**7**

**Serverless Platforms and Applications**

Serverless platforms enable users to focus more on the applications they build than on infrastructure management. In this sense, the word *serverless* is a misnomer because serverless applications still require a server infrastructure to run. However, the servers on which applications run and the management of these servers generally are abstracted from developers by the cloud infrastructure provider. The rise of serverless platforms can be seen as a direct result of cloud adoption, in many cases. Applications are increasingly adopting cloud native design principles such as automation, immutability, and observability to take advantage of the elasticity, resilience, and cost advantages offered by public cloud providers. Developers thus can offload most of the infrastructure management to the cloud provider itself. This means that challenges that previously required custom solutions, such as scaling a system based on application metrics, can now be fully handed over to a cloud provider. For new applications, this often delivers a significant reduction in the time it takes to bring it to production: Teams no longer need an initial infrastructure setup that can potentially delay the start; the process of owning and operating the application also becomes significantly simpler, with no infrastructure management activity such as patching and updating servers.

The term *serverless* stretches across a wide range of services with varying degrees of capabilities. The term also can vary in meaning depending on how infrastructure is abstracted from developers. Some approaches require developers to use specific libraries and packaging methods; other approaches impose a lesser burden on the application development and packaging. The level of management and control that various platforms offer over the infrastructure can also differ. Some platforms autonomously manage scalability, and others let developers assign a guaranteed set of resources to their applications, with more control over scaling and cost management. Ultimately, these differences offer a choice to developers in building applications so that they can use the tools and processes that offer the right level of control and agility.

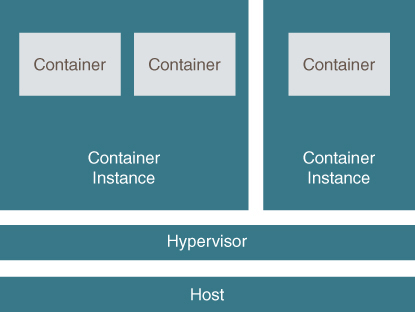
**Container Instances**

Container Instances in Oracle Cloud Infrastructure (OCI) offers a streamlined way of running application containers. At its core, Container Instances offer a simple and secure way to quickly launch containerized workloads on OCI, without the need to plan and manage servers or other infrastructure. The service manages the provisioning, lifecycle, patching, and upgrades for the infrastructure, allowing users to focus on application development. Compared to the alternative (in which a compute instance is spun up, a container runtime is installed, and then its lifecycle is managed through scaling and patching), Container Instances offer a faster path for deploying existing container workloads without modification and then operating them without infrastructure management overhead.

The experience for creating a container instance is simple: The user picks the number of CPU cores and memory, along with the container images that the user wants to run. Advanced controls throttle individual containers within a container instance, manage graceful container termination, provide environment variables, and override start-up parameters for the containers that users want to run. Container images can be pulled from any container image registry, including the OCI Container Image Registry (OCIR), and third-party registries such as ghcr.io, quay.io, and DockerHub, among others. When a container instance is launched, it can be allocated public or private IP addresses, and the application can be exposed to users. The Container Instances service also provides logging and monitoring capabilities out of the box. All infrastructure management, including patching and upgrades, is handled by the service transparently, without any burden on the user or the workload.

**Architecture**

A container instance is an OCI resource similar to a compute VM that is fine-tuned to run containerized applications. Container instances offer hypervisor-level isolation between each other and can support multiple containers within a given container instance. Within a container instance, the containers can interact with each other, similar to containers in a Kubernetes pod. Containers can communicate with each other over the loopback interface and 127.0.0.1, and they can share ephemeral storage. The strong isolation between container instances that this architecture provides can even protect workloads from being compromised by potential container escape vulnerabilities. A malicious user that escapes a container has visibility to the other containers within the container instance but is isolated from other container instances at the hypervisor level, thus limiting the attack vectors. [Figure 7-1](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch07.xhtml#ch07fig01) illustrates the architecture for container instances.



