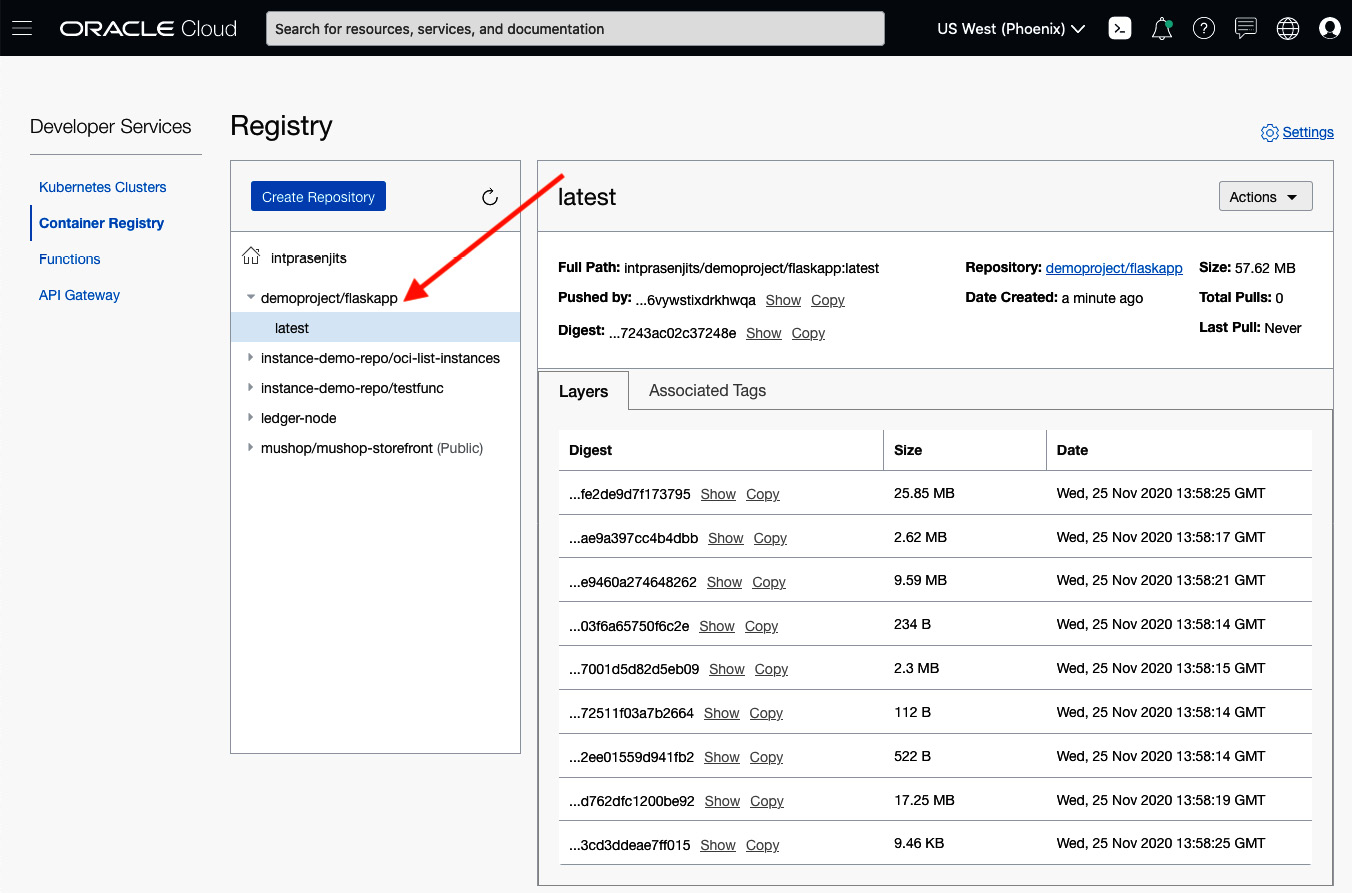


Figure 7.10 – Docker image layer

1. To pull the image, you must use the same Docker CLI. Pull the Docker image from OCIR to the client machine by entering the following:

**docker pull region-key.ocir.io/tenancy-namespace/repo-name/image-name:tag**

The following is an example of this:

**docker pull phx.ocir.io/intprasenjits/demoproject/flaskapp:latest**

1. Once you've pulled this image into your local dev machine, check if you can run this container and access the application endpoint.

Run the container by entering the following:

**docker run -d -p 5000:5000 phx.ocir.io/intprasenjits/demoproject/flaskapp**

Access the application endpoint by entering the following:

**curl -X GET -H "Content-type: application/json" http://127.0.0.1:5000/ping**

With that, you've seen how easy is to use the OCIR to store your private container images securely and then run it anywhere you wish. You can even use the same repository and image to deploy the application on top of OKE. For that, you need to create a secret within Kubernetes using the auth token and go from there.

**Deploying microservices on OKE**

Before we deep dive into Oracle's managed Kubernetes offering, let's provide a short introduction to Kubernetes itself. Learning about Kubernetes is a huge topic. This chapter will not cover all the aspects of Kubernetes; it will only focus on the key concepts and how OCI's managed Kubernetes service works.

**Getting started with Kubernetes**

Kubernetes is an open source project that's maintained by a community of developers and was created to solve container orchestration problems. By its very nature, Kubernetes is highly available and forms a single cluster unit. Kubernetes allows you to deploy and run your application in the form of disjoined/distributed software services, without pinning it to run on a specific computer.

You can manage your cluster through a web UI or through a CLI (**kubectl**). A Kubernetes cluster has two types of resources, as follows:

* A **master** node, which is responsible for managing the state of the cluster and coordinates the jobs within the cluster.
* A worker or **node**, which is responsible for running the scheduled containers (applications).

A Kubernetes worker node is responsible for running the containers on them and they can be either in the form of a virtual machine or even a physical machine. Each worker machine runs a process called a kubelet. This kubelet is responsible for managing nodes and communicating with the Kubernetes master. The following is a diagram of the Kubernetes architecture:

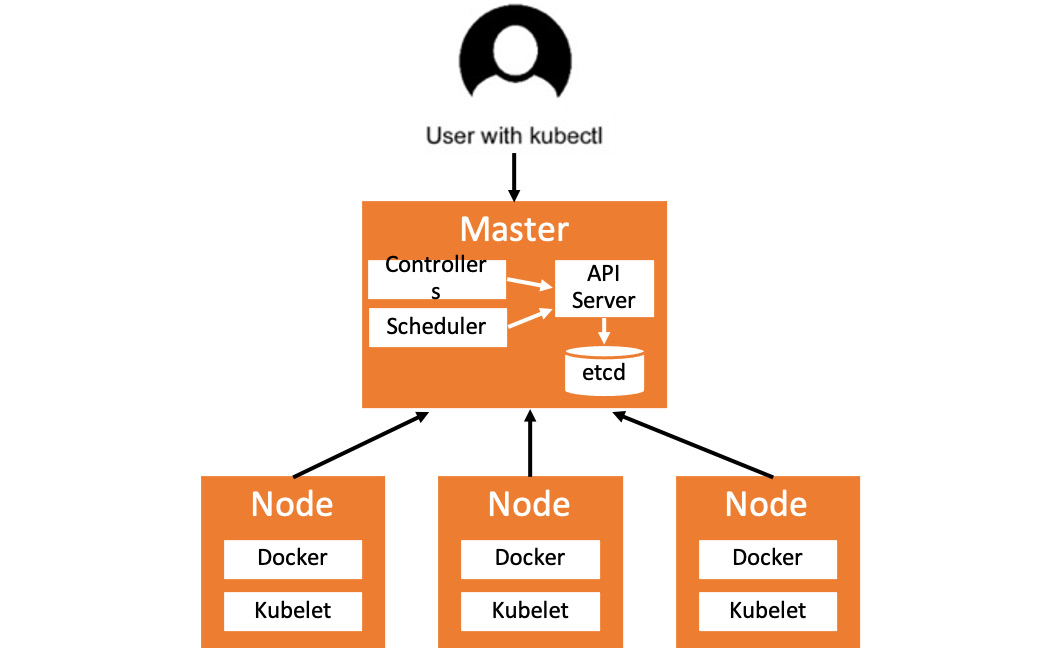


Figure 7.11 – High-level architecture of Kubernetes

You deploy your distributed applications in the form of containers on top of a Kubernetes cluster. You need to create a declarative file in YAML format to deploy your application. Sometimes, you can deploy your application using the command line as well. The advantage of using this declarative method is that you can update the parameters later on, to change how Kubernetes should maintain the state of this application.

When you send this YAML file using the **kubectl** command line, the Kubernetes master node that runs the API server sends an instruction to the kube scheduler, which schedules the container on an available node.

When you create a deployment, Kubernetes create **pods** to host your application instance.

However, Kubernetes doesn't run the container directly; instead, it creates a wrapper on top of it called a pod. This pod can run one or more containers within it. A pod also shares some key resources with those containers, such as its storage and a unique IP address.

Each pod gets scheduled on a particular node by the kube scheduler and remains running on that node until you terminate or delete it. If your Kubernetes cluster observes a node failure, then identical pods will come up, based on the number of pods to maintain in the deployment YAML.

The following diagram shows the high-level architecture of how pods are tied up to nodes:

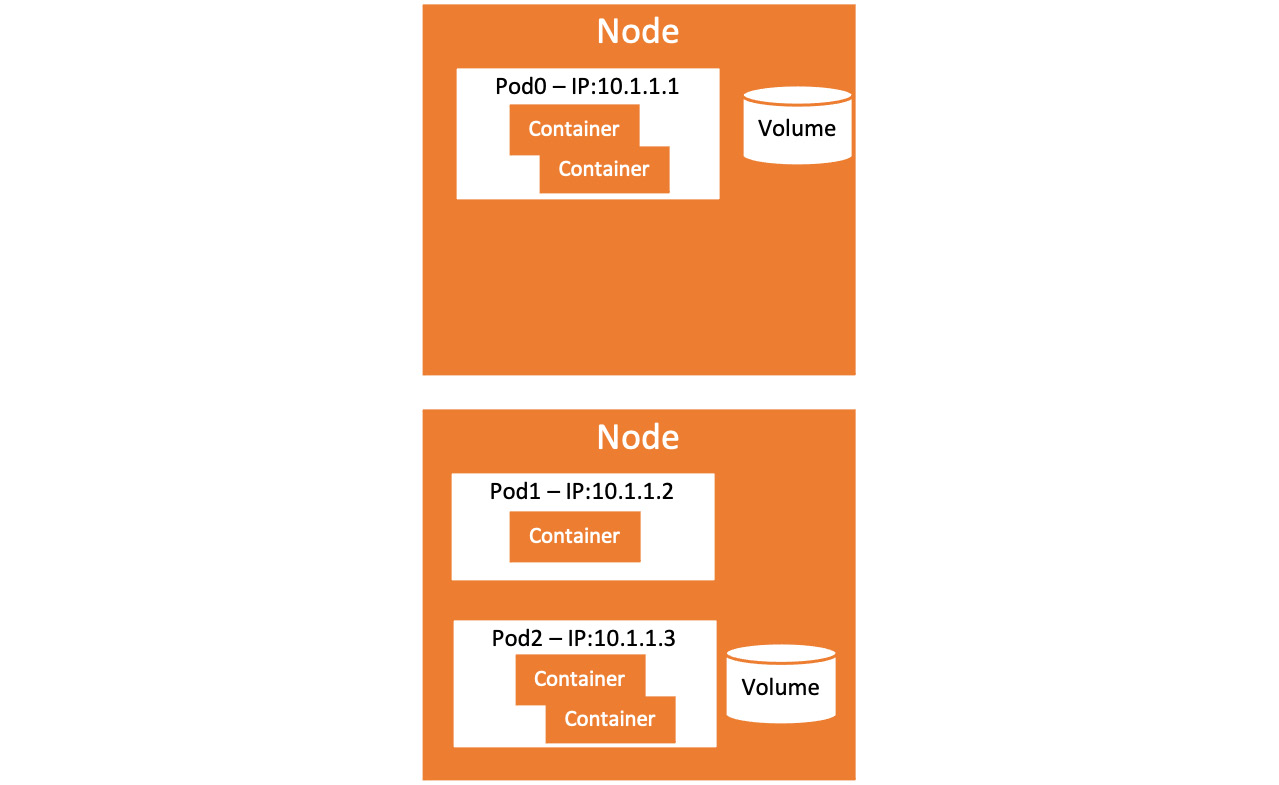


Figure 7.12 – Kubernetes deployment

So, now you know that you can deploy an application on top of a Kubernetes cluster. But how do you access it? By using a Kubernetes service.

