**3**

**Cloud Native Services on Oracle Cloud Infrastructure**

The Cloud Native Computing Foundation, a part of the nonprofit Linux Foundation, is the premier body whose charter is to foster the development of open-source and vendor-neutral cloud native projects. According to the Cloud Native Computing Foundation (CNCF):

Cloud native technologies empower organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds. Containers, service meshes, microservices, immutable infrastructure, and declarative APIs exemplify this approach.

Cloud native applications are defined by their resilient design, capability to elastically scale using cloud platforms, and capacity to implement efficient lifecycle management that leverages observability and automation. The technology platforms and tools that enable developers to build cloud native applications are generally termed *cloud native technologies*. The fundamental outcome and goal of the cloud native paradigm is to dramatically improve software development velocity, thereby creating innovation that disrupts traditional business models.

Several turnkey technologies and patterns characterize modern cloud native applications, including container orchestration with Kubernetes, the use of service meshes and observability, and stream- and event-based service architecture. This chapter introduces the various services in Oracle Cloud Infrastructure (OCI) that developers can use as their fundamental building blocks when building cloud native applications. The chapter covers the role each of these services plays within the cloud native paradigm so that you get a panoramic view of the cloud native ecosystem that OCI offers. OCI’s fundamental focus on openness also means that these services are compliant with open standards or are built on top of industry-standard open-source platforms and are interoperable with them.

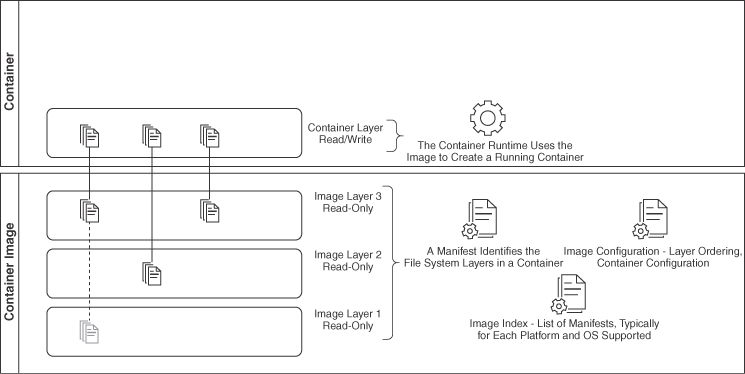
**Oracle Container Image Registry**

Containers are a fundamental turnkey technology that enables the creation of cloud native applications. Containers provide the capability to bundle applications and their dependencies into packages called *container images*. Container images contain the application code and its required dependencies, runtime, system settings, tools, and libraries. This packaging format makes it easy to quickly transmit application containers and run them in dynamic environments while providing a consistent execution environment for the application. Docker originally developed the container image specification and runtime. For this reason, containers and Docker are synonymous for most users. Docker donated the container image specification and the runtime to the Open Container Initiative[**1**](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch03.xhtml#ref3_1a) to help establish open standards as the container ecosystem evolves.

**Note**

This book uses the abbreviation OCI to mean Oracle Cloud Infrastructure and calls out uses of the abbreviation for Open Container Initiative when discussing container images.

At its core, a container image is a directory of files with associated metadata. The container image format[**2**](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch03.xhtml#ref3_2a) defines the *layers* in a specific order that make up this directory. These layers are combined with *union mounting*, a way to combine the contents of various directories, to create a directory that seems like it contains the combined contents. The order of the layers is important because the layers are overlaid on top of each other. When the same files are present on multiple layers, the file on the upper layer overwrites (or deletes) the same files on the lower layer. When creating the image, each layer that makes up the image is archived into a tar ball and compressed with GZIP. A container image *manifest* describes the various layers of the image (in their respective order) and additional metadata such as the OS and architecture. A container runtime can take this package, the container image, and create an isolated execution environment for the application contained in the container image. The container image provides all the information for a container runtime, such as the manifests to identify the file system layers, an index that provides a list of manifests for various platforms, configuration documents that describe image ordering, and more. Using this information in the container image, a container runtime can obtain all layers and configure a running container. [Figure 3-1](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch03.xhtml#ch03fig01) shows this model, with an image made up of several layers of files and described by manifests and configuration documents, while a container runtime uses the information in an image to create a running container.



**Figure 3-1** Container/Container Image Model

Container images need to be stored in a repository, where they can be requested, or “pulled.” Registries provide an address for a container image—a URL that uniquely identifies a container image, called a *container reference*. Oracle Cloud Infrastructure Registry (OCIR, or Container Registry) is a managed service that provides a container image registry to store, serve, and manage container images. OCIR supports multiple image formats or specifications, including Docker V2 and the Open Container Initiative image spec. This allows OCIR to work with several container image specs and container runtimes that support these image specifications. This means that, when using OCIR, you can work with all standard container tools, such as Docker, Podman, cri-o, and containerd. Additionally, the service supports manifest lists, also known as image indexes in the Open Container Initiative specification. Manifest lists allow a single container reference to represent multiple forms of an image (multiple manifests). Each of these manifests is typically associated with a specific OS or architecture so that a container runtime can pick the manifest based on the platform that it’s running on. This is crucial when working with images that are built for different CPU architectures, such as amd64 (x86) and arm64 (ARM).

A typical image reference on OCIR looks like this:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch03_images.xhtml#f057-01a)

iad.ocir.io/idi2cuxxbkto/demo-site:1.0.1

Breaking this image reference into its parts, you can see the following:

* **The registry URL:** iad.ocir.io.
* **The namespace:** idi2cuxxbkto. OCI provides every tenancy with a namespace, and this is shown when you create a repository.
* **The repository:** demo-site.
* ■ **The tag:** 1.0.1. Tags point to the digest of the image manifest. A tag is a more human-readable pointer to an actual digest.

The Container Registry provides a manifest (or a manifest list) at this location for container runtimes to pull the image (its layers and its metadata). You can also use the docker manifest command to inspect the manifest (or manifest list) available at a container reference:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch03_images.xhtml#f058-01a)

$ docker manifest inspect iad.ocir.io/idi2cuxxbkto/demo-site:1.0.1

{

"schemaVersion": 2,

"mediaType": "application/vnd.docker.distribution.manifest.v2+json",

"config": {

"mediaType": "application/vnd.docker.container.image.v1+json",

"size": 9439,

"digest": "sha256:8bdcd2821a78c7bf91dffe1d3f0380cd6c977efe0214c0bc6962

7611e7205881"

},

"layers": [

{

"mediaType": "application/vnd.docker.image.rootfs.diff.tar.gzip",

"size": 208,

"digest": "sha256:b1c13aac26c6d0816d720f6afed6292bde309137d4894819d3

c6e49265490c8c"

},

{

"mediaType": "application/vnd.docker.image.rootfs.diff.tar.gzip",

"size": 2009946,

"digest": "sha256:c3c2acf3bfb91ca8a0220d3d411f8f91f92bc3725fd617b5f7

d962c5d06beb91"

}

]

}

**Working with OCIR**

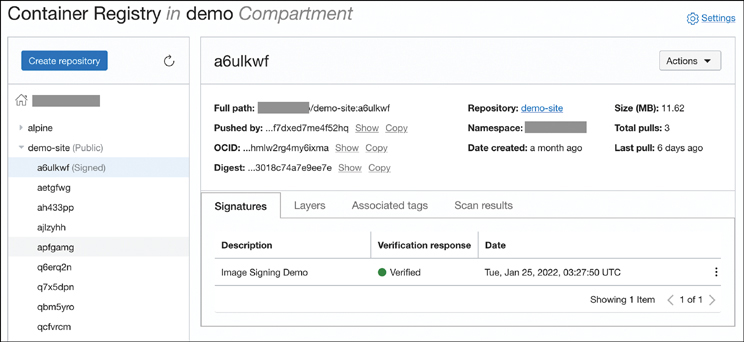
As a fully managed service, OCIR ensures that you can reliably store and serve your container images without managing or scaling the storage and other infrastructure resources typically required for running and operating a container image registry. OCIR supports both public and private access to container images managed by the service. To expose an image publicly, you do not need any resources (such as an Internet gateway or a load balancer) in your tenancy because the service is fully managed. When working with private images, you can efficiently access the registry through a service gateway in your virtual cloud network (VCN), which ensures that the resources that access the images can do so in a highly efficient and secure manner, completely within the OCI network fabric. If an image is exposed publicly, the image can still be accessed over a service gateway by resources inside OCI for better throughput, and the image will also be publicly available over the Internet. Private registries are ideal in environments where Internet access is disallowed for security reasons.

**Image Signing**

The ease of creating and distributing container images comes with an increased exposure to security vulnerabilities. Container images provide a way to build software packages in layers, with developers building on top of popular open-source images that provide basic functionality and runtimes (such as Java) and then laying down their application binaries and artifacts on top of these base layers to create the final image. From a security perspective, developers need to ensure not only that their own software is secure, but also that every layer in their final container image, including the layers that they built on top of, are secure and free from vulnerabilities. They also must consider the possibility of image tampering, in which a malicious actor injects security vulnerabilities into an image. Therefore, to ensure a secure container supply chain, you must ensure that the container images (and each layer that makes up an image) are free from vulnerabilities and that you can verify that the image came from a trusted source and has not been tampered with.

To ensure the provenance of container images that you use in your environments, the OCIR service supports image signing. Image signing is a way to confirm that the container images you are deploying come from a trusted source and to verify that these images have not been tampered with. With OCIR, you can use asymmetric cryptographic keys to ensure the authenticity of an image’s origin and guarantee its integrity. The developer (or the CI process) building the image pushes the image to OCIR. This creates the image in OCIR and assigns an OCID for the image. Now the developer can sign the image with a master encryption key stored in the OCI Vault service and associate the signature with the image’s OCID. For better security, OCIR supports only asymmetric key algorithms, such as RSA or ECDSA; symmetric key algorithms such as AES are not supported.