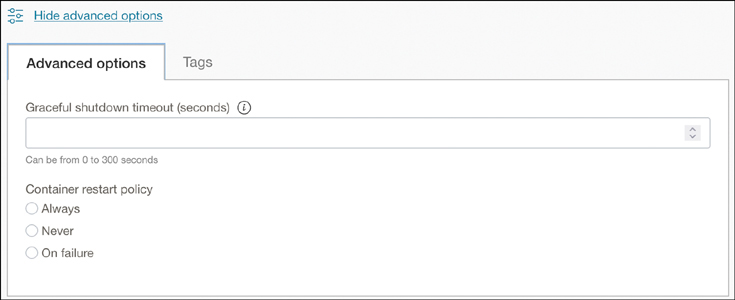
**Figure 7-1** Container Instances Offer Strong Isolation Without Infrastructure Overhead

**Using Container Instances**

When you launch a container instance, you can choose several parameters that control how the container instance behaves and reacts to events. In addition to choosing the name for the container instance, you can choose the number of OCPUs and memory to allocate to the container instance. This compute capacity is shared among all containers in the container instance, but the service also offers the capability to throttle resources allocated to each container so that a single container cannot hog the resource in the container instance. You can also select the networking properties for the container instance, including the subnets, NSGs, and more. If you allocate a public IP address to the container instance, the instance can be reachable from the Internet, as long as the NSG rules or security lists allow traffic and as long as an application within one of the containers in the container instance is listening on an open port.

Container instances support setting a restart policy. This is important because the containers in a container instance are not managed by a container orchestration tool such as Kubernetes. The restart policy for the container instance determines how container exits within the container instance are handled. A restart policy of *Always* restarts an exited container always, and this is a good choice for services that you always want running, such as a web server or a database. On the other hand, a container that just performs some one-time activity is generally expected to exit once the job is done, and these can choose a restart policy of *Never*. The choice of *OnFailure* is appropriate for containers that you expect to exit, but with an exit code of 0; this setting restarts the container if it exits with a nonzero exit code.

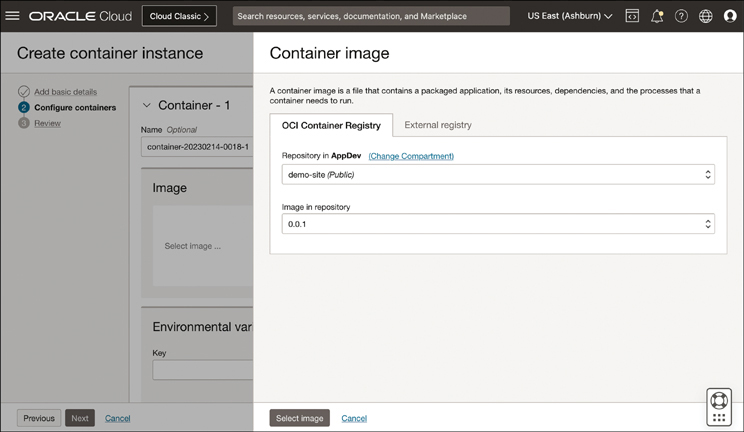
Container instances can also be configured with a graceful shutdown timeout. This value determines how long the service will wait for a container inside a container instance to exit before forcibly killing the container after the container instance has received a termination request. This is important for application containers that might be holding external resources, such as database connections, and need to clean up before exiting. When you terminate a container instance, the container instance terminates the containers, which gives containers a way to clean up before exiting. The container instance then waits for the duration set for the *graceful shutdown* before forcibly terminating it. [Figure 7-2](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch07.xhtml#ch07fig02) shows the restart policy and graceful shutdown time period.



**Figure 7-2** Container Instances Offer a Restart Policy That Manages How Containers Within the Container Instance Are Restarted