A Kubernetes service creates a logical boundary by using a set of pods. This is called a service. Each of these logical boundaries will have a unique IP address assigned to it. Although your pods have IP address, they are not exposed outside the cluster.

Kubernetes services have been designed to receive traffic for the applications that are deployed within them. A pod within the cluster can send the network traffic or another service outside of the cluster can also send network traffic towards the service; for example, from the internet via a load balancer. The following is the architecture diagram of the Kubernetes service:

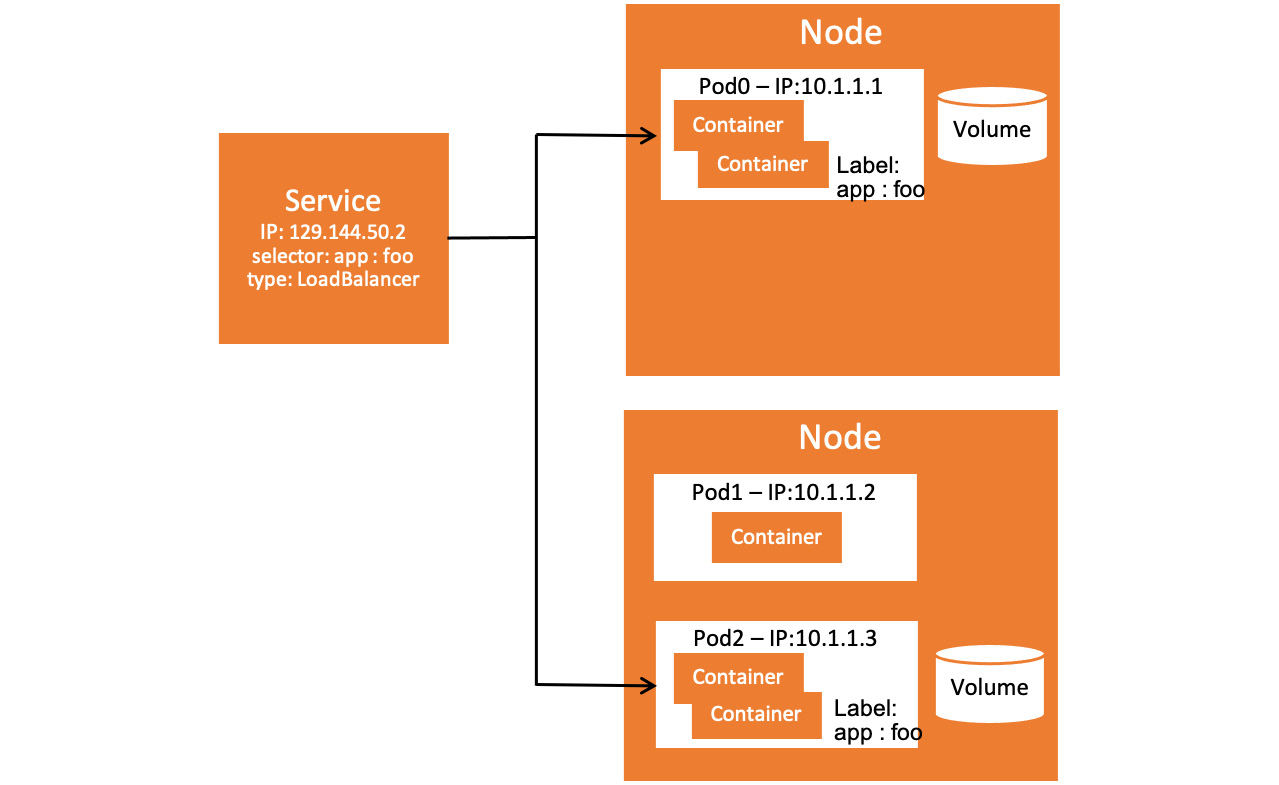


Figure 7.13 – Kubernetes service

Let's take a quick look at the components that make up a Kubernetes cluster:

* **kubectl**: This is a command-line tool that helps you manage Kubernetes resources using REST. You can perform all kinds of CRUD operations against Kubernetes resources.
* **Kubernetes API server**: Commands from **kubectl** are translated into REST API calls and issued to the API server. Users, other control plane components, and other tools such as CI/CD tools communicate with it.
* **Scheduler**: This works with the API server to schedule pods to the nodes. It provides information about the resources that are available on the nodes and those requested by pods. This is then used to decide which worker node will be selected for deployment.
* **Controller manager**: This is a daemon that watches the state of the cluster and reconciles the current state with the desired state; for example, Kubernetes components replication, keeping track of worker nodes, and handling node failures.
* **etcd store**: This is a simple, reliable, distributed, and consistent **Key Value** (**KV**) store. It persists the state of all REST API objects, how many pods are deployed to each worker node, and so on. Think of it as the **config.xml** domain.
* **Kubelet**: This is a service that's placed on each node that manages containers and is managed by the master. Kubelet keep the communication with the API server and responsible for managing resources on its node. Think of it like the node manager.
* **Docker**: This is the container runtime that runs on each node. It understands the Docker image format and how to run Docker containers.
* **Pod**: This is the smallest deployable unit that can be created, scheduled, and managed. It's a logical collection of containers that belong to an application.
* **Containers**: These are your applications, packaged and run in Docker or another supported container runtime format.
* **Proxy**: It runs on each node, acting as a network proxy and load balancer for a service on a worker node. Client requests that comes through an external load balancer will be redirected to the containers running in a pod through this proxy.
* **Load balancer**: This is an external load balancer that directs traffic to and from the proxy.

The following is a high-level Kubernetes architecture diagram showing all these components:

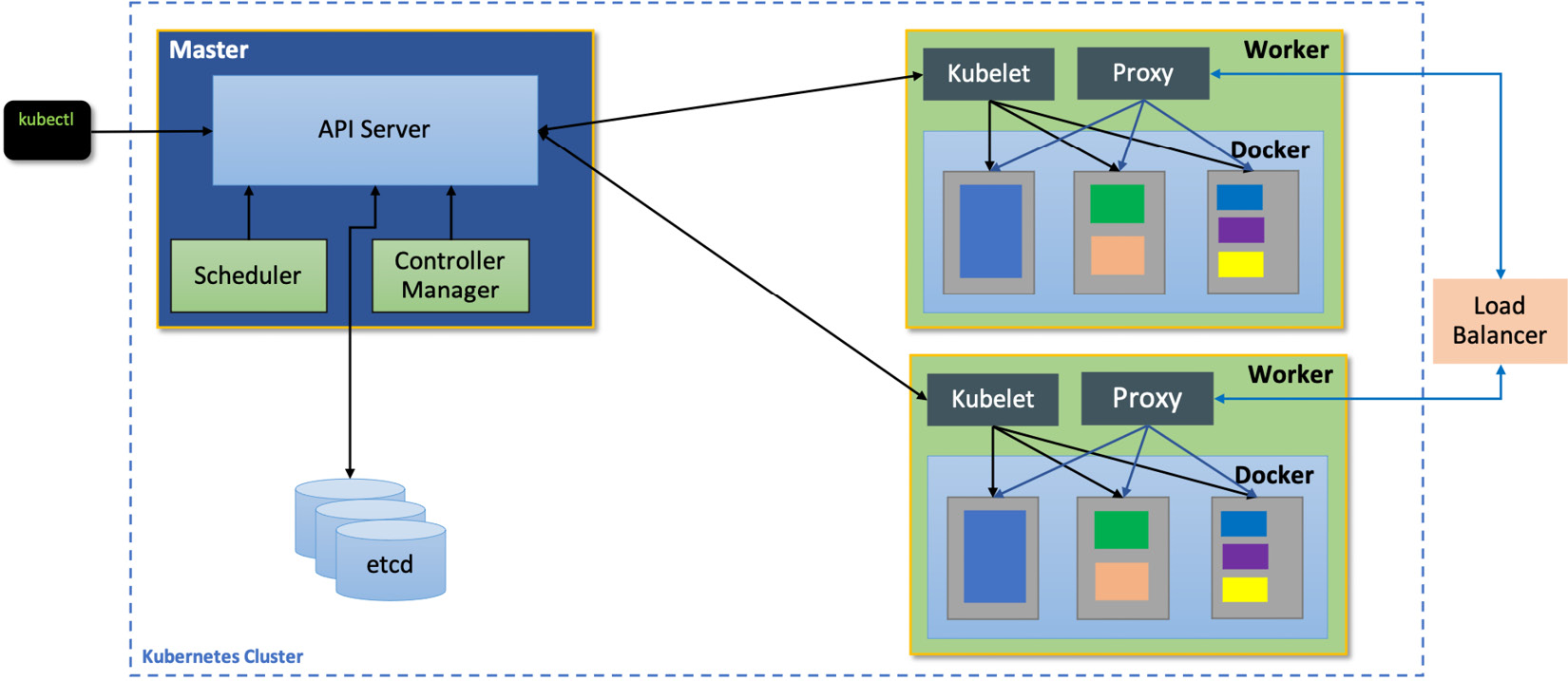


Figure 7.14 – Kubernetes service

So far, we have looked at the basic components and concepts of Kubernetes itself. We'll talk about Oracle's managed Kubernetes service in the next section.