When a client such as a Kubernetes cluster validates the signature, the Vault service is checked to ensure that the signature is valid. The service also ensures that the user who pushed the image into the registry had access to the master encryption key at the time the image was signed. The signature is based on the content of the image; any tampering or changes to the image invalidate the signature. This gives users or systems pulling a signed image from OCIR confidence that the source of the image is trusted and that the image has not been tampered with. To further ensure security within the container image supply chain, other OCI services, such as Oracle Cloud Infrastructure Container Engine for Kubernetes (OKE), can be configured to accept only signed images. [Figure 3-2](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch03.xhtml#ch03fig02) illustrates the signatures for an image in OCIR and shows the results of a signature verification. After an image in OCIR has been signed, the signature can be verified at any time.



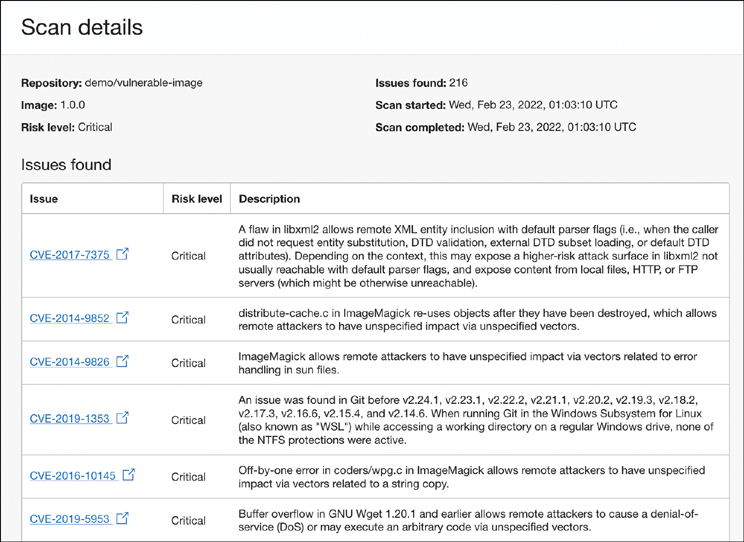
**Figure 3-2** OCIR Showing a Repository with Signed Images and Image Signature Verification

**Image Scanning**

OCIR also supports image scanning to identify vulnerabilities in your container images. These vulnerabilities could be within the application you are developing or within the tools or runtimes that exist on the base layers of the image you are building upon. Image scanning can be enabled on a per-repository basis by adding a scanner to the repository. The scanner looks for vulnerabilities published in the publicly available Common Vulnerabilities and Exposures (CVE) database.

The scanner is powered by the Oracle Cloud Infrastructure Vulnerability Scanning Service, and it also provides a Vulnerability Scanning REST API that you can integrate into your CI pipeline so that a build pipeline can build an image, push it to an OCIR repository, and programmatically get scanning results. This enables you to integrate image scanning into your development pipeline so that you can identify images with vulnerabilities as early as possible and then create CI/CD workflows that can prevent the images from being promoted to critical environments. [Figure 3-3](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch03.xhtml#ch03fig03) shows the result of an image scan, summarizes the issues found for the image, and gives details on each CVE that has been identified.

When enabling the scanner on a repository that already has images in it, the most recent four images are scanned immediately. The scanner produces a report showing an overall risk assessment for the image based on the scan, as well as the individual vulnerabilities found and their risk levels, with pointers to the CVE database for more information on the vulnerability. OCIR automatically scans the images as new vulnerabilities are added to the CVE database and retains the scan results for a period of 13 months so that the risk level of the image can be assessed over time.



**Figure 3-3** Scan Results for an Image Showing the CVEs That Are Identified on That Image, with Risk Levels, Descriptions, and Links to the CVE Database

**Creating Containers from Images**

After a container image has been created and pushed to a container image repository, the next logical step is to create a running container from this image. A container runtime, or a container engine, is the software that can run containers on a host operating system.

**Container Runtime vs Container Engine**

Although the two terms are used interchangeably in many situations, they have subtle but important differences. A *container runtime* simply manages the creation and management of a container. On Linux, this includes making the system calls to create and configure the kernel features that enable resource limiting, process isolation, and more. However, this does not include capabilities such as pushing/pulling images from remote repositories. These container runtimes are sometimes known as low-level runtimes; examples include runc and crun. A *container engine*, on the other hand, comprises more tools and utilities, including CLIs for managing containers, pulling and pushing images, and so on. These are also called high-level container runtimes; examples include containerd and cri-o.

The container runtime uses the host operating system kernel’s capabilities to create isolated sandboxes for processes with resource limits. The container runtime uses the metadata in the image to configure the isolated execution environment that has been created using the underlying kernel features, union mount the image layers, and configure the mount as the root file system for the isolated environment. Typically, this is done through operating system–level isolation and virtualization, such as with cgroups, namespaces, and chroot in Linux, or Hyper-V in Windows.

Oracle Cloud infrastructure offers several choices to run and manage container workloads, from the basic approach of running and managing containers on compute instances to fully autonomous and serverless offerings.

**Compute Instances**

The most obvious and trivial way to run a container on OCI is to create a compute instance, install a container runtime on that instance, and then use the tooling provided by the container runtime to create and manage containers. Although this is a perfectly valid model, it often involves more (and often unacceptable) management overhead for the developers because of the need to keep the container runtimes, tools, and other infrastructure components updated and patched on a rigorous schedule. However, this approach affords you the highest amount of control in managing your workloads.

The trivial method for booting a compute instance with a container runtime is to install it at first boot using cloud-init[**3**](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch03.xhtml#ref3_3a). [Listing 3-1](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch03.xhtml#list3_1) shows an example cloud-init configuration for Oracle Linux 7, to install Docker, enable the service, and start it. The example also adds the default opc user to the docker group so that this user can use the docker command without using sudo.

**Listing 3-1** cloud-init Example for Bootstrapping an Oracle Linux 7 Instance with a Container Runtime

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch03_images.xhtml#f062-01a)

#cloud-config

bootcmd:

- [ cloud-init-per, once, enable-epel, yum-config-manager, --enable, ol7\_

developer\_epel]

groups:

- docker

users:

- default

- name: opc

groups: docker

shell: /bin/bash

sudo: ALL=(ALL) NOPASSWD:ALL

packages:

- docker-engine

- docker-cli

runcmd:

- [ systemctl, daemon-reload ]

- [ systemctl, enable, docker.service ]

- [ systemctl, start, --no-block, docker.service ]

**Note**

On Oracle Linux 7, the package *docker-engine* refers to the Oracle Container Runtime for Docker. This package is based on the upstream docker releases.

Oracle Linux 8 does not feature the Oracle Container Runtime for Docker and instead uses Podman, Buildah, and Skopeo, which is a set of container tools based on the Open Container Initiative. All the tools are available conveniently in a single module that can be installed with the following command:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch03_images.xhtml#f063-01a)

sudo dnf module install container-tools:ol8

This command can be used from within the cloud-init configuration as well.

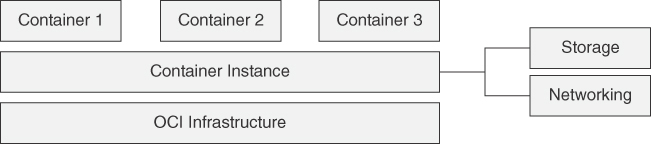
Aside from using cloud-init to set up required packages, you can also create custom OS images with the tools preinstalled. This approach avoids the installation process at instance creation time. It saves several seconds when launching an instance, which can be quite significant if you have highly performance-sensitive workloads that frequently need ephemeral compute instances. Several task-based workloads belong to this category. When using this approach, an instance is created and then the required packages and settings are configured. A custom OS image is created from this instance, which now includes the packages and customizations that were applied to the instance. New instances can thus be created using the custom image as the OS image.

**Container Instances**

Container instances address the primary drawback of running a container runtime directly on top of infrastructure, which is the high setup and maintenance overhead associated with it. Using the methods described in the previous section, the setup process and methods can be streamlined to a certain extent. However, the maintenance of infrastructure poses an entirely different challenge. Using infrastructure directly, developers need to take on the responsibility for routinely updating, patching, and rebooting their compute instances. It is also important to keep the container runtime up to date with the latest patches and CVEs while ensuring consistent configurations of these runtimes. Sizing the infrastructure becomes another challenge in this model because you need to always ensure compute capacity for container workloads to scale dynamically while still optimizing for cost. Instance provisioning times can be orders of magnitude higher than container creation and startup times, so always having “just enough headroom” is essential to seamlessly scale the container workload. Developers also need to ensure that logs and metrics from containers can be collected and pushed to analytics tools. Workload isolation is an equally challenging problem, particularly for multitenant applications and SaaS platforms. When building multitenant applications and platforms, the shared infrastructure needs to be managed carefully to prevent data leakage and container escape attacks. These often result in solutions that require a significant amount of custom code to ensure that containers are placed in optimally sized instances, containers have room to scale when needed, shared resources are well isolated, and the fleet can be managed and patched efficiently from an operational perspective.

Container instances address these concerns by offering a service that enables you to create one or more containers without managing infrastructure. The experience is like compute instance creation, in that it enables you to specify the CPU, the memory, network, and other resource characteristics required for one or more containers and then for providing container images to run. OCI provides the compute, the container runtime, and other resources, such as networking and storage; then it uses the metadata provided to pull the images and create a running container or set of containers. Here the OCI service takes care of creating and maintaining the underlying infrastructure. The service manages activities such as OS patching and restarts, container runtime setup, network setup, storage attachment, and so on. This greatly simplifies the workflow for developers while addressing the drawbacks of the traditional approaches. From the developer’s perspective, the workflow is very similar to launching a compute instance. Instead of providing an operating system image, the developer provides the CPU memory and other resource constraints, as well as the container images that need to be part of the container instance.