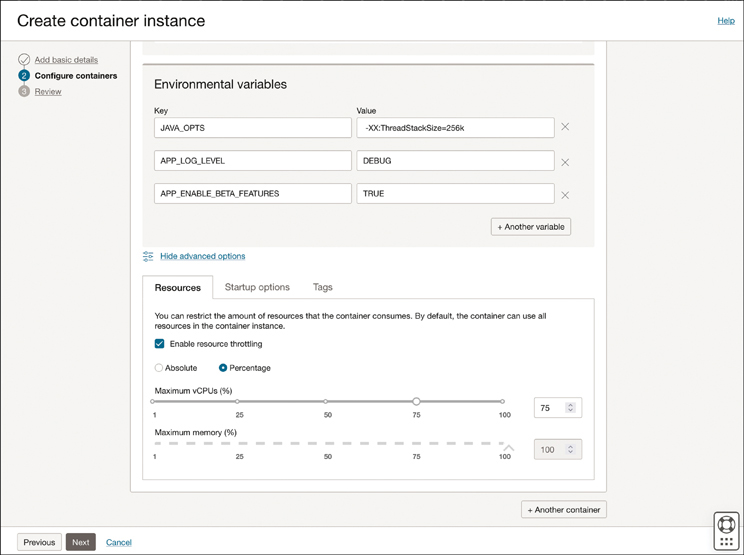
A container instance can run multiple containers within it, and each of these containers can also be configured independently. These options include the capability to throttle resources per container, overriding the container image’s startup options and setting the environment variables for the container. An example can help in examining these settings.

Imagine that you are deploying an application that is accompanied by a sidecar container that exports logs from the application to an external datastore. You assume that the application is following standard best practices for configuring the application using environment variables and externalized configuration. You start by creating the container instance and choosing the desired amount of OCPUs and memory for all containers that you plan to place within the container instance. After you choose the network’s configuration and, optionally, the restart policy, you can choose the containers that you want to place in the container instance (see [Figure 7-3](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch07.xhtml#ch07fig03)). The container image for each container can be from a different repository; OCIR is the default option; however, there is support for any external or third-party registry.



**Figure 7-3** Container Images Can Be Sourced from the Oracle Container Image Registry (OCIR) or Any External Registry

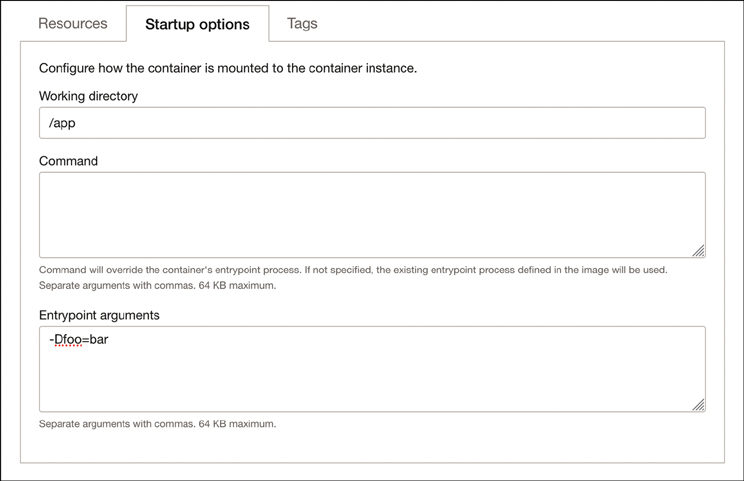
After you choose the image, you can configure it. The example in [Figure 7-4](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch07.xhtml#ch07fig04) uses a Java application, so here you can set some application properties and JVM arguments using the environment variables. Optionally, you can also enable resource throttling on the instance, which limits the amount of resources (such as CPU and memory) that a container can consume from the container instance. Resource limits can be set using absolute units or percentages. In the example in [Figure 7-4](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch07.xhtml#ch07fig04), the CPU usage for the application container is limited to 75% of what is available to the container instance.



**Figure 7-4** Container Instances Can Throttle Resources for Each Container Within It, Giving Developers Greater Control over How Resources Are Allocated Among Containers

Similarly, you can add a sidecar (say, a fluentd container from DockerHub) and throttle it to 25% of the CPU. This configuration for the resource throttling ensures that the app container does not consume more than 75% of the CPU and that the fluentd container does not consume more than 25%. Assuming that these are the only containers in the container instance, this effectively guarantees up to 25% of the CPU to fluentd and up to 75% of the CPU to the app container. By default, containers can consume as many resources as possible; therefore, you could have resource-throttled only the fluentd container to 25% and left the application container without any limits. Such a configuration would limit the fluentd container from consuming any more than 25% of the CPU, while allowing the app container to use as much CPU as it needs, without guaranteeing any fixed quota of CPU for the fluentd container. Based on application needs, these resource throttles can be used to optimally allocate resources to the containers in your container instance.