**Getting started with Oracle Container Engine for Kubernetes**

As we mentioned earlier, **Container Engine for Kubernetes**, also known as **OKE**, is a fully managed Kubernetes service where Oracle will deploy the control plane in a highly available architecture and provide the master nodes free of charge. The customer will only pay for the resources that they are going to consume, such as their compute nodes, storage, load balancer, and so on. The following is the architecture diagram for OKE:

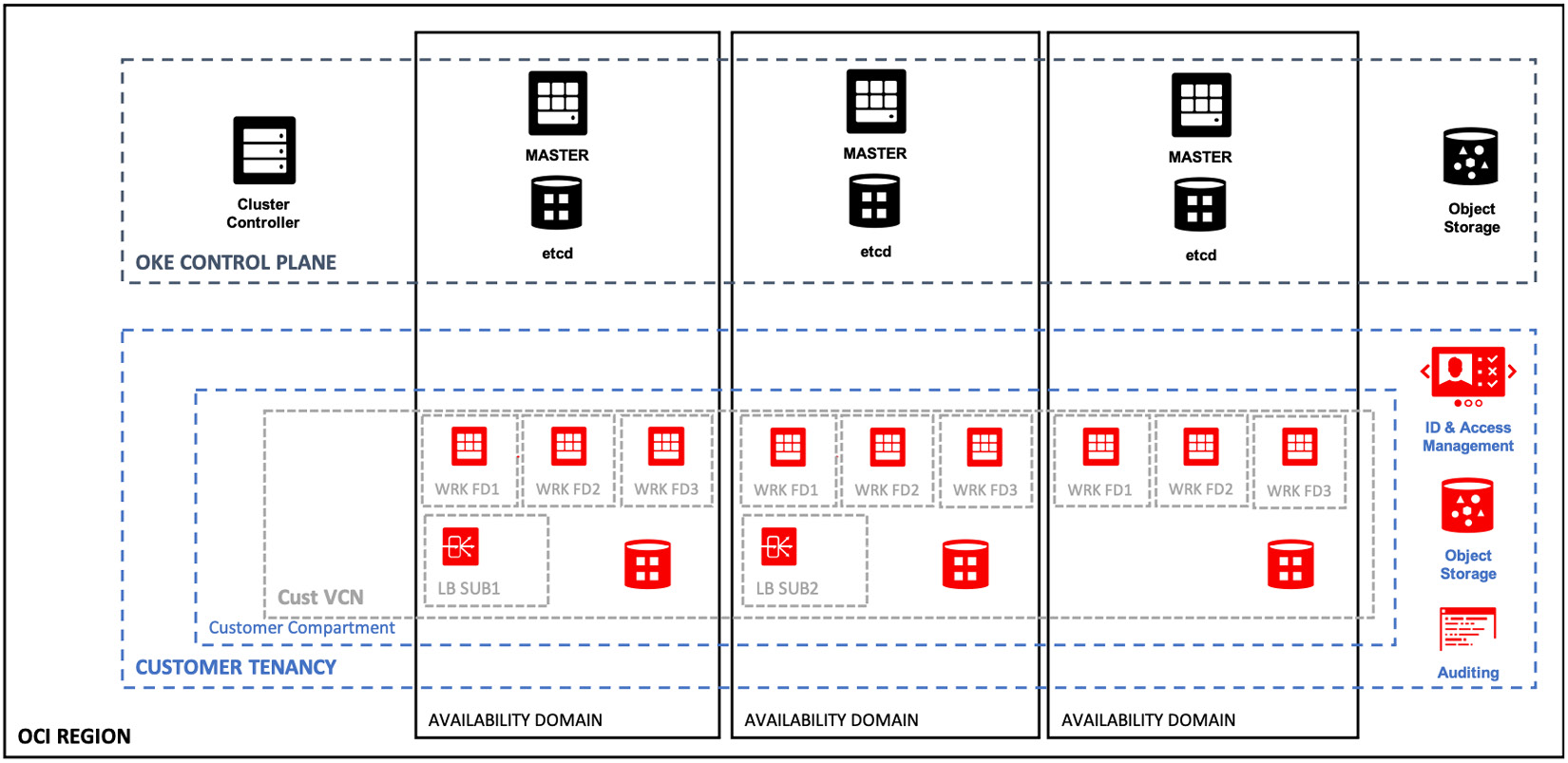


Figure 7.15 – OKE architecture

But what are the use cases for choosing OKE in the first place? Let's take a look.

**Lifting and shifting an application to OKE**

An Oracle-specific but popular use case for containerization is lift and shift WebLogic. WebLogic consists of WebLogic Application and WebLogic Server. WebLogic works with a database, such as an Oracle database, to serve web requests for, say, a sales portal. The entire WebLogic Application and Server are then containerized and defined in a Dockerfile, without any refactoring. After that, a CI/CD tool such as container pipelines or Jenkins is used to build, test, and push the resulting container image to the Cloud Infrastructure Registry. You can see a workflow of the lift and shift WLS on OKE here:



Figure 7.16 – Lifting and shifting WLS on the OKE architecture

This diagram, as well as the WebLogic Operator diagram (source available on GitHub at <https://github.com/oracle/weblogic-kubernetes-operator>), have been pulled from the registry using Oracle Container Engine for Kubernetes. The WebLogic Application and Server, as well as its operator, are then deployed to production on Kubernetes worker nodes. The resulting application is more scalable, available, and performant.

**Refactoring an existing application**

A general use case for leveraging containers is refactoring existing applications. In order to do this, an existing application, consisting of a user interface, an app server, and data access is rewritten as microservices, with each microservice running in a separate Docker container. The data store is also containerized. Databases such as MySQL, Cassandra, MongoDB, and more are available on the Docker Hub. Typically, you store the code in an **source code management** (**SCM**) system, such as GitHub.

The application and associated build scripts are then pushed into a CI/CD toolchain, such as container pipelines or Jenkins. After building and testing, Docker images are generated, which are then pushed into a private registry such as OCIR. Oracle Container Engine for Kubernetes, an enterprise-grade orchestration system for containers, can then be used to pull these Docker images and deploy the application and data store into production. The use of microservices allows the application to be more agile (code pushed more frequently), efficient, scalable, and easier to debug. The following diagram shows the workflow for refactoring an application and deploying it on OKE:

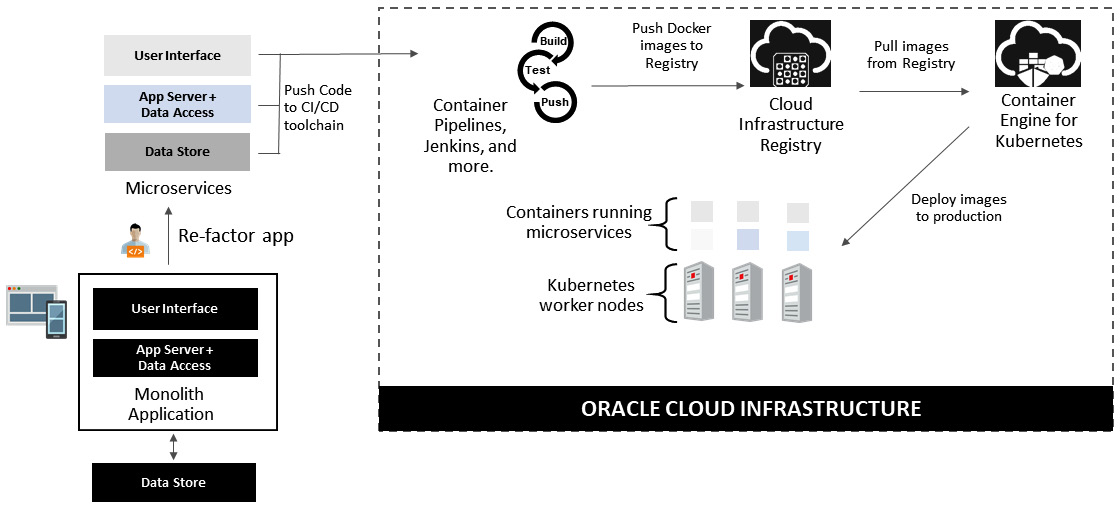


Figure 7.17 – Refactoring an application and deploying it to OKE

So far, you have learned about the characteristics of OKE and how it's used. Now, let's start creating a OKE cluster and then deploy an application and access it over the internet.

**Creating an OKE cluster**

Deploying an OKE cluster on OCI is really easy. They have two modes: one is a quick create option for most users, and most use cases can be solved using this workflow. But if you have more advanced use cases and you want to define everything on your own, then you can use the custom method that's available. Let's take a look at this:

1. Sign into the OCI console.
2. Click on the **Navigation** menu and go to **Developer Services**. Click on **Kubernetes Clusters**.
3. Click on **Create Cluster**.
4. Select the **Quick Create** option (this is the default option).
5. Click on **Launch Workflow**:

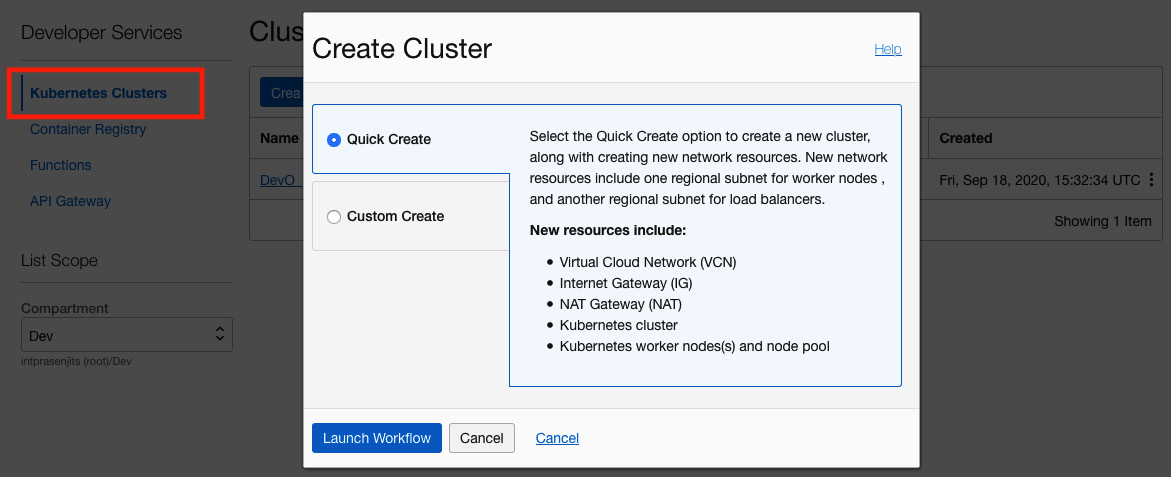


Figure 7.18 – OKE deployment workflow

1. Fill in the following details:

- Provide a name for the cluster.

- Select a compartment where you want to deploy it.

- Select an upstream **Kubernetes version** from the dropdown. By default, OCI will select the latest available version.

- Set the **visibility** type to **Private** or **Public**. For this example, we will create a private node.

- Choose a **shape** for the worker node. You can either choose a bare-metal node or a virtual machine. Both compute types are supported. If you choose the E3 flex type, then you can specify the amount of CPU and memory that you need for your worker node.

- Select your **number of nodes**.

- (Optional) You can choose to specify a **custom boot volume size** as well.

- Click on **Show Advanced Options**, it will show you the **Public SSH Key** field. Paste your public SSH key in there.

- Click on **Next**.