A container instance is a lightweight container-optimized VM that can have more than one container in it. This enables developers to start containers much faster than provisioning VMs while providing the same level of hypervisor-level isolation and avoiding the management overhead of traditional VMs. The hypervisor level allows for a better security posture, even in the face of container escape attacks. The containers within a container instance all share the CPU, network, and storage resources. This is somewhat like a *pod* in Kubernetes, although a container instance should really be thought of as a lightweight VM that can run one or more containers; it differs greatly in the level of container orchestration features when compared to platforms such as Kubernetes. [Figure 3-4](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch03.xhtml#ch03fig04) shows how the container instance provides hypervisor level isolation and an environment that can run multiple containers that share the container instance’s resources.



**Figure 3-4** Container Instance and Hypervisor-Level Isolation

Container instances integrate with OCI features such as instance pools and autoscaling for fleet management. This means that you can create a container-based workload fleet that is consistently configured and can scale elastically. The container instance also makes it easy to access container logs and metrics for each container within the container instance and execute commands within the containers. As with containers, container instances are also immutable. When a container instance is created, changing resources such as CPU or storage is done by creating a new container instance and discarding the old one. This includes updating the image tags and changing the configurations for the containers, in keeping with standard container lifecycle management practices. This dramatically improves the workflow for developers working with container applications by providing a fully managed platform for infrastructure and container runtimes.

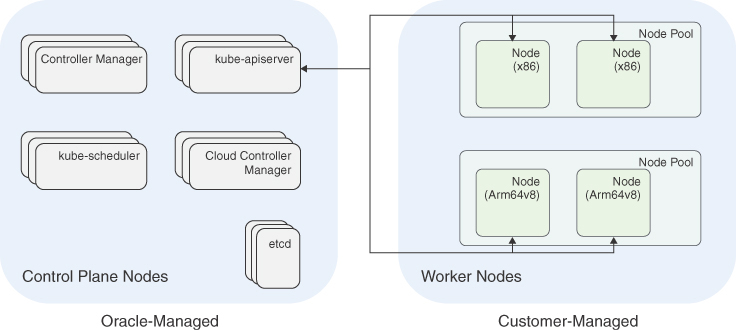
Ideal applications for container instances include data processing jobs such as video encoding or data analytics, build jobs, CRUD applications, event-based actions, and task automation.

**Container Engine for Kubernetes**

As containerized applications grow in scale, they tend to become smaller and more distributed. Modern distributed applications are designed as a network of microservices that each implement a specific feature and communicate with each other using well-defined interfaces. When combined with container-based packaging, this design paradigm enables each of these smaller services to scale, update, and expand in their features, independent of each other. This also helps increase the overall development velocity and supports a more frequent release of smaller changes. As the number of these containers rises, however, so does their management overhead; the overall application also increases in its complexity. Manually or statically wiring together the containers and keeping track of their status and health soon becomes an untenable approach. This calls for more automation to orchestrate the container workloads. For large container-based workloads that require significant higher-level abstractions and orchestration, Kubernetes is the platform of choice. Kubernetes offers much beyond container management, and it provides features such as autoscaling, resource management, service discovery, load balancing, and deployment management. Kubernetes is a CNCF graduated open-source project and can be installed and run on public cloud, hybrid, or on-premises infrastructures.

Kubernetes exemplifies the “pets versus cattle” approach. The premise is that you do not treat your infrastructure as individual hosts or resources, each with a designated purpose and name, like a pet. Instead, you see your infrastructure as a fleet of servers, with none serving any special role and all being completely replaceable. For instance, with Kubernetes, you can configure an application container so that it is allowed to use two cores and 8GB of memory to run, and you can request that three instances of the application be running at any one time. Using this configuration, called a *manifest* and typically represented in YAML, Kubernetes can create the required number of containers to meet your specification and keep track of their health. Kubernetes can move your containers as the fleet’s status changes and failures occur, all without intervention. In this manner, Kubernetes enables you to describe the configuration you desire in the deployment manifests; you can then apply these manifests to a Kubernetes cluster that keeps track of the containers, nodes, and other resources and ensures that your configuration defined in the manifest is always met. Because these manifests can be versioned, releasing new changes and rolling back to previous configurations becomes trivial for most applications.

Oracle Cloud Infrastructure Container Engine for Kubernetes (OKE) is the managed Kubernetes platform for developing modern applications from Oracle. Although you can install Kubernetes on any infrastructure yourself, the installation and upkeep of the administrative and platform components in Kubernetes can be challenging. [Figure 3-5](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch03.xhtml#ch03fig05) illustrates the various components of an OKE cluster.



**Figure 3-5** Components in an OKE Cluster, Showing the Oracle-Managed Control Plane and the Data Plane Where the User Has Full Control

As a software suite that manages container-based workloads, Kubernetes has a set of administrative components that manage and control the cluster for tasks such as keeping track of the nodes, workloads, configurations, health status, and so on. The nodes on which these control components run are called the *control plane nodes*. In a managed Kubernetes platform such as OKE, these are installed and managed by the cloud provider. The control plane nodes do not run any workloads other than the management processes for the cluster itself. Users do not have access to these nodes.

The workloads themselves run on compute instances called worker *nodes*. The cluster control plane processes monitor and record the state of the worker nodes and schedule workloads onto them. A *node pool* is a subset of worker nodes within a cluster that all have the same configuration. A cluster must have a minimum of one node pool, but a node pool need not contain any worker nodes.

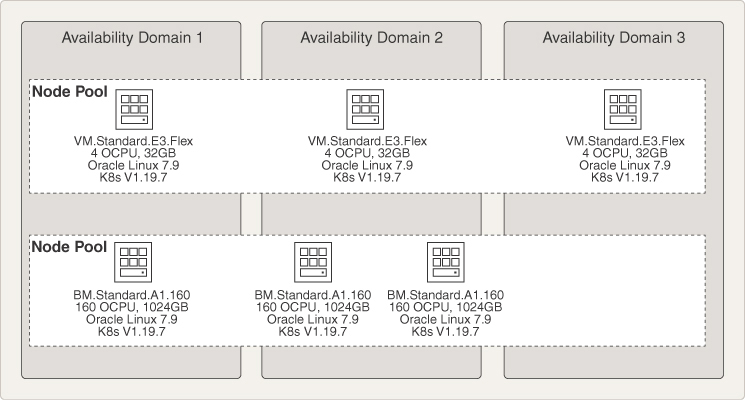
OKE supports two types of node pools that differ in how the nodes in the pool are managed. *Managed* node pools have nodes that are controlled by the user. *Virtual* node pools, on the other hand, are fully managed by OCI. Managed node pools and virtual node pools address different use cases and usage models. A managed node is a compute instance of the user’s choice of shape. Users have full access to these nodes, including SSH access and the capability to customize the nodes with user-created OS images and cloud-init scripts. Nodes run the kubelet process, which is responsible for ensuring that the pods scheduled on the node are running and reporting on the node health conditions acting as a node agent for Kubernetes. Virtual nodes, on the other hand, leverage Kubernetes pod configurations to create an isolated compute environment for the pod. Each Kubernetes pod is therefore isolated from other pods at a hypervisor level. The configuration of the execution environment, such as the number of CPU cores and memory, is inferred from the resource requests and limits set on the containers in the pod configuration. The execution environment for the pod is fully managed by Oracle and runs abstracted, away from the user. Virtual node pools therefore completely remove the need to manage infrastructure when deploying Kubernetes workloads and can be considered to be a serverless Kubernetes platform. Although managed nodes give users a high degree of control in accessing and managing their nodes (as with using custom cloud-init scripts to customize nodes), they come with the additional overhead of managing the node’s OS and Kubernetes upgrades. Virtual nodes, on the other hand, offer an experience that is focused on your workload, with little or no infrastructure management overhead. However, that comes at the expense of having control over the configuration of nodes. A single cluster can have both provisioned and virtual node pools.

[Table 3-1](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch03.xhtml#ch03tab01) offers a comparison of managed and virtual nodes.

**Table 3-1** Managed versus Virtual Nodes

|  | **Managed Nodes** | **Virtual Nodes** |
| --- | --- | --- |
| **Infrastructure control** | Users maintain control over nodes. | Users can control the workload but not the infrastructure. |
| **Upgrades** | Users upgrade the nodes. | Upgrades are fully managed by OKE. |
| **Isolation** | A node’s resources are shared by the pods that run on it. | A virtual node has no physical resources. Each pod runs in its own hypervisor-level isolated compute environment. |
| **Resource management** | Users decide the shapes of the nodes and set resource requirements and limits for pods. The Kubernetes scheduler matches pods to nodes based on availability. | Nodes need not be created or managed. Users should set resource requirements and limits on pods, to create dedicated compute environments for each pod. |

Node pools can also have placement configurations that control the placement of the nodes in the node pool. These placement configurations can be used to spread the nodes in a node pool across multiple availability domains or fault domains to ensure better resiliency. Creating multiple node pools enables you to create groups of machines within a cluster that have different configurations. [Figure 3-6](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch03.xhtml#ch03fig06) shows two node pools, one with E3.Flex shapes and another with A1 bare-metal machines. Here, the first node pool is based on AMD (x86)–based Flex shape virtual machine instances; the second node pool is an ARM-based bare-metal shape. This example also demonstrates that different node pools can be of different shapes, CPU architectures, and bare metal or virtual machines. Having this flexibility in your cluster resources lets you right-size the workloads and progressively introduce infrastructure changes to your environments—for example, introducing ARM-based compute for a subset of workloads or using bare-metal or GPU-enabled nodes for compute-heavy workloads and VMs for supporting workloads. Node pools also let you control the placement of nodes across availability domains and fault domains in OCI. [Figure 3-6](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch03.xhtml#ch03fig06) shows the first node pool placing nodes across all three availability domains and the second node pool restricting nodes to just two of the three availability domains.



**Figure 3-6** Node Pools in a Cluster Can Be Used to Control the Node Types and Their Placement, as Well as Create Clusters with Multiple Types of Nodes in Separate Node Pools

The node pools act as the control unit for scaling and can be used to scale the number of compute instances up or down, to add or remove compute capacity in the cluster. The scaling also can be automated based on metrics. Autoscaling is covered in more detail 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).”