The following is a sample workflow:

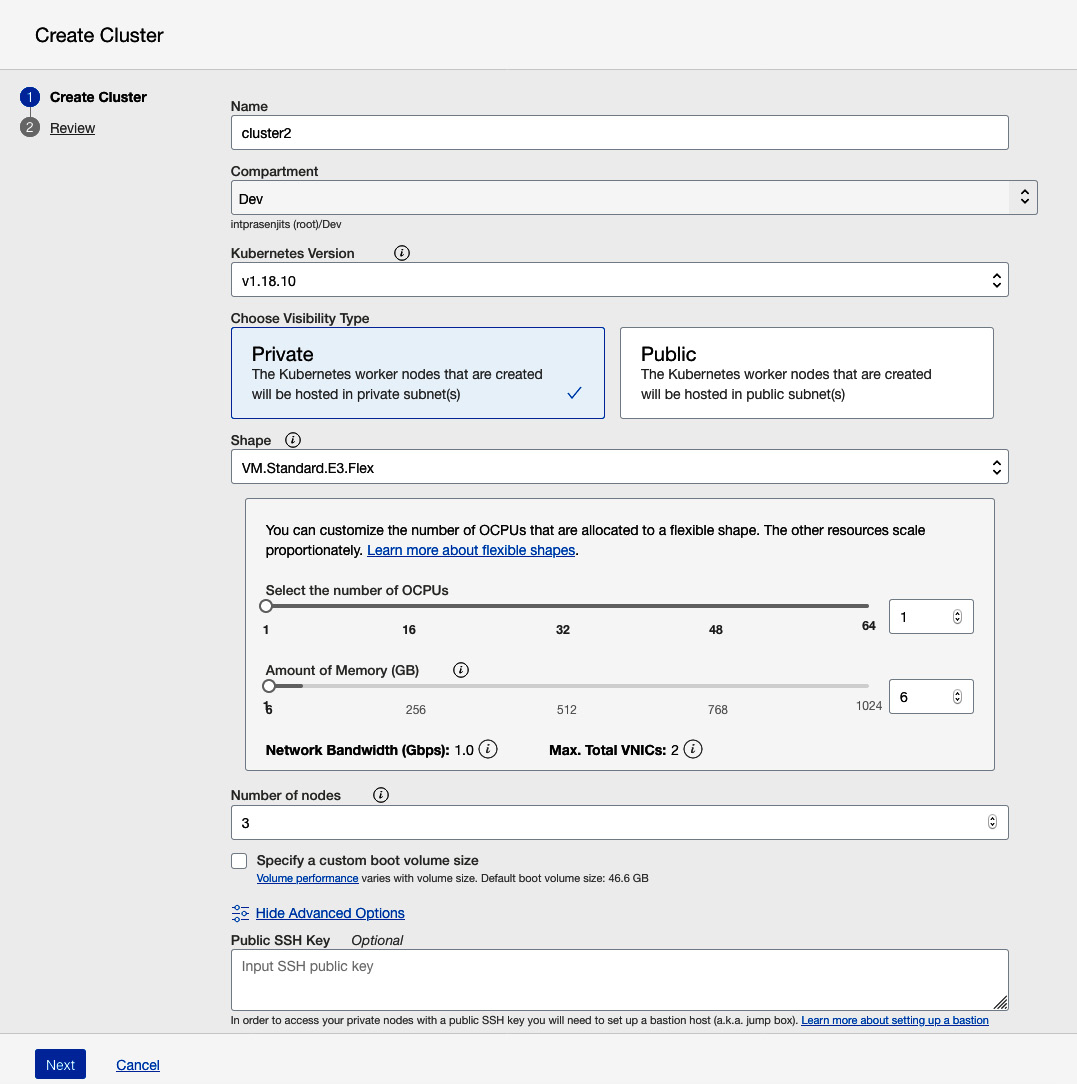


Figure 7.19 – OKE creation options

1. Review your options and click on **Create Cluster**.

Cluster creation will take about 5 minutes. Then, you can go through the next steps to access this cluster from the OCI Cloud Shell.

**Accessing an OKE cluster**

Once your cluster has been created, you can use the OCI Cloud Shell to access this Kubernetes cluster. Cloud Shell is outside the scope of this book, but you can read more about it at <https://docs.oracle.com/en-us/iaas/Content/API/Concepts/cloudshellintro.htm>. Follow these steps to access an OKE cluster:

1. From the **Cluster Details** page, click on **Access Cluster**.
2. Choose **Cloud Shell Access**, which is the default option.
3. Click **Launch Cloud Shell**.
4. Once Cloud Shell opens, copy the given command and run it. This will download the **kubeconfig** file from the master so that you can use **kubectl** to communicate with the cluster. You can see the workflow in the following screenshot:

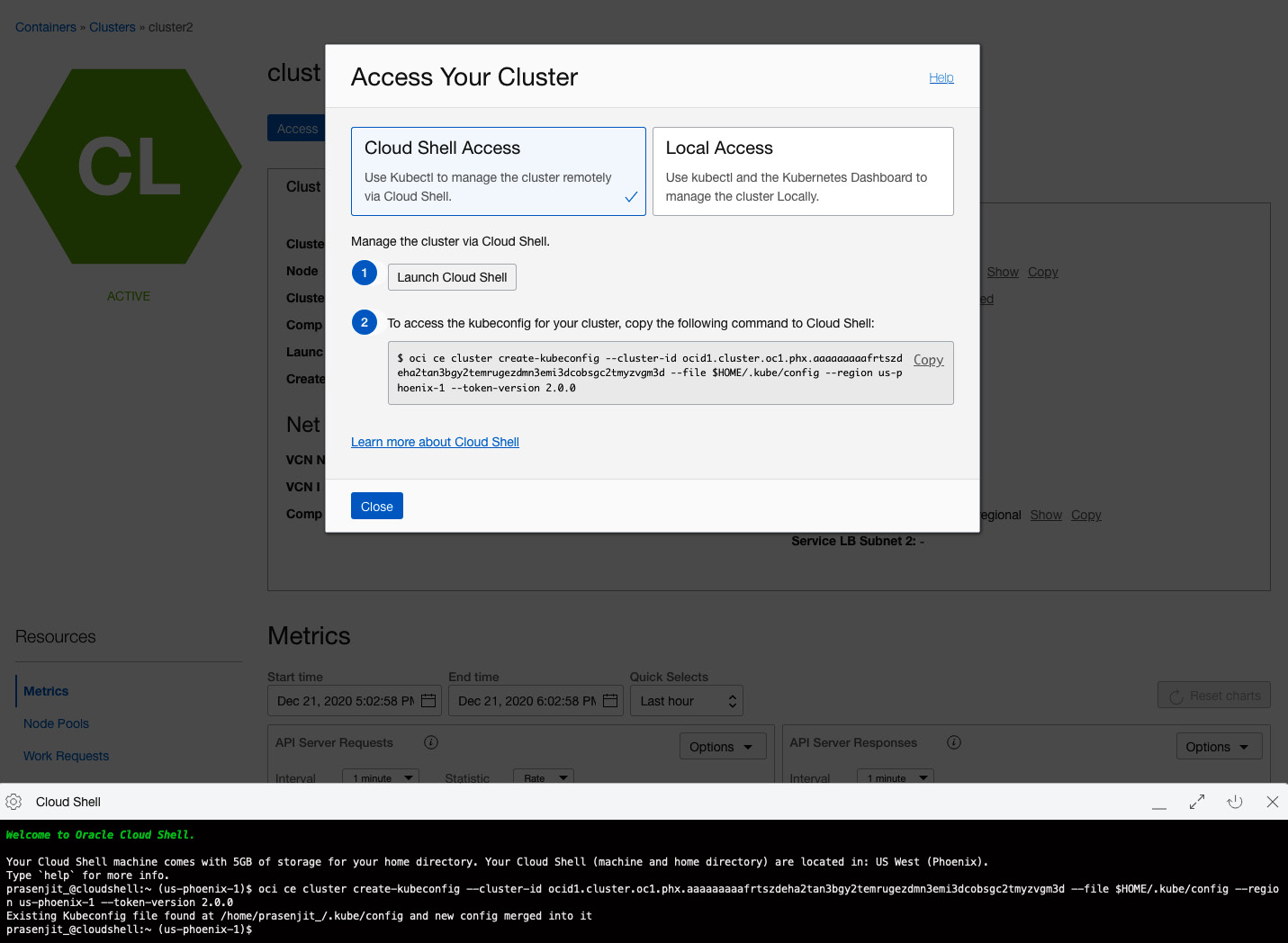


Figure 7.20 – Accessing the OKE cluster using Cloud Shell

1. Type **kubectl get nodes -o wide** to view the details of the worker nodes of this cluster:

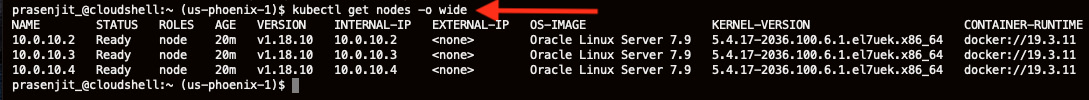


Figure 7.21 – Running kubectl to access the cluster

With that, you have created this cluster and can access it using Cloud Shell. Now, let's move on and deploy a simple web application and try to access it.

**Deploying a sample web application on an OKE cluster**

You can deploy an application in the same way you can deploy an application on any Kubernetes cluster. Let's take a look:

1. From Cloud Shell, type **kubectl apply -f**<https://raw.githubusercontent.com/stretchcloud/flask-rate-limiter-cors-auth/master/api-deployment.yml> to deploy the application.
2. Type **kubectl get deployment ratelimitapi** to see the status of the deployment.
3. Type **kubectl get po** to see the status of the pods. It must be in a running state:

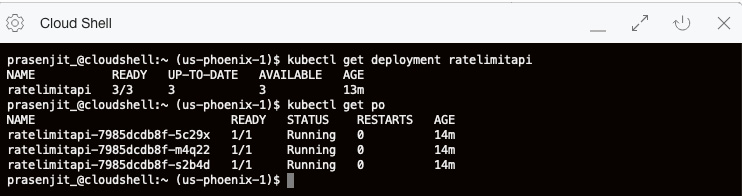


Figure 7.22 – Verifying the deployed microservice

1. So, this application has been deployed, but as we've already mentioned, to access this application, we need to create a Kubernetes service. Type **kubectl apply -f**<https://raw.githubusercontent.com/stretchcloud/flask-rate-limiter-cors-auth/master/api-service.yml> to create a Kubernetes service of the **LoadBalancer** type. In the background, this will deploy an OCI **LoadBalancer** that's 100 Mbps in size.
2. Type **kubectl get svc** to find out the public IP address of this application. You will need this to access this application over the specified port. In this case, it is **5000**.
3. Pick up the IP address of the service and type **curl -X GET -H "Content-type: application/json" http://<IP-Address>:5000/ping** to access this application, like so:

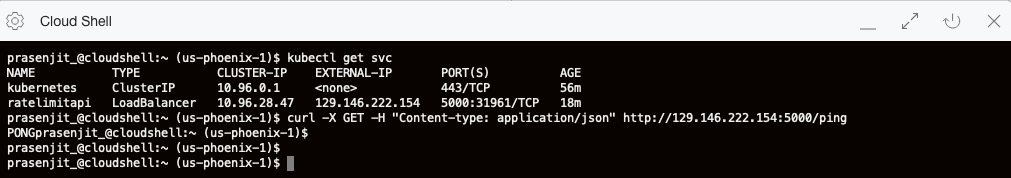


Figure 7.23 – Accessing the Kubernetes service

So far, you have learned how to create an OKE cluster, access it using Cloud Shell, and deploy a sample web application. In the next section, we will talk about how to upgrade the Kubernetes version of the cluster.

**Upgrading the Kubernetes version of an OKE cluster**

With the fast pace of upstream Kubernetes release, OCI is also keeping up the pace to make those bits available on OKE. If you have a cluster running an old version of Kubernetes, you can upgrade the cluster to the latest version with zero downtime.

Also, you have the choice to either perform an in-place upgrade for the worker node's Kubernetes version or you can choose to do this out-of-place. During the in-place upgrade, you don't need to perform any extra steps. But in the case of out-of-place, you are responsible for creating a new node pool that uses the latest available Kubernetes bits and then eventually delete the old node pool.

Let's upgrade the OKE cluster:

1. First, let upgrade the Kubernetes version of the cluster. To do that, go to the **Cluster List**page, choose the name of the cluster, and you will see a yellow exclamation mark on the **Version** field if an upgrade is available.
2. Click on the actions menu of the cluster where the upgrade is available. Click on **Upgrade Available**, as shown here:

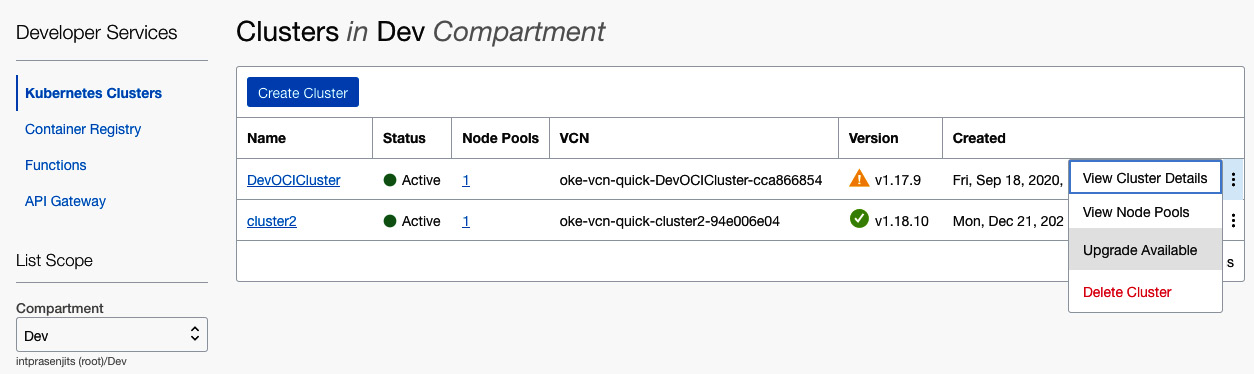


Figure 7.24 – Upgrading the OKE version

1. On the **Upgrade Cluster Master** screen, select the Kubernetes version you want the master to be upgraded to, and then click on **Upgrade**, like so:

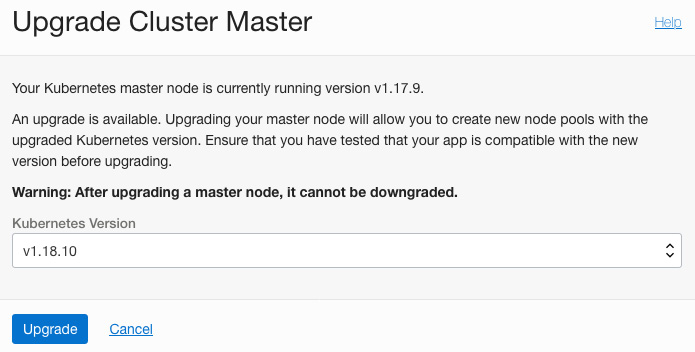


Figure 7.25 – Upgrading the cluster master

This operation will upgrade the master to the version that you choose. Let's upgrade the node's Kubernetes version. We will perform an in-place upgrade in this example:

1. Let's upgrade the Kubernetes version of the node pool. To do that, go to **Cluster** page, select the **Node Pools** tab. Select the node pool to go into the **Node Pool Details** page.
2. Click on **Edit**. Specify the required Kubernetes version from the list in the **Version** field.
3. Click on **Save Changes**:

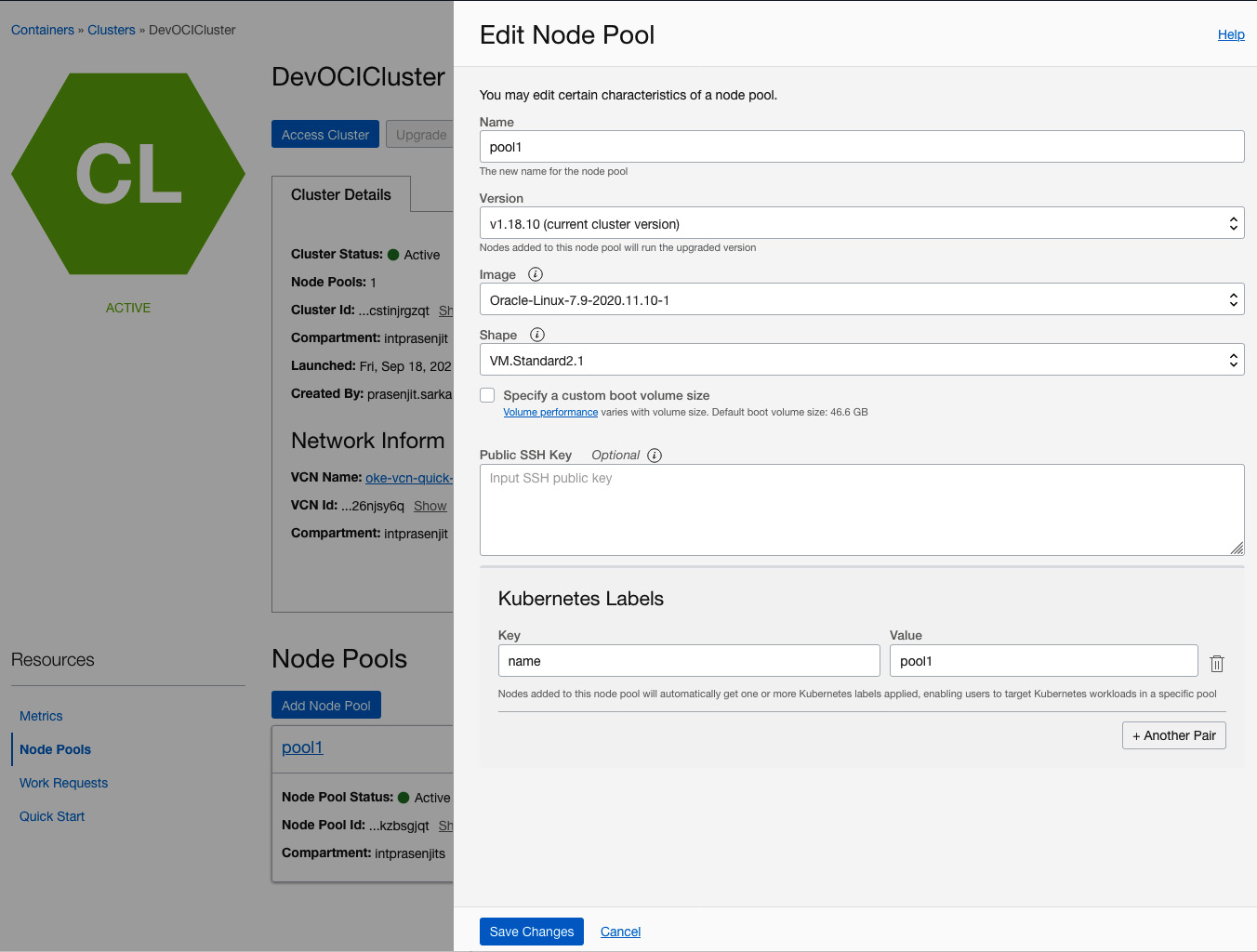


Figure 7.26 – Upgrading the worker nodes

1. Type **kubectl drain <node\_name>.** This will prevent any new pods from starting and at the same time, it will delete the existing pods as well. You are now all set to terminate this instance of worker node and start up a new worker node that has upgraded Kubernetes software version.
2. From the **Node Pool** page, select the **Node** by going into the **Nodes** tab. It will take you to the **Instances** page.
3. Select **Terminate** from the **More Actions** menu of the instance that you want to terminate.
4. Repeat *steps 4* to *6* for all the other worker nodes.

With that, you've seen how quickly and easily the upgrade process of Kubernetes happened for not only on the master but also on the worker nodes without gaining downtime for your application. In the next section, we will discuss using an API gateway as an ingress method for the applications running on OKE.

**Exposing microservices using the OCI API gateway**

An API gateway is a network attached device, much like **Load Balancer as a Service** (**LBaaS**). It is fully managed by Oracle; the customer does not need to manage it. If you want to expose a private application endpoint to the public internet and implement authentication, authorization, **cross-origin resource sharing** (**CORS**), rate limiting, routing, and so on, then the API gateway is the answer. Not only do you get the benefits of running an API gateway as an ingress to the application, but you can also get fine-grained monitoring and logging entry points for your code flow via the API that's invoked by a client.

You can create one or more gateways and attach them to a regional subnet, which then processes traffic coming in from the clients and then routes those requests to the defined backend services. You can use the same gateway to serve multiple endpoints. This can be a load balancer, compute instances, or Oracle functions. The following is a diagram of an API gateway:

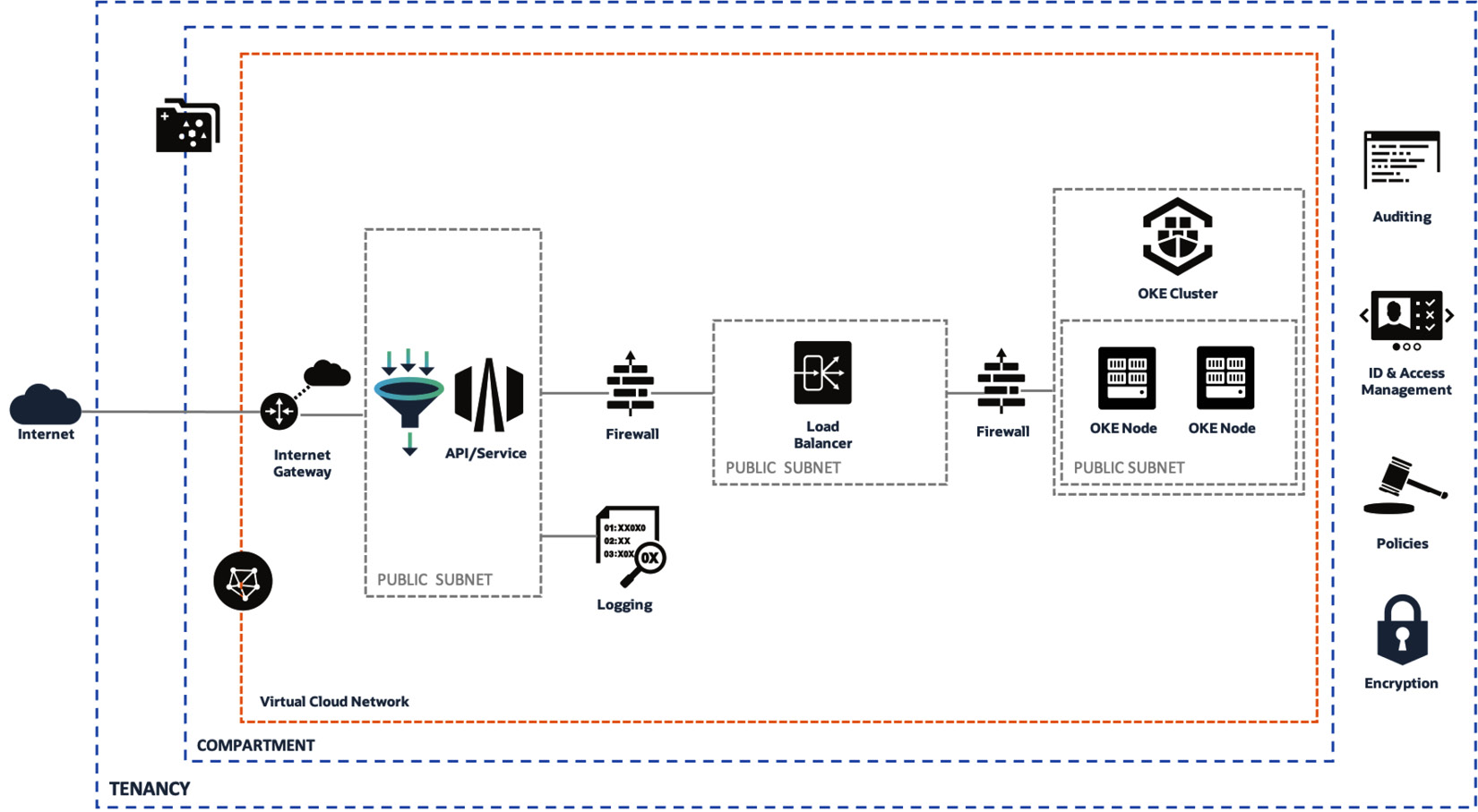


Figure 7.27 – Anatomy of an API gateway

Let's look at the different use cases and modeling for an OCI API gateway.