Note that the number of pods that can be scheduled or placed on a node is still dependent on the network address space available on a node, up to a maximum of 110. Larger nodes with more CPU cores and memory are therefore ideal to accommodate pods with much higher resource consumption needs. The memory and network throughput are also important considerations when choosing a shape for your nodes. The maximum available memory and network bandwidth changes, based on the shape of the node and the number of OCPUs (an OCPU is a complete core, not just a hardware thread). Thus, the choice of shape for the nodes also depends on the memory and network throughput expectations for the workloads.

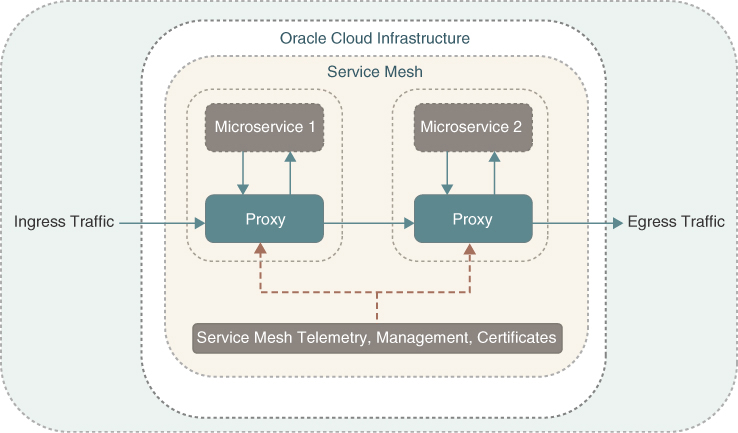
As one of the highest-velocity open-source projects, Kubernetes provides support for three minor versions. This support policy is sometimes also called an N-2 support policy, in which the latest version and the two preceding minor versions of Kubernetes get patches for security and bug fixes. OKE as a managed service does not force users to upgrade as new versions of Kubernetes are released, although keeping your Kubernetes version up to date with the latest security fixes and bug fixes is an important consideration. For creating new clusters, OKE always supports at least three versions of Kubernetes. Version choice for new clusters moves like a rolling window as well. When OKE adds a new version of Kubernetes as a choice for creating new clusters, the oldest version choice remains a choice for at least 30 days, beyond which it may be removed. Exiting clusters that use that version are unaffected; the removal simply means that new clusters will have newer version choices. As support for new Kubernetes versions is added to OKE, you can update the control plane to the new version with a click of a button or an API call. The control plane upgrades are completely managed by Oracle and are transparent to the user. The Oracle-managed control plane is always in a highly available configuration, and the upgrade is performed in a rolling fashion so that it does not impact the cluster’s normal operations. After the control plane has been upgraded, the node pools can be upgraded as well. [Chapter 4](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04.xhtml#ch04) does a deep dive into OKE, providing best practices, strategies, and tips for building and deploying applications to OKE.

**Service Mesh**

As we become more accustomed to microservice-based architectures and distributed applications, we start seeing applications as consisting of services that communicate with each other and forming a network of services that implement the application. This design paradigm of creating a network of services affords us a lot of advantages in flexibility, development velocity, and resiliency. However, as applications evolve, the network of microservices grows and the complexity of managing the entire application also increases. The task of ensuring reliable and secure service-to-service communication and implementing observability can have a significant impact on the development of the services. Building these features directly into the services makes them more brittle, impedes their capability to change, and slows the development of features because the application code now needs to take on additional responsibilities. In a cloud native environment, a service mesh is a tool that can be used to add cross-cutting functionality, such as security or observability, to a set of microservices.

A service mesh operates by inserting proxies between services into the network of microservices. These network proxies are typically deployed as a set of sidecar containers, or containers that are deployed alongside microservice containers. They handle the service-to-service communication between the microservices and can transparently implement security, observability, and patterns for resiliency.

Oracle Cloud Infrastructure Service Mesh is a fully managed service mesh implementation that provides security, observability, and traffic management to cloud native applications without any application changes or dependencies. OCI Service Mesh creates proxies that are containers deployed alongside your applications, in the same Kubernetes cluster. The proxies handle traffic to the application and provide telemetry, security, and load balancing across pods. The proxies communicate with each other and are aware of policies that govern communications so that they allow only permissible communications between services. [Figure 3-7](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch03.xhtml#ch03fig07) shows how the proxies deployed as sidecars to microservice containers communicate with the managed components of the service mesh to provide features such as telemetry and security.



**Figure 3-7** OCI Service Mesh Is Based on Proxies That Are Injected as Sidecar Containers to Your Pods

The service mesh has managed components that include an IngressGateway, VirtualServices, and VirtualDeployments, along with policy management to support customized and secure traffic routing. These are mapped to an application’s services and provide abstractions for those services. When using the OCI service mesh, the applications services are mapped to virtual services and virtual deployments and associated policies. These virtual services are then bound to the existing services using binding objects. The applications are exposed using the IngressGateway, which routes traffic to the various VirtualServices. VirtualServices are in turn bound to Kubernetes services that the application exposes. A virtual service represents a customer-managed microservice in the mesh. Each virtual service has its own configuration for the service hostname, Transport Layer Security (TLS) certificates (for both client and server), and Certificate Authority (CA) bundles. A virtual deployment represents a version of a virtual service; each virtual service has up to five virtual deployments. Route tables are a virtual service feature that routes ingress traffic to specific versions of the virtual service. A virtual deployment binding associates the pods in an application cluster with a virtual deployment in your mesh. The virtual deployment binding resource allows service mesh to discover pods, backing the virtual deployment for service discovery.

The service mesh components and custom resources are installed on a Kubernetes cluster using the OCI Service Operator for Kubernetes (OSOK). OSOK is a collection of operators for OCI services that includes the operator for the service mesh. This operator provides the CRDs, roles, and other resources required to allow users to perform actions on the OCI service mesh using the Kubernetes API. With the operator installed, users can interact with the service mesh using standard tools such as kubectl.

When using the service mesh with the operator, users can apply annotations at the namespace level to enable sidecar injection for all pods in the namespace. The effect of these namespace-level annotations can be overridden at a pod level by specifying the sidecar injection on a per-pod basis.

**Serverless Functions**

Serverless functions are at the pinnacle of building scalable and distributed business logic implementations. They are called *serverless* simply because the server and its runtime environment are fully managed by the cloud vendor and are not exposed to the application developer. The essential idea is to create functional units of business logic that can be packaged and executed in isolation. These are generally focused, well-defined tasks such as file processing, in which a file is read from a source, minimally processed or transformed, and then either sent to a destination or takes an action in response to an event. Functions also find use in Internet of Things (IOT) applications, image processing, machine learning (ML) inferencing, and other applications. Functions usually do not exist in isolation; many times, they are chained together to create a network of functions (similar to microservices in that respect) that interact with external systems. Functions are usually time bound: They are expected to finish their execution within a slice of time, beyond which they can be terminated. In many cases, functions are also event driven. Their fundamental design of running small, time-bound processes that perform well-defined actions makes them naturally scalable and distributed when assembled into larger systems. Functions can be written in Python, Go, Java, NodeJS, and other commonly used programming languages and runtime environments. Functions should be fully self-contained and cannot depend on any outside software or code to operate other than calling other APIs. Being packaged into these self-contained units allows them to start up, execute their functionality, and then shut down quickly. Functions abstract all infrastructure management from their users, and a cloud platform (such as OCI) guarantees a secure execution environment that can scale quickly as the calls to the function increase. This also means that functions follow a very different cost model based on actual usage rather than the typical infrastructure that might be billed based on resources allocated.

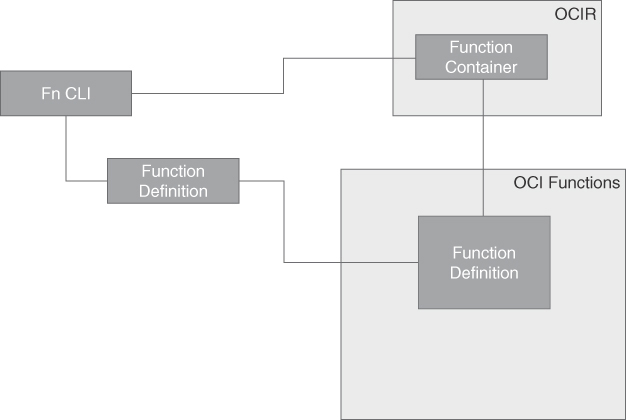
Oracle Functions is a fully managed Functions-as-a-Service platform that is built on enterprise-grade Oracle Cloud Infrastructure and powered by the Fn Project open-source engine. The serverless and elastic architecture of Oracle Functions means there is no infrastructure administration or software administration for you to perform. You do not provision or maintain compute instances, nor are you responsible for operating system software patches and upgrades. Oracle Functions simply ensures that your app is highly available, scalable, secure, and monitored. With Oracle Functions, you can write code in Java, Python, Node.js, Go, and Ruby (and, for advanced use cases, bring your own Dockerfile and Graal VM). You can then deploy your code, call it directly, or trigger it in response to events, and you get billed only for the resources consumed during the execution.

Oracle Functions is based on Fn Project.[**4**](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch03.xhtml#ref3_4a) Fn Project is an open-source, container-native, serverless platform that can be run anywhere—in any cloud or on-premises. Fn Project is easy to use, extensible, and performant. You can download and install the open-source distribution of Fn Project, develop and test a function locally, and then use the same tooling to deploy that function to Oracle Functions. You can access Oracle Functions using the console, a Command Line Interface (CLI), and a REST API. You can invoke the functions you deploy to Oracle Functions by using the CLI or by making signed HTTP requests. Oracle Functions is integrated with Oracle Cloud Infrastructure Identity and Access Management (IAM), which provides easy authentication with native Oracle Cloud Infrastructure identity functionality.