**API gateway within a cloud environment**

In this model, APIs are called via the public internet and access backend services running in the cloud. It is fully supported by the OCI security model. It uses **Open Authorization** (**Oauth**) and JWT to authenticate those API calls. Typical use cases for this are full cloud-native applications, accessing SaaS, and migrating applications to the cloud. The following is the architecture diagram for this use case:

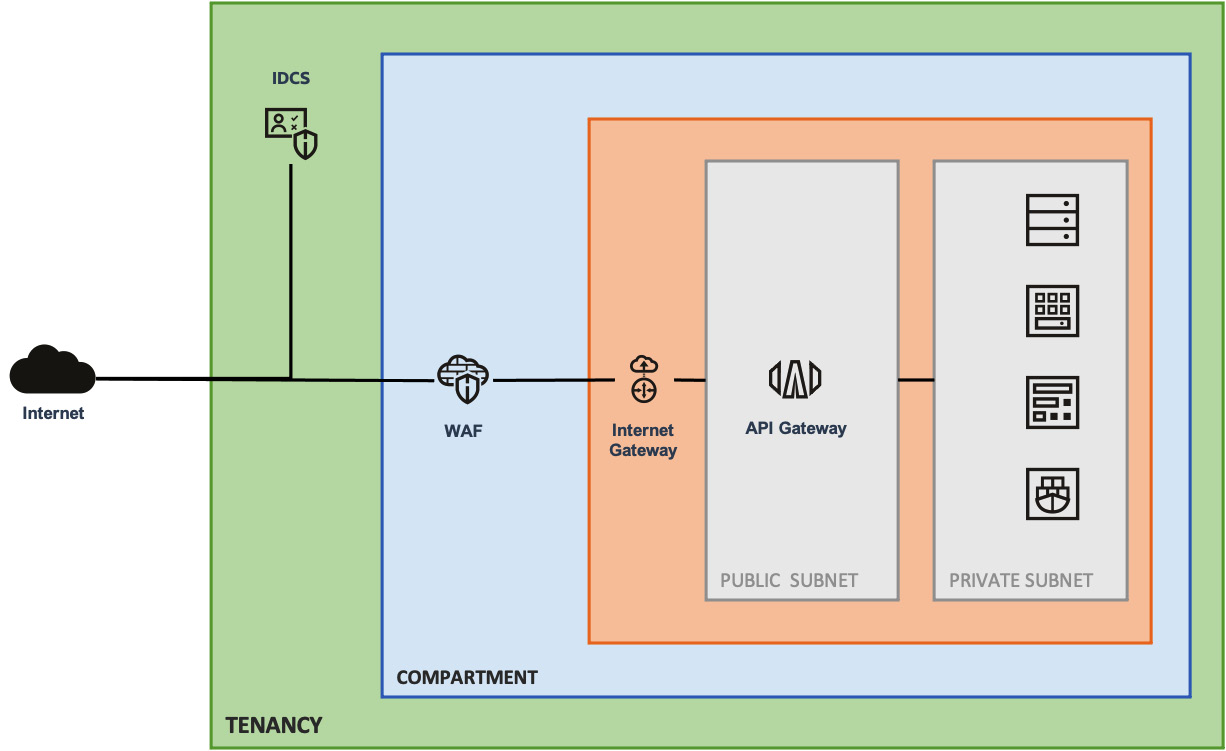


Figure 7.28 – API gateway hosting resources from the cloud

This diagram shows that the API gateway is frontending the resources being hosted within the same cloud environment.

**API gateway in a cloud to on-premises model**

In this use case, APIs are called via the public internet and access the backend services running on the customer's data center. This model extends the customer DMZ into OCI, thus making use of OCI's security and scalability. A typical use case for this model is to use OCI security and APIGW rate-limiting to protect internal applications. You can use this model to start migrating to the cloud. The following is the architecture diagram for this use case:

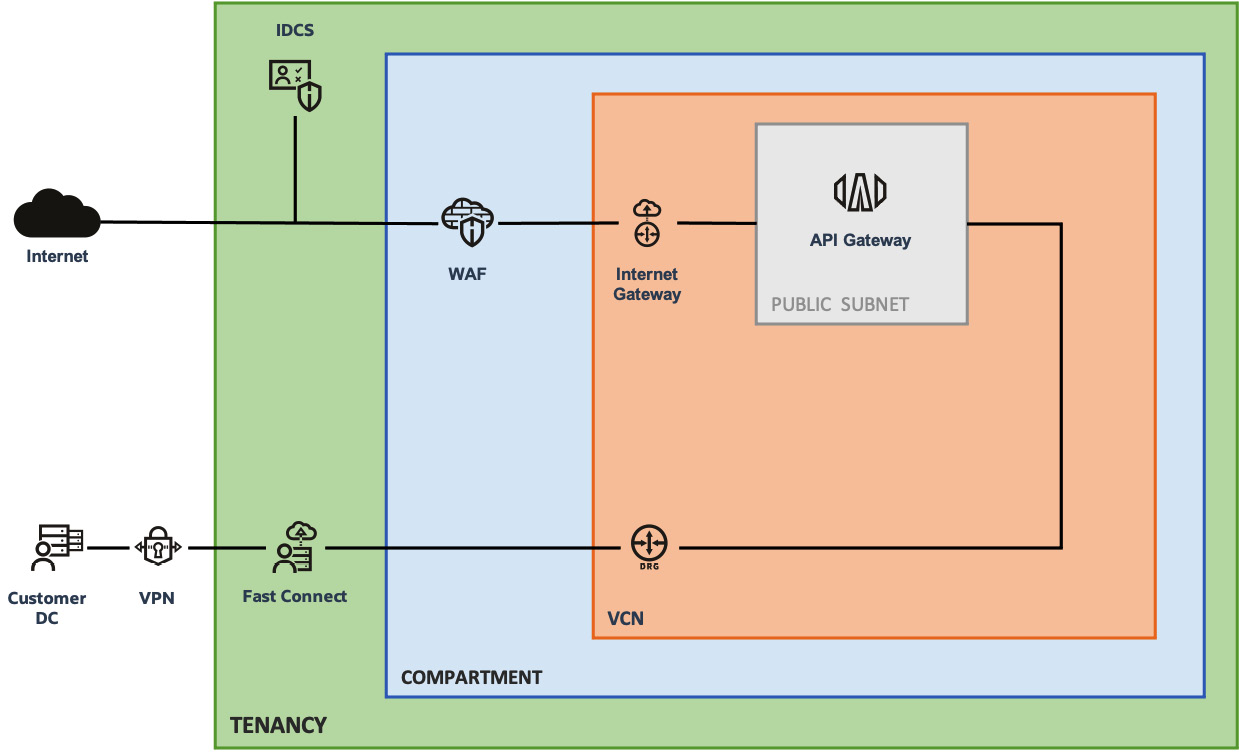


Figure 7.29 – API gateway hosting resources from on-premises

This diagram shows that an API gateway is frontending the application endpoint being hosted in a customer DC. A fast connect connection has been established to connect the customer DC to the cloud.

**API gateway in a hybrid model**

In this model, APIs are called via the public internet and access the backend services running on OCI and the customer's DC. You can use this model to extend internal applications with new capabilities (for example, microservices). You can also use this to protect external access to internal systems.

The following is an architecture diagram of this use case:

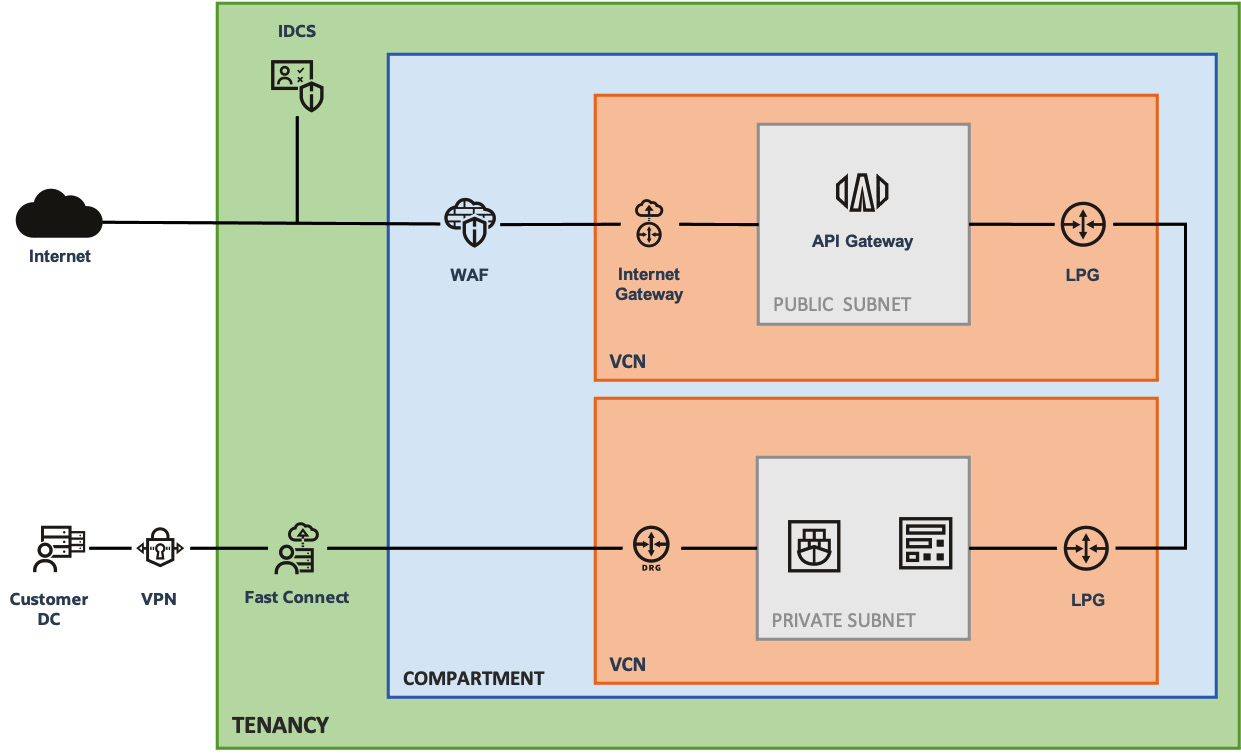


Figure 7.30 – API gateway in a hybrid model

This diagram shows that the API gateway is frontending the application endpoint being hosted in both the cloud and customer DC. A fast connect and local peering gateway has been established to make this hybrid networking possible.

**API gateway in a private cloud model**

In this model, APIs are called from internal users accessing services in the OCI and internal systems. You can use this model to extend internal applications with new capabilities (for example, microservices), application modernization, data center evacuation, and so on.