When you have written the code for a function and it is ready to deploy, you can use a single Fn Project CLI command to perform all the deploy operations in sequence:

1. Build a Docker image from the function.
2. Provide a definition of the function in a func.yaml file that includes:
   1. The maximum length of time the function is allowed to execute
   2. The maximum amount of memory the function is allowed to consume
3. Push the image to the specified Docker registry.
4. Upload function metadata (including the memory and time restrictions and a link to the image in the Docker registry) to the Fn Server and add the function to the list of functions shown in the console.

[Figure 3-8](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch03.xhtml#ch03fig08) shows how the Functions CLI interacts with the various components to deploy a new function.



**Figure 3-8** The Components of Oracle Functions

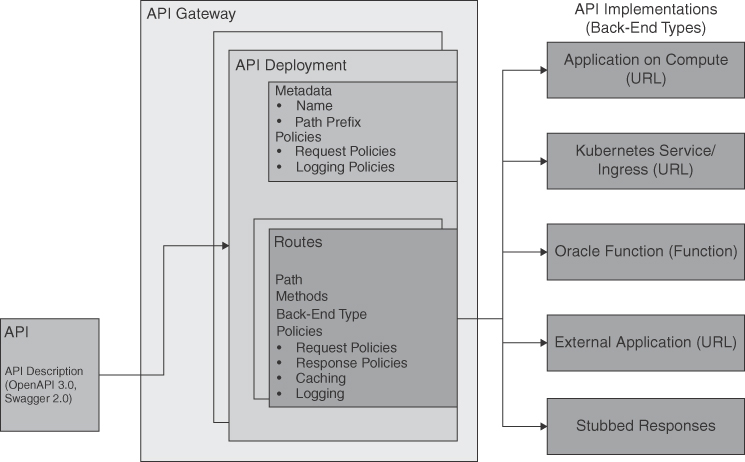
**API Gateways**

An API gateway provides a layer of abstraction for the Application Programming Interfaces (APIs) that your applications expose. For microservice-based applications that operate at scale, this becomes almost indispensable, although, in many ways, an API gateway predates the popularity of microservice architectures. API gateways can be software-defined, self-managed platforms that you deploy on top of infrastructure, or they can be a fully managed service delivered by a cloud provider. Regardless of the model, at its core, an API gateway is a service that provides a facade around one or more of your applications’ APIs, with added features. Common features offered by API gateways include the capability to implement a common and consistent authentication and authorization model across the back-end APIs, rate limiting for APIs, caching, monitoring, API versioning, and the capability to transform both requests and responses.

The API Gateway service in OCI is a fully managed service that is implemented as a virtual network appliance that you can deploy to a regional subnet. Regional subnets are required because API gateways are always highly available, with fault tolerance built in. When deployed in regions with multiple availability domains, API gateways are automatically configured across multiple ADs for fault tolerance. In single-AD regions, an API gateway is configured across fault domains. API gateways can be public (accessible from anywhere on the Internet) or private (accessible from within the VCN). API gateways always have a private endpoint, and this is optionally exposed publicly to create a public API endpoint. You can use a single API gateway to link multiple back-end services and route inbound traffic to them. These back-end services can include HTTP APIs exposed through compute instances, load balancers, external API providers, and OCI Functions.

**Components of an API Gateway**

When working with the API Gateway service, you generally start with the gateway resource. This is the infrastructure component, the virtual network appliance that is managed by OCI. You can then deploy API deployment specifications on this gateway resource. An API deployment specification is a way to describe the back-end APIs as a set of *routes*. A route is the mapping from a path to one or more methods and then a back-end service. Routes capture the type of resources that provide the underlying API and how to reach that resource. For instance, an HTTP URL exposed by an application that you are running on a compute instance captures the private IP address or domain name, the port on which the service is available, and the path under which to expose or present the API and the HTTP operations that the gateway supports for the route. The API deployment spec also describes the policies that can validate and transform the requests or responses. The policies are applied to every request or response. You can also use policies to add authentication, authorization, and monitoring. When the deployment specification is deployed to a gateway resource, it becomes an API deployment. The API deployment causes the gateway to expose the API as defined in the spec and is ready to direct traffic to the back ends described in the spec. As API traffic flows in, the gateway applies the policies that are specified by the API deployment spec. You can add policies to an API deployment specification that applies globally to all routes in the API deployment specification, as well as policies that apply only to particular routes. [Figure 3-9](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch03.xhtml#ch03fig09) shows how an API gateway can contain multiple API deployments with policies and routes that connect it to various back ends.



**Figure 3-9** An API Gateway Can Host Multiple API Deployments, with Routes and Policies That Connect It to Various Back Ends

Apart from resources such as the gateway and an API deployment, the service exposes another resource called the *API*. As shown in [Figure 3-9](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch03.xhtml#ch03fig09), an API resource can be used to create an API deployment as well. The API resource is a representation of an API description in an open format such as OpenAPI 3.0 or 2.0 (also known as Swagger 2.0). An API description like this establishes the public contract for your API, which automatically documents the endpoints, paths, HTTP operations, and type of responses to expect from the API. Representing the API contract in an open format such as OpenAPI 3.0 helps to ensure the portability and tooling for working with these APIs; these industry-standard formats have attracted an established ecosystem of tools around them. The API description format is machine readable and can be processed by tools with which you can generate documentation for the API, generate stubs for clients to call the API without an actual implementation, generate test cases and Software Development Kits (SDKs), and so on. An API resource is created by uploading an API description. The API resource can be deployed to an API gateway, creating an API deployment. When an API resource is deployed to an API gateway, the routes are created from the API description because it provides the paths, the HTTP methods supported, and expected responses. Policies and references to the actual API implementations are added to the API description when creating an API deployment. Creating API resources in the API Gateway service is optional, but it is highly recommended. You can also create an API deployment that does not initially have an API description and then add an API description later.

**Working with the API Gateway Service**

The workflow for using an API gateway is best demonstrated by starting with a simple API. This example shows a minimal API for product data. The API will have methods by which users can request a list of products or get the details of a single product. To build this API, a developer can start with the infrastructure resources and build an ad-hoc deployment. This is done by creating an API deployment from scratch, defining its routes, back ends, and so on. Alternatively, a developer can first define the API contract and then create the infrastructure resources and deploy the API definition to it. The best practice is to use an API-first approach, to define the API and its behavior without focusing on the implementation. This API definition defines an interface that potential consumers can start consuming even before an implementation is created. After all, the implementation for the API simply materializes the behavior described by the API definition with concrete back-end systems. The actual implementation is hidden from consumers and can potentially be swapped out, if needed. This is because the consumers always consume the API through the API Gateway, which maintains the API behavior expressed in the API definition and routes the requests to a back end that implements that behavior. After the API is defined, a developer can create the infrastructure and deploy the API definition to that infrastructure. The example presented here uses OpenAPI Spec 3.0[**5**](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch03.xhtml#ref3_5a) to define the API, and the code snippet in [Listing 3-2](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch03.xhtml#list3_2) shows how an API of this nature would appear.

**Listing 3-2** An Example API Definition Expressed Using the OpenAPI Spec v3 Standard

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch03_images.xhtml#f076-01a)

{

"openapi": "3.0.0",

"info": {

"version": "1.0.0",

"title": "Minimal Product API"

},

"paths": {

"/products": {

"get": {

"summary": "get all products",

"operationId": "getProducts",

"responses": {

"200": {

"description": "An array of products",

"content": {

"application/json": {

"schema": {

"$ref": "#/components/schemas/Products"

}}}}}}

},

"/products/{productId}": {

"get": {

"summary": "Info for a specific product",

"operationId": "getProductById",

"parameters": [{

"name": "productId",

"in": "path",

"required": true,

"description": "The id of the product to retrieve",

"schema": {

"type": "string"

}}

],

"responses": {

"200": {

"description": "Expected response to a valid request",

"content": {

"application/json": {

"schema": {

"$ref": "#/components/schemas/Product"

}}}}}}}

},

"components": {

"schemas": {

"Product": {

"type": "object",

"required": [

"id",

"name"

],

"properties": {

"id": {

"type": "integer",

"format": "int64"

},

"name": {

"type": "string"

}}

},

"Products": {

"type": "array",

"items": {

"$ref": "#/components/schemas/Product"

}}}}

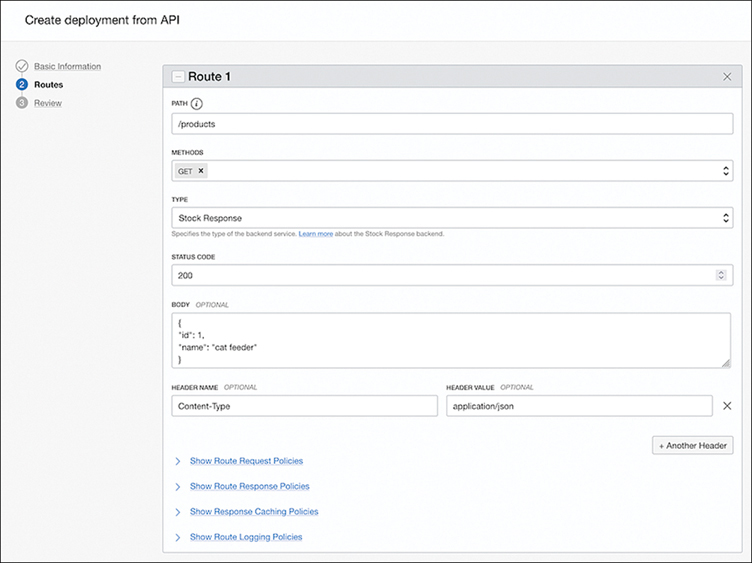
}

OpenAPI definitions always start with the version of the OpenAPI Spec used. Explicitly setting the version is mandatory. Here, the spec version used is 3.0.0. This is followed by some API metadata in the info object. The paths section is one of the most important in the API definition; it defines the various URL paths or routes that the API exposes. The example here shows two paths: /products and /products/{productId}. Each path then defines the HTTP methods that the path will support. For each method, the definition identifies the request parameters and request body, where applicable, as in the case of PUT and POST methods. Paths can contain path parameters such as /products/{productId} or other parameter types, such as query parameters, cookie parameters, or header parameters. Each HTTP method also identifies the various possible response codes and response structures.

**Note**

Several optional elements have been removed; for brevity, the example shows only necessary attributes. The OpenAPI spec website has more details on the various attributes within the specification: <https://spec.openapis.org/oas/v3.0.0>.

With the API defined, the developer can now create an API resource and upload the API definition to it. After the API definition is uploaded, the API resource validates the definition to ensure that it conforms to the specification. Once validated, the API definition is ready to be deployed to an API gateway. The developer can create infrastructure resources such as the gateway at this point or use pre-existing resources. Deploying the API to an *API gateway* creates the *API deployment* resource. During deployment, the service parses the API definition and creates the routes and methods based on the definition. At this stage, the developer can specify additional information about these APIs, such as the back ends that implement the APIs or the policies that need to be applied. [Figure 3-10](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch03.xhtml#ch03fig10) shows how an API deployment infers the routes and HTTP methods from the paths specified in an API spec. To get started, developers can use a stubbed-out back end; as real back-end services are built out, they can simply be switched in.



**Figure 3-10** The Service Can Parse the API Spec and Populate the Routes in an API Deployment

**Messaging Systems**

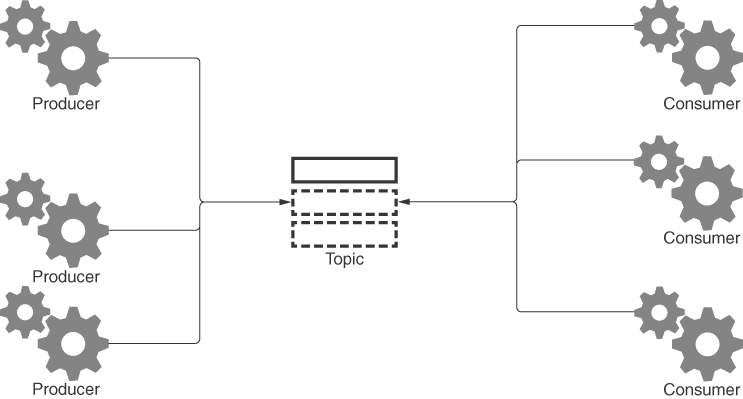
Modern cloud native architecture that emphasizes resiliency through scalable, loosely coupled components relies on asynchronous messaging between components. Messaging-based architectures allow microservices to scale and become location transparent because systems are not connecting to each other directly; instead, they are communicating through messages brokered by a messaging system. This enables load management, elasticity, and flow control by shaping and monitoring the message queues in the system. It also enables developers to manage failures as messages. The asynchronous and nonblocking communication in message-driven architectures consumes resources only while active, delivering optimized resource usage and cost optimizations. Today several leading CNCF projects exist in the messaging space, including NATS.io and Cloud Events, providing a wide array of features, programming models, and performance characteristics.

In message-driven architectures, a message usually represents an object and its state or a change in its state. For instance, in an ecommerce application, a message that is sent from a cart service to an order service can identify an item by its ID, the quantity to be purchased, payment information, and the user who is purchasing it. The orders system can process the messages as they arrive, checking inventory, processing the payment, and sending more messages to other systems, such as a fulfillment system for the orders placed. Here, the systems are not directly communicating, nor do they know the other systems that are receiving the messages. This allows for flexible architectures—if one service were to be replaced, the other services would not even be aware of this change. This allows systems to scale independently and maintain well-defined failure boundaries, avoiding cascading failures. If a service fails, the messages intended for that failed receiver simply wait for the system to return to an operational state and pick up from the next message that is to be processed. Developers thus can work with well-defined contracts expressed as message formats, and rollout changes to parts of the system can be done with greater velocity and agility, yet without the need for highly coordinated release workflows.

Several messaging approaches offer different semantics, including message queuing and publish subscribe. These models have three key components: message producers, the messaging system, and message consumers. Message producers are applications that generate messages. Message consumers are applications that consume and process messages. They are connected by the messaging system, which provides features such as message storage, message order, at-least-once delivery, and at-most-once delivery. Note that the messaging system decouples the producer and consumer. Each component knows only about the messaging system, not about each other. This isolation of each component allows them to be independently deployed, scaled, and patched.

In message queuing, messages sent by the message producer are stored in a buffer until they are dequeued or consumed by another component. The message is processed only by the message consumer that dequeues it from the queuing system.

In the publish subscribe (Pub/Sub) model, shown in [Figure 3-11](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch03.xhtml#ch03fig11), the producers (also known as publishers) publish messages onto a topic, and the message can be processed by all consumers (also called subscribers) who have subscribed to the topic. Note that in the Pub/Sub model, the same message is processed by many subscribers, unlike in a queue.



**Figure 3-11** The Pub/Sub Model Connects Distributed Systems Using Messages That Are Published and Consumed on a Topic

Among these three components, the messaging system is usually the most complex. It handles message persistence, ordering, delivery guarantees, and more. The producer and the consumer interact with the messaging system using an API that the messaging system provides. The APIs provided by the messaging system aim to make the producer and consumer code as simple as possible. For most platforms, the producer API simply lets an application publish a message, and the consumer API lets an application read these messages. In practice, when there is a multitude of producers and consumers, the messaging system provides the heavy lifting of ensuring the throughput, infrastructure resources, and delivery semantics. This makes the task of running and maintaining open-source messaging systems such as Kafka and NATS appreciably complex and nontrivial. As a solution to this problem, most cloud vendors provide a fully managed messaging system that either is built on top of the open-source tools or has API compatibility with open-source tools.

**Streaming**

The OCI Streaming service provides a scalable messaging system with durable storage. It is a fully managed service that can be used for ingesting continuous streams of data. Streaming service is suited for building web-scale applications and microservices that use a message-driven architecture. These applications are typically designed around data that is produced and processed continually and sequentially in a Pub/Sub messaging model. The OCI Streaming service is also suited for applications that ingest logs, metrics, and operational telemetry, as well as other fast data streams, such as website clickstreams. As a fully managed service, OCI Streaming manages all infrastructure needed to operate and scale the service, from provisioning, deployment, maintenance, and replication to configuration of the hardware and software that enables you to stream data. As a user of the service, you create a *stream* and configure the *partitions*. Streams and partitions are resources provided by the service; they are discussed in the next section. You can securely put and get your data from Streaming through SSL endpoints using the HTTPS protocol. The service ensures that user data is encrypted both at rest and in transit, and you can bring your own encryption keys that you manage in the OCI Vault service. Streams also support private endpoints, which limits the visibility of your streaming endpoint so that it is restricted within your virtual cloud network (VCN), preventing access through the Internet.

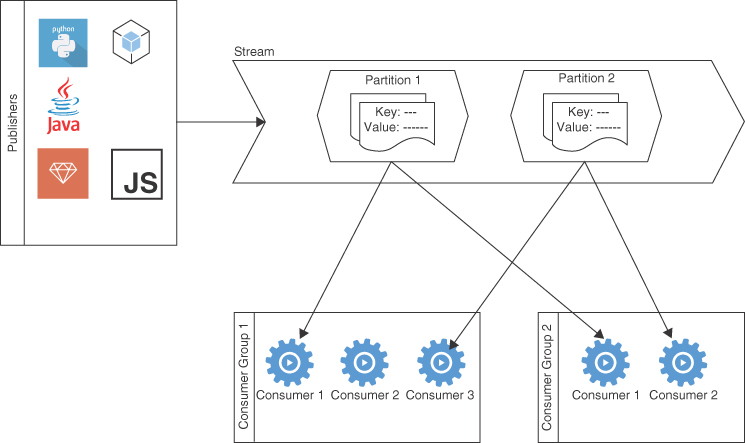
**Understanding the Streaming Service**

The OCI Streaming service is a fully managed service. As such, it exposes a resource called a *stream* that encapsulates the infrastructure required to operate a messaging system and manage its lifecycle. Developers first create a stream in the streaming service using the console, CLI, Terraform, or the APIs. A stream is the primary resource you interact with, and it can be thought of as an append-only log. Streams are organized into *stream pools* that provide a way to manage the settings for all the streams in a pool. If you do not explicitly associate a stream with a stream pool, the stream is created in the default stream pool.

After a stream has been created, applications can publish messages to it. In most cases, applications use the OCI SDK or the APIs directly to publish messages. You can also use the OCI console and the CLI to send messages to your stream for testing. Another popular way for applications to interact with the streaming service is to use the Kafka APIs, which the streaming service supports. Every message consists of a key and a value, both of which can be set by the developer. [Listing 3-3](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch03.xhtml#list3_3) shows a snippet of code using the Python SDK to publish a message. Multiple publisher applications can publish messages to the stream at the same time.

Subscribers or subscriber applications can consume messages from the stream either individually or as part of a *consumer group*. The streaming APIs and SDK offer many options for consumers to control how messages are delivered to them.

As applications publish messages to a stream, these messages are distributed to *partitions* that are managed by the streaming service. Each partition stores a subset of the messages that were published. Having multiple partitions allows message consumers to consume messages from multiple partitions at the same time. Because publishers and subscribers can use partitions in parallel, the number of partitions has an impact on the message throughput of the stream. There are limits to this as well. Each partition is limited to 1MBps of data write and 5 get requests per second from each consumer group. When a new stream is created, the number of partitions it should use needs to be specified. Once created, the number of partitions in the stream cannot be changed. Messages that are published onto a stream by producers are routed and stored on one of the partitions in the stream. [Figure 3-12](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch03.xhtml#ch03fig12) shows an overview of how applications can publish and receive messages using the streaming service, as well as the various components of the streaming service itself.