The following is an architecture diagram of this use case:

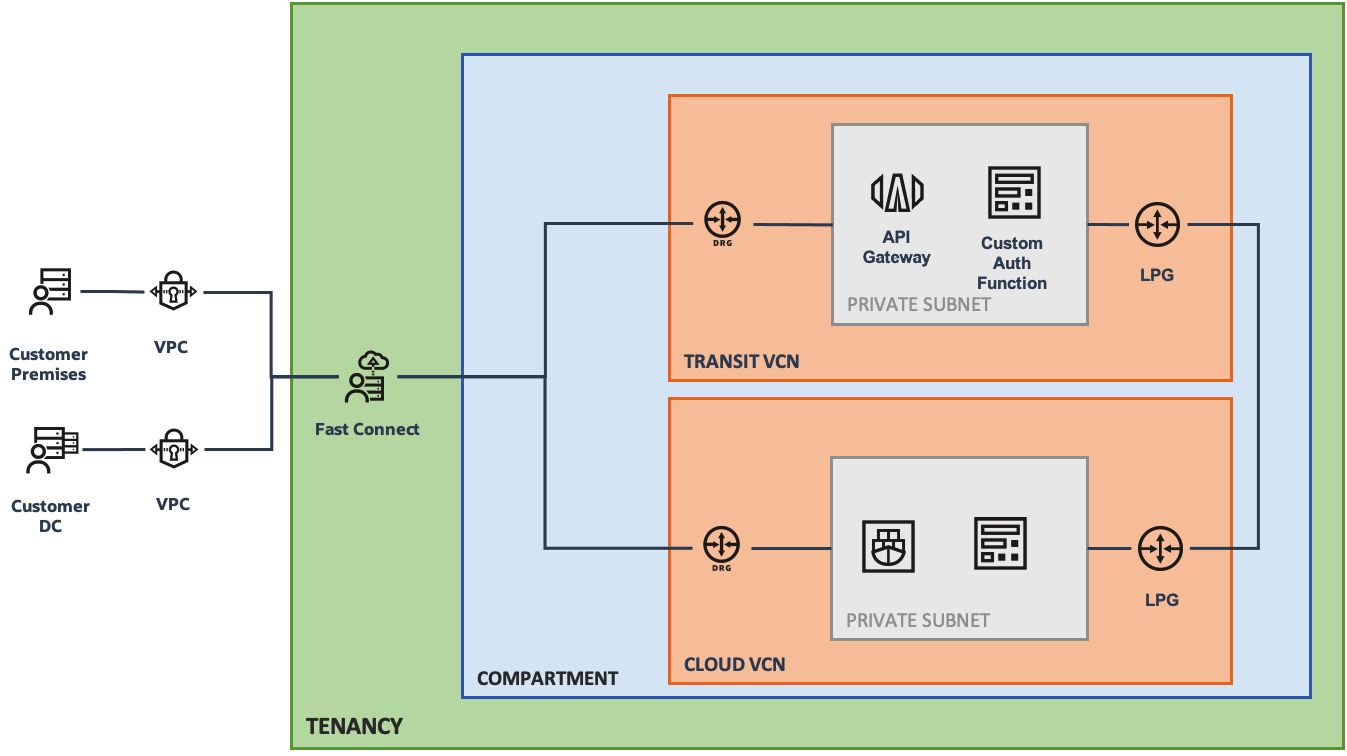


Figure 7.31 – API gateway in a private cloud model

This diagram shows that the API gateway is frontending the application endpoint being hosted in the cloud but is being accessed from the customer DC.

**API gateway concepts**

At this point, we've provided an overview of the API gateway and its use cases. Let's go through the OCI API gateway concepts that you need to know about before creating an API gateway.

**API gateways**

Oracle deploys API gateways as virtual appliances, similar to LBaaSes, which are hosted in a regional subnet. API gateways route the incoming requests to the appropriate backend services, which includes a private or public API endpoint. Not only that, OCI API gateway can also route incoming requests to Oracle functions.

**APIs**

An API is a combination of a set of resources, and the methods that are needed to access those resources (for example, **GET** and **PUT**).

**API deployments**

If you want to deploy an API endpoint, then you need to deploy an API gateway. You need to have an API deployment so that you can serve an API endpoint.

**API deployment specifications**

This describes the properties of an API endpoint through the deployment specification. An API deployment specification is required while you're creating an API gateway.

You define one or more backend services within those API deployment specifications. You also define the routing and methods to access those services.

**Routes**

A route is what connects to the backend service using a specified method. You define this routing in your API deployment specification.

**Policies**

API gateways implement two types of policies. One is called a request policy, while the other is called a response policy:

* The request policy performs a specified action on an incoming request without sending it to the backend. An example of a request policy is limiting the number of incoming requests and enabling CORS support.
* The response policy performs a specified action on a response that has been returned from a backend before but yet to be sent to the caller.

**Authentication and authorization policies**

The OCI API gateway offers an authentication policy that can be applied to all the routes in the specification, as well as an authorization policy, which is specified and can be applied at the route level.

**JWT authentication**

With this feature, the API manager can simply configure JWT validation by either manually providing the public verification keys from their authorization server or, alternatively, by configuring the OCI API gateway to pull those keys at runtime.

This will enable them to easily protect access to the APIs that have been deployed on an OCI API gateway. It does this by integrating with any existing JWT-capable identity provider such as Oracle IDCS, Okta, or Auth0.

If a token passes the authentication and authorization policies, the claims are available in the context variables. They can be referenced in the deployment specification; for example, **http://150.136.174.81:3000/users/${request.auth[user\_id]}/todos/${request.path[todoId]}**.

Let's go ahead and create an API gateway and then deploy it so that we can host an application with policies attached to it.

**Creating an API gateway**

We will use the OCI console to create this gateway. Let's get started:

1. Once you log in to the console, click on the navigation menu and select **Developer Services**, and then click on **API Management**.
2. On the **Gateways** page, click on **Create Gateway**.
3. Provide these values:

- **Name**: The name of the new API gateway.

- **Type**: The type of API gateway to create. In this example, we will choose a public gateway.

- **Compartment**: Where you want to create this gateway.

- **Custom DNS**: This option lets you determine the TLS certificate (and associated domain name) that the API gateway will use.

- **VCN**: Which VCN you want to create this gateway in.

- **Subnet**: Provide a public regional subnet.

1. Click **Create** to create the new API gateway. The following screenshot shows all the aforementioned variables filled in:

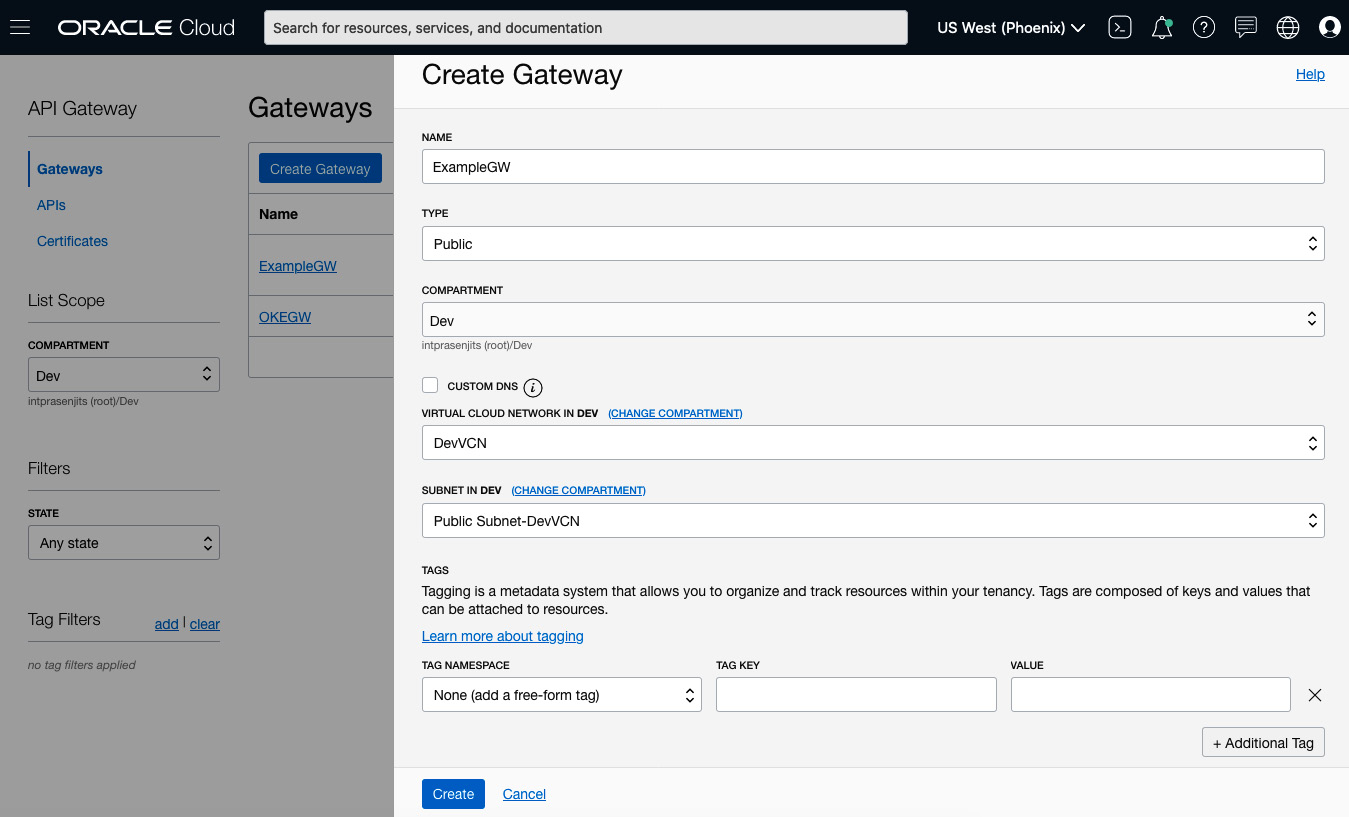


Figure 7.32 – Creating an API gateway

It will take a few minutes to create the new API gateway. Once the gateway is active, you need to create the deployment. Let's do that now.

**Creating an API gateway deployment**

We will use the OCI console to create this gateway. Let's get started:

1. Once you log in to the OCI console, click on the navigation menu and select **Developer Services** and then select **Gateways**.
2. Select the API Gateway where you want to deploy the API deployment, which will take you to the **Gateway Details** page.
3. Click on the **Deployments** link in the **Resources** list.
4. Click on **Create Deployment**.
5. By default, **From Scratch** is selected.
6. Enter the name of the deployment.
7. Specify a path prefix, such as **/v1** or **/v2**.
8. Choose the compartment where you to deploy this.
9. Specify **Authentication**, **CORS** and **Rate Limiting** policies in the **API Request Policies** section.
10. Select the **Execution Log Level** from the drop-down menu of the **API Logging Policies** section. By default, it is set to **Information**. This can be seen in the following screenshot:

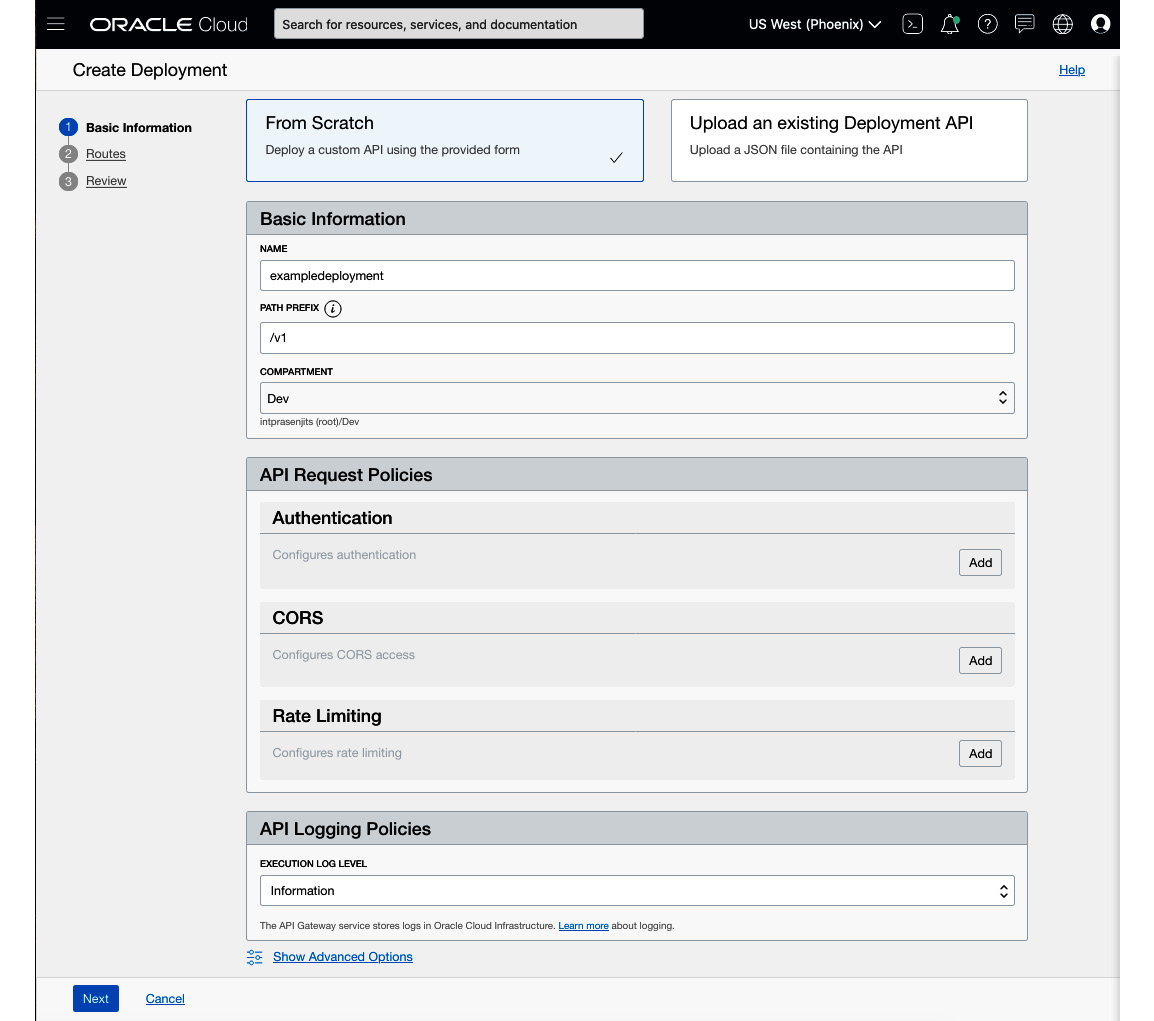


Figure 7.33 – Creating a deployment

1. Click **Next**.
2. In the **Route 1** section, you need to define your first route that maps a path and one or more methods to a backend service:

- **Path**: A path for API calls that uses the listed methods to the backend service.

- **Methods**: One or more methods accepted by the backend service, separated by commas. In this example, our application handles the GET method, so we specified that method.

- **Type**: The type of the backend service. In this example, we will choose an HTTP endpoint, but you can choose any of the following options:

- **HTTP**: In this field, you need to specify a URL, timeout details, and whether you want to disable the SSL verification or not.

- **Oracle Functions**: You need to specify the application and function.

- **Stock Response**: In this field, you need to specify the HTTP status code, specify the content in the body of the response, and one or more HTTP header fields.

- **URL**: The URL of the actual service endpoint where the service is running.

The following is a screenshot showing how to set up the **Route 1** section:

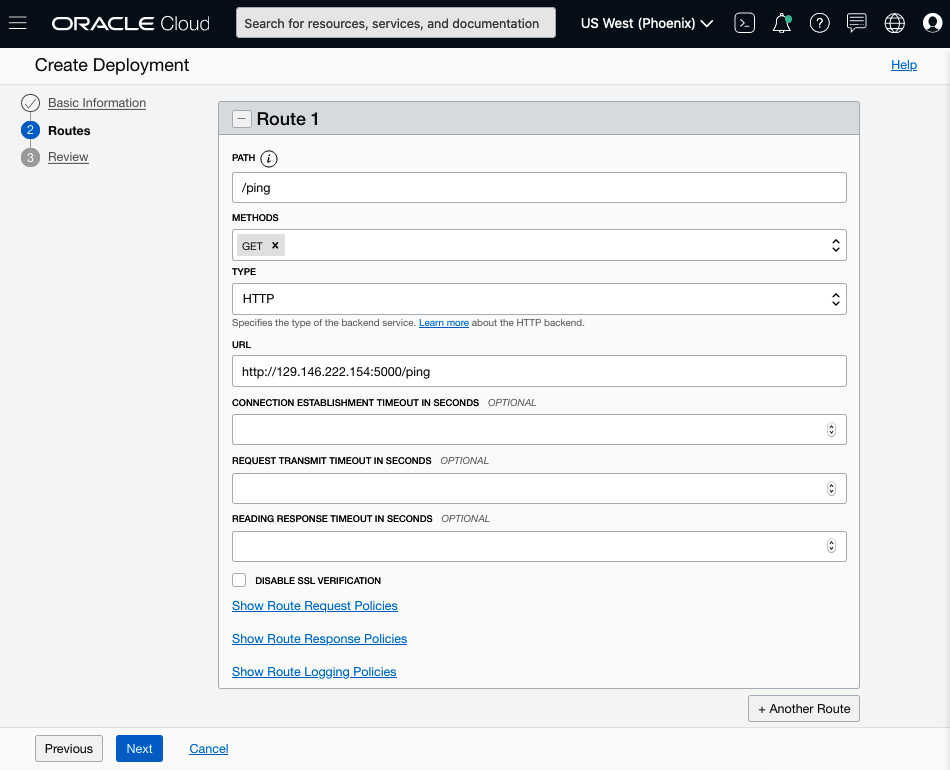


Figure 7.34 – Defining routes for the deployment

1. Optionally, you can add another route. To do that, click on **Another Route** to enter details for additional routes.
2. Click **Next** to review these details.
3. Click **Create**.

With that, you've just finished creating a deployment that will help you access the application endpoint through an API gateway. Now, let's learn how to access the endpoint.

**Accessing the API endpoint through an API gateway**

Once your deployment has been created, you just need to get the endpoint from the deployment and access the URL with the path that you set on the deployment. Let's get started:

1. Once you login to the OCI console, click on the navigation menu and click on the **Developer Services**and then select **API Gateway**. It will take you to the **Gateways** page.
2. From there, select the hyperlink of the gateway that has been deployed and it will take you to the **Gateway Details** page.
3. Click on the **Deployments** from the **Resources** list.
4. On the row of your deployment that you have just created, check the **Endpoint** column and click on **Show link**.
5. This will show you the endpoint's URL. Copy this to a web browser of your choice and append the route path that you set in the deployment. In this case, it was set to **/foo**, so the URL will be **https://<Endpoint URL>/foo**. This can be seen in the following screenshot:

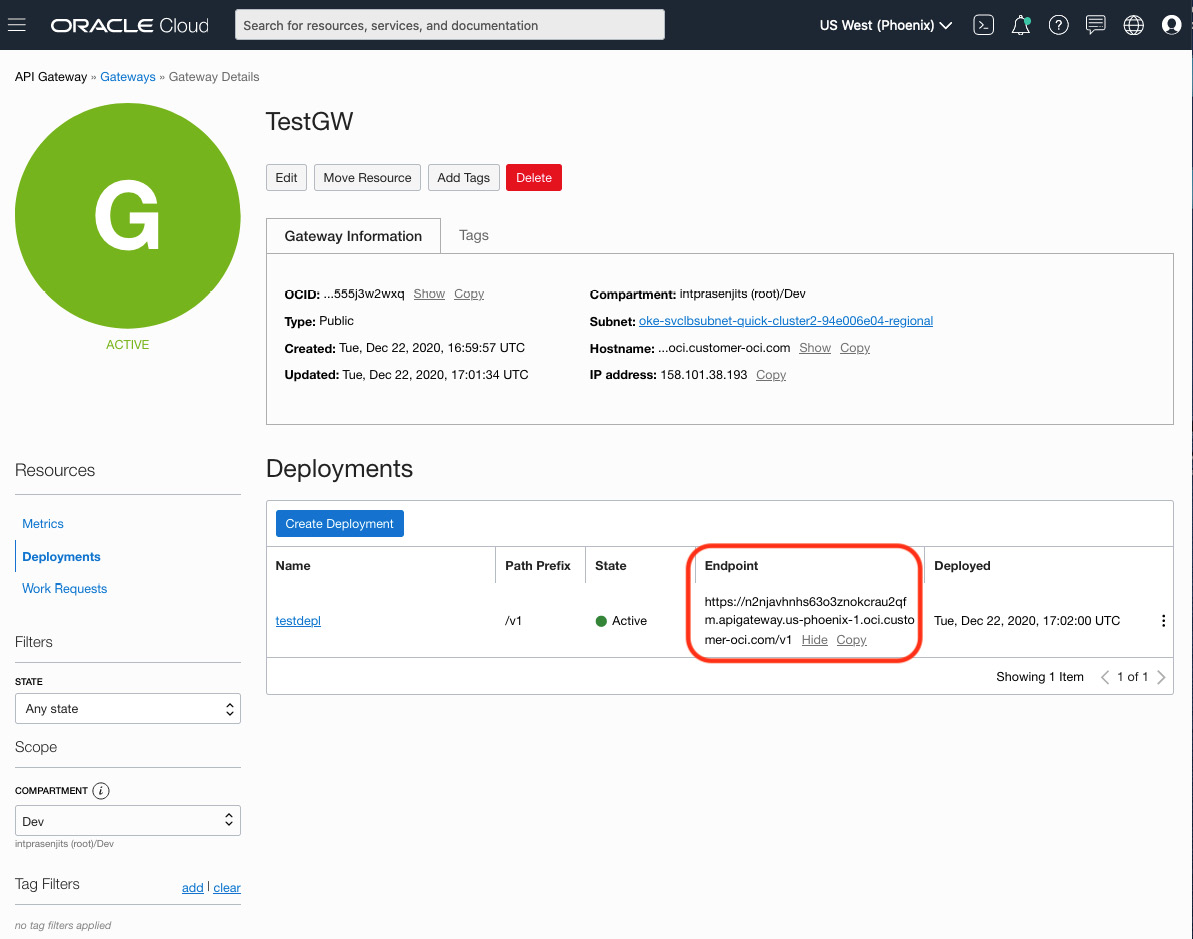


Figure 7.35 – Accessing the API endpoint

With that, you've seen how quickly you can expose an application endpoint based on your choice of topology and also control access to it using authentication, authorization, and request and response policies.

**Summary**

In this chapter, you learned about the nuances of cloud-native application development, and also why large IT enterprises are adopting more cloud-native approaches for developing their next version of an application or refactoring their old monolithic applications. You also learned about how Oracle is shipping cloud-native services, such as OCIR, OKE, and the API gateway. Throughout this chapter, you have learned how to create a container image and store it in a registry, deploy it on top of a managed Kubernetes service, and then expose that application using an API gateway.

In the next chapter, you will learn how to use a small and fractional compute to satisfy on-demand and serverless computing using the robust Oracle functions service.