**Figure 3-12** An OCI Stream Showing Various Partitions and How Publishers and Subscribers Can Communicate Using Messages

**Working with the OCI Streaming Service**

A producer publishes a message onto the stream. The various SDKs for languages such as Java, Python, Go, JavaScript, and TypeScript provide wrapper methods to access the streaming APIs. A single call to publish messages can include multiple messages, but the total size of payload must be 1 mebibyte (MiB) or less. Each message that is published to the stream should contain a *key* and a *value*. If there is more than one partition, the steaming service determines the partition where the message is published using the message key. Based on the key, two messages with different keys could potentially be published on the same partition; however, messages with the same key always go to the same partition. If you do not specify a key, the service considers the message to have a *null* key and generates a random key for the message. Messages with null keys trigger the generation of random keys, so these messages do not pile up within the same partition. This avoids accidental hot spots, with messages with null keys all ending up on the same partition and impacting the throughput of the system. [Listing 3-3](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch03.xhtml#list3_3) shows a snippet of Python code that connects to a stream on the OCI Streaming service and publishes two messages with the single call. It shows the Python SDK reading the config to connect and authenticate, with OCI being loaded from a file and a streaming client being created.

**Listing 3-3** Example Code to Publish a Message Using the Python SDK

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch03_images.xhtml#f083-01a)

config = oci.config.from\_file()

streaming\_client = oci.streaming.StreamClient(

config, "https://service\_endpoint.url")

streaming\_client.put\_messages(

stream\_id="<stream\_OCID>",

put\_messages\_details=oci.streaming.models.PutMessagesDetails(

messages=[

oci.streaming.models.PutMessagesDetailsEntry(

value="FirstMessage",

key="key\_one"),

oci.streaming.models.PutMessagesDetailsEntry(

value="SecondMessage",

key="key\_two")])

The StreamClient provides the function put\_messages, which wraps the streaming service’s API for publishing messages. It requires the stream ID, which is the OCID for the stream, as well as a list of messages to publish. As mentioned previously, there is no limit to the number of messages that can be included in this function call, as long as the total size of the payload is 1MB or less. The message keys can be up to 256 bytes. The SDKS for other languages provide similar constructs.

**Note**

In general, applications should strive to design message keys that help spread the messages evenly. If a vast majority of the messages produced in a system have a common attribute, then using that attribute as a message key will lead to an overwhelming number of messages in a single partition, while other partitions remain relatively idle. Better throughput could be achieved by picking keys so that a large number of unique keys can be generated and as few messages as possible share the same key. This ensures better distribution of messages across the various partitions. Messages with the same key are guaranteed to be stored in the order in which they are published and are delivered to consumers in the same order that they were produced. Because messages with the same key go to the same partition, this guarantee applies only at the partition level.

Consumer applications consume messages from a stream using the API or the SDKs in a manner similar to how a producer publishes messages onto the stream. A consumer needs to start consuming messages from some point in the stream. Consumers use a *cursor*, which is a pointer to a specific location within a stream, to do this. Messages then are consumed starting with the one that the cursor points to. The streaming service guarantees that the messages from a partition are always delivered in the same order they were produced. After a cursor has been created, the consumer uses the GetMessages API to fetch messages. Similar to publishing messages, a single call to the GetMessages API returns multiple messages. By default, the number of messages that are batched inside a single response is based on the average message size, so as to not exceed the stream’s throughput. You can also specify the number of messages to be returned, as long as you do not exceed the throughput of the stream. As the number of messages returned from a call to the GetMessages API can vary based on the message size, the call also returns a cursor for use with the next GetMessages call. The cursor is returned as a response header value in the custom header opc-next-cursor. The next call to GetMessages can use the value returned in the header as the cursor parameter, to get the next batch of messages.

Individual consumers can start consuming messages from different relative points in the stream using different types of cursors. The types of cursors include ones that point to the following:

* A specific time (cursor type AT\_TIME)
* The earliest message available on the stream (cursor type TRIM\_HORIZON)
* A relative position within the messages on the partition, called an offset (cursor type AT\_OFFSET or AFTER\_OFFSET)
* Only messages published after the cursor has been created (cursor type LATEST)

This enables consumers to keep track of the various partitions, the position of the last message the consuming application has consumed from the partition, and from what position in the partition the consuming application needs to start in case it is interrupted or terminated and needs to restart consuming from where it left off.

Consumers can also be grouped into *ConsumerGroups* that coordinate the consumption of messages from a stream. In streams that have numerous partitions, keeping track of offsets and partitions while dynamically scaling the number of consumers can be cumbersome. ConsumerGroups can push to the streaming service most of the heavy lifting required to manage offsets and partitions when consumers are scaled up or down. This helps developers focus on what to do with messages instead of having to orchestrate message consumption. ConsumerGroups consist of multiple consumers, called *instances*. The ConsumerGroups automatically manage offset tracking, assign the various instances in the group to specific partitions, and balance the group as instances are created and removed in the ConsumerGroups. ConsumerGroups are more efficient and practical for most purposes than individual consumers simply because of the benefits they provide at no extra cost. ConsumerGroups use a cursor called a *GroupCursor*, which creates a *group name* and *instance name* association, in addition to performing the duties of a normal cursor. The first time a GroupCursor is created with a new group name, the ConsumerGroup by that name is created. When a group cursor is created with an existing group name and a new instance name, the consumer that requested the group cursor is added to the group as a new instance in the group. Each instance in a group is assigned a partition, and an instance may be assigned more than one partition. However, two instances will never be assigned to a single partition; if a ConsumerGroup has more instances than partitions, the extra instances remain idle. ConsumerGroups automatically remove instances that have not consumed messages for more than 30 seconds. In these cases, the idle instances in the ConsumerGroup are assigned to a partition whose assigned instance has been removed.

The 30-second window to request additional messages essentially means that consumers should ideally limit the number of messages requested to something that it can process within 30 seconds. If it takes longer than 30 seconds to process the message and call getMessages again, the service assumes that the consumer went offline and allocates the partition to an idle consumer. Data is not lost in these scenarios, though, because the default behavior of the GroupCursor is to commit messages on the next call to getMessages. So in a scenario in which a consumer has been terminated, fails, or cannot process all messages within 30 seconds, the messages are not considered committed (or processed). The partition is allocated to another consumer (when one comes online in the group, if there are no idle consumers), and these messages are delivered to the consumer for processing again. Some of these messages might have been processed by the failed consumer before it failed, so these messages appear as redundant to the second consumer. This also illustrates the “at least once” delivery model of the streaming service. How the consumer applications handle redundant messages is up to the consuming application, and they should be designed to account for multiple message deliveries in situations like the aforementioned one.

[Listing 3-4](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch03.xhtml#list3_4) shows a typical ConsumerGroup using a group cursor to consume messages.

**Listing 3-4** Consumer Group Using a Group Cursor

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch03_images.xhtml#f085-01a)

config = oci.config.from\_file()

streaming\_client = oci.streaming.StreamClient(

config, "https://service\_endpoint.url")

cursor\_details = oci.streaming.models

.CreateGroupCursorDetails(group\_

name="group01", instance\_name="instance01",

type=oci.streaming.models.CreateGroupCursorDetails.TYPE\_TRIM\_HORIZON,

commit\_on\_get=True)

response = sc.create\_group\_cursor(sid, cursor\_details)

cursor = response.data.value

while True:

get\_response = client.get\_messages(

stream\_id="ocid1.test.oc1..xxxxx.

streamId-Value",

cursor,

limit=10)

if not get\_response.data:

return

# Process the messages

print(" Read {} messages".format(len(get\_response.data)))

for message in get\_response.data:

print("{}: {}".format(b64decode(message.key.encode()).decode(),

b64decode(message.value.

encode()).decode()))

time.sleep(1)

# use the next-cursor for iteration

cursor = get\_response.headers["opc-next-cursor"]

[Listing 3-4](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch03.xhtml#list3_4) shows a stream client being created. A group cursor is also created that creates a ConsumerGroup called group01. This consumer within the group (instance) is named instance01. The initial group cursor is used to call the get\_messages API with a message limit set to 10. This is done to illustrate the fact that all instances in a ConsumerGroup should try to limit messages to what they can process within 30 seconds; a gap of more than 30 seconds between calls to the get\_messages API causes the service to consider the instance as offline, as previously discussed. After the messages are processed, the opc-next-cursor response header is extracted to get the cursor for the next call to get\_messages. Note that, in this example, with commit\_on\_get set to True when creating the GroupCursor, the first 10 messages that were returned are committed when the instance calls the get\_messages the second time. If this instance takes too long to process the first 10 messages or it went offline unexpectedly, then these messages are not committed and they are delivered to another instance if and when one becomes available.

**Service Connector Hub Integration**

The Streaming service is integrated with the OCI Service Connector Hub. The OCI Service Connector Hub is a messaging bus that enables you to orchestrate data movement between services in OCI. Using the Service Connector Hub, you can define the source for the data, a set of tasks that you can optionally apply to the data to process it (such as transforming the data), and a target service to deliver the processed data. Using the service bus connector, you can enable use cases in which you can use a stream as a data source, use Serverless Functions to transform the stream’s messages, and deliver the transformed messages to a target while maintaining Streaming’s order guarantees.

**Kafka Compatibility**

Streaming is compatible with most Kafka[**6**](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch03.xhtml#ref3_6a) APIs, enabling you to use applications written for Kafka to send messages to and receive messages from the Streaming service without having to rewrite your code. Streaming makes it possible to offload the setup, maintenance, and management of the infrastructure that hosting your own Apache Kafka cluster requires. Streaming also takes advantage of the Kafka Connect ecosystem to interface directly with first-party and third-party products by using out-of-the-box Kafka source and sink connectors. At the time of writing, the service offers compatibility with the Kafka APIs outlined in [Table 3-2](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch03.xhtml#ch03tab02).

**Table 3-2** OCI Streaming Compatibility with Various Kafka APIs

| **Compatible** | **Incompatible** |
| --- | --- |
| **Producer** | Compaction |
| **Consumer** | Transactions |
| **Kafka Connect** | Dynamic Partition Addition |
| **Group Management** | Idempotent Production |
| **Admin** | Kafka Streams |

If you use Kafka APIs to publish messages to Streaming, you can choose to do custom partitioning and explicitly map messages to partitions. Although this gives you more control and predictability over what messages are sent to which partitions, the Streaming service avoids this, to keep from accumulating too many messages in the same partitions and creating “hotspots.” When developers take control over partitions with custom partitioning, they also take on the responsibility to avoid hotspots from having too many messages within the same partitions.

The Kafka Connect support in OCI Streaming allows developers to leverage the Kafka Connect ecosystem of connectors to move data between systems. Several connectors make it easy to create integrations with Oracle platforms:

* Kafka Connect JDBC, for working with the Oracle database
* Oracle Integration Cloud
* Oracle Golden Gate
* Kafka Connect Amazon S3 connectors, which can use the Oracle Object Storage S3-compatible APIs

When using Kafka Connect, you need to create Kafka Connect Configurations called *harnesses* on the OCI Streaming service. A single harness can be used to configure multiple connectors and the harness needs to be created within the same compartment as the stream. Kafka Connect uses internal topics to track and manage connector and task configurations, offsets, and status. These internal topics are automatically created by the Streaming service and follow the convention <stream ocid>-{config|offset|status}. These topics can be configured in the distributed worker configuration file of the connector, typically connect-distributed.properties.

**OCI Events Service**

All OCI services emit *events*. Events can be thought of as status updates about lifecycle state or activities that these services are performing. The occurrence of a change, such as a compute instance being created (“Instance creation started”) or the completion of a block volume backup (“Block Volume backup complete”), can be represented as events. Events typically capture some context about the occurrence so that the event is actionable. An example is the OCID of the compute instance that was created. A downstream system can potentially take action based on this contextual information that is captured in the event. In large distributed systems like cloud platforms such as OCI, numerous services and components can emit events that signal their normal operations; in most cases, only a few events would be interesting and acted upon. With voluminous events being produced, the Events service provides a way to listen to or filter only interesting events and then use the context captured in the event to take an action. Although any system can emit events, in the context of the OCI Events service it is the OCI services that emit events.

In a general sense, events are a way for systems to communicate facts about their operations or statuses to other systems. Although events might sound similar to a message in a messaging system, an event described using the CloudEvents format represents a fact. As a signal about the occurrence of a change in a system and bundled with contextual information about the change, an event is not particularly intended for any one consumer. An event also notably lacks intent. This contrasts with messages, which usually convey some intent from one system to another. As cloud platforms gain popularity and application design evolves to become distributed, resilient, and more aware of its surrounding systems, events play a crucial role in enabling that transition. This has also led to a proliferation of event formats, which limits the interoperability of events across platforms. The CNCF project CloudEvents is an emerging standard that aims to unify how event publishers can standardize on the format used to describe an event. This enables events to be described in a standardized manner so that developers can build systems that can interoperate and handle events across cloud platforms. The OCI event service uses the CloudEvents format to describe events. [Listing 3-5](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch03.xhtml#list3_5) shows an OCI event that uses the CloudEvents specification.

**Listing 3-5** OCI Event Using the CloudEvents Specification

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch03_images.xhtml#f088-01a)

{

"eventType": "com.oraclecloud.computeapi.launchinstance.end",

"cloudEventsVersion": "0.1",

"eventTypeVersion": "2.0",

"source": "ComputeApi",

"eventTime": "2019-08-16T12:07:42.794Z",

"contentType": "application/json",

"eventID": "unique\_ID",

"extensions": {

"compartmentId": "ocid1.compartment.oc1..unique\_ID"

}

"data": {

"compartmentId": "ocid1.compartment.oc1..unique\_ID",

"compartmentName": "example\_compartment",

"resourceName": "my\_instance",

"resourceId": "ocid1.instance.oc1.phx.unique\_ID",

"availabilityDomain": "availability\_domain",

"additionalDetails": {

"imageId": "ocid1.image.oc1.phx.unique\_ID",

"shape": "VM.Standard2.1",

"type": "CustomerVmi"

}

},

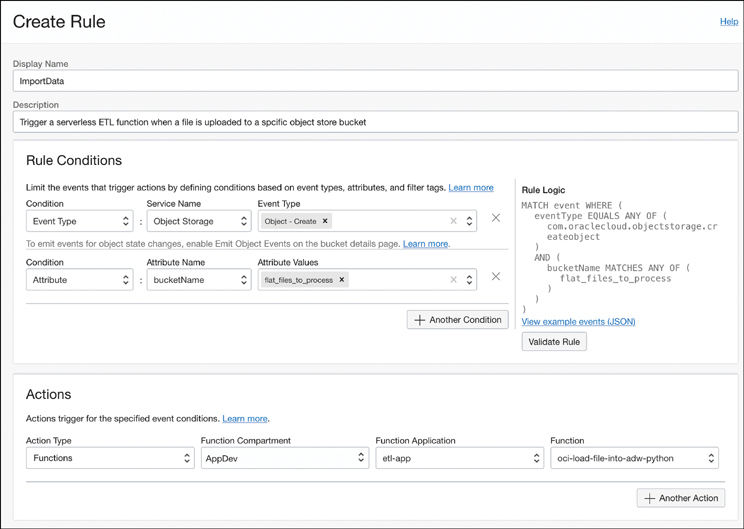
}

The structure of the event message can be broken down into two broad parts:

* **The event envelope:** The event envelope encompasses all the attributes at the top level, including the eventID, version, eventType, and other attributes. The event envelope is based on the cloud event specification, and the eventType usually provides the most basic mechanism to identify specific events to filter for processing. In the example presented in [Listing 3-5](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch03.xhtml#list3_5), the value for eventType is com.oraclecloud.computeapi.launchinstance.end, and predictably it represents the completion of a compute instance launch.
* **The payload:** The data or *payload* that is pertinent to the event itself is contained in the data attribute. The content and structure of the JSON payload differ for each event type, to be pertinent to the event. The schema for the payload can also be versioned to support schema evolution. This is when the content of the payload and its structure change over time to support newer attributes and features. The eventTypeVersion can be used to indicate what version of the payload structure or data schema the payload is using for the event type.

Services emit events continuously. You work with events by creating rules that match only certain event types, tags, or attributes contained in the event payload itself. The filtered events are then delivered to a target service that can act on the event. The OCI event service supports Notifications, Streaming, and Functions as target services where filtered events can be delivered.

Any attribute of the event can be used to filter events into a flow of events that a developer would be interested in. The filtering is done by creating conditions. Conditions match the event message structure and produce a flow of events that match the conditions, which can now be delivered to a target service. Conditions can use various operands such as any, all, or even wildcard-based matching. [Figure 3-13](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch03.xhtml#ch03fig13) shows the console with a matching rule.



**Figure 3-13** Working with the Events Service to Create a Matching Rule and an Action to Trigger When the Rule Condition Is Met

The console provides an intuitive interface to create and validate rules with sample payloads and tests for matching conditions. Here you see that a matching rule has been created for the event type com.oraclecloud.objectstorage.createobject. This event type represents an event for which a new object has been created in object storage—in other words, a new file has been uploaded to an object store bucket. The conditions further check whether the bucket where the new object has been created is named flat\_files\_to\_process.

After events have been filtered, the filtered events are directed to one of the supported target systems, such as streaming, functions, or notifications. The broad goal here is for the downstream systems to process the event data to perform an action. For instance, when an object storage bucket has been created, the event that notifies the completion of the bucket creation could be used to run a serverless function on a regular schedule to check for data inside this bucket.

In the example, there is a single action to invoke a specific function. The event will cause the function to be triggered, and the function can take any action based on the event data. In the example, you can presume that the function will use the event payload to access the file uploaded into the object storage bucket and perform some ETL job to import the data contained in it to a database. In this example, the event-driven model with a serverless function allows developers to build completely event-driven applications that process data when data becomes available so that they do not have to worry about infrastructure management.

The same event-driven principles can be applied to a variety of situations. The event service therefore enables application developers to build applications that can react to changes in the infrastructure and events or occurrences happening in the infrastructure layer directly, and then react to them from within their applications. This means that developers can build applications that are more resilient, autonomous, and elastic, thereby making the event service an important tool in building cloud native applications.

**Summary**

This chapter introduced several OCI services that are key to building cloud native applications. Although not all aspects of these services are explored in detail, these brief introductions should help you to see the big picture of the various OCI services and tools at your disposal and how they may interact. A few of the key services introduced in this chapter, such as Container Engine for Kubernetes, Container Instances, and OCI Functions, are examined in much greater detail in [Chapters 4](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch04.xhtml#ch04), [5](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#ch05), and [7](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch07.xhtml#ch07). It is not mandatory to use all or any of these services in all your cloud native applications; these are services that aim to help developers build loosely coupled, scalable applications at a high development velocity. Some of the managed services aim to remove as much operational overhead as possible so that developers can be less focused on handling the operational aspects and more focused on developing their applications. You should also keep in mind that OCI and the various open-source platforms and standards themselves are constantly evolving. New platforms and services may be added, and some services and standards might be eclipsed in the future by newer and more evolved versions of themselves.

**References**

[1](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch03.xhtml#ref3_1) Open Container Initiative: <https://opencontainers.org/>

[2](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch03.xhtml#ref3_2) Open Container Initiative image format specification: <https://github.com/opencontainers/image-spec/blob/main/spec.md>

[3](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch03.xhtml#ref3_3) Cloud Init: <https://cloud-init.io/>

[4](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch03.xhtml#ref3_4) Fn Project: <https://fnproject.io/>

[5](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch03.xhtml#ref3_5) OpenAPI Specification 3.0.0: <https://spec.openapis.org/oas/v3.0.0>

[6](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch03.xhtml#ref3_6) Apache Kafka: <https://kafka.apache.org/>