Configuration for each container also includes overriding some of the image defaults, such as the working directory, the command, and the entry point arguments, as illustrated in [Figure 7-5](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch07.xhtml#ch07fig05). These parameters correspond to the respective WorkingDir, Cmd, and Entrypoint configuration properties in the Open Container Initiative Specification.[**1**](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch07.xhtml#ref7_1a)



**Figure 7-5** Each Container in a Container Instance Can Be Configured Using Environment Variables and Startup Options

**Serverless Functions**

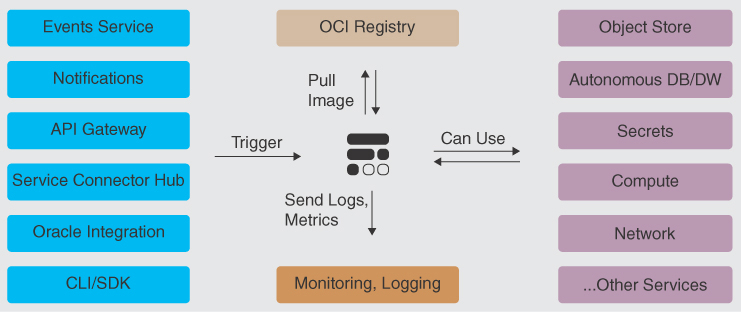
Functions-as-a-Service (FaaS) platforms are the ubiquitous serverless computing model in which you develop an application and then deploy and operate it without any infrastructure management. Functions typically use a programming model that leverages a software stack that includes components that are purposely built for integrating the function with a service that manages the infrastructure needs and scaling for the application. This enables you to quickly build highly scalable applications and operate them purely from the application’s business functionality and feature needs. This model lets you focus on application development without factoring in infrastructure management and thereby enables a drastically faster time to bring applications to production usage.

As the name indicates, a serverless function typically models a single *action*. Functions can be invoked directly or can be triggered by events. Most cloud platforms have an events service that provides a mechanism for your applications or other services to be notified about actions that occur within a cloud environment. An event service captures metadata about the occurrence of an action in the environment (say, a file being uploaded to an object storage bucket) and delivers the event to any service or application that is subscribing to it. Functions are commonly written to respond to events, as in the aforementioned example in which a file is being uploaded to an object storage bucket and now can be read by the function to perform some data processing. Functions and events can be chained together to create complex flows of data that are triggered by various events. This allows for the creation of an elastic and reactive system that springs into action when it needs to. Because the operational infrastructure for functions is abstracted away from the user and usually managed by a cloud provider, these platforms can optimize the resource usage by automatically scaling the resources for function invocations. Functions can be spun up when they are invoked, scaled up to meet demand peaks, and scaled down when they are no longer being invoked. Therefore, functions do not always need a running process because they are instantiated and scaled based on actual usage or invocations.

Functions also typically enforce the use of a prescriptive programming model, an SDK, or libraries that coordinate and manage the lifecycle of the function from the cloud service. These models can also place requirements and take an opinionated approach to how functions are developed, run, and managed. For instance, because most FaaS platforms abstract infrastructure management, functions are typically charged by the number of invocations instead of the usual infrastructure units of billing, such as CPUs. Functions can use as many CPUs as they need, and scale as they need, and the user pays for only how many times the function is invoked. With models like these, platforms can limit how long a function can run, what programming languages can be used to build functions, and more. These choices can also prevent portability of your functions across cloud providers, causing a gradual vendor lock-in for your applications. Care should be taken to evaluate the functions’ platform so that you do not have to sacrifice portability and productivity to gain the features of a serverless FaaS platform.

**OCI Functions**

Oracle Cloud Infrastructure Functions is a fully managed FaaS platform that is based on the Fn Project open-source engine. Fn Project is an open-source, container-native, serverless platform that can be run anywhere—in any cloud or on-premises. With OCI Functions, you develop your application in Java, Python, Node, Go, Ruby, and C# using the Function Development Kit (FDK) and then deploy it to the platform. Advanced users can also bring their own Dockerfile or use GraalVM. No infrastructure administration or software administration is necessary for you to perform. You do not provision or maintain compute instances, and operating system software patches and upgrades are applied automatically. OCI Functions simply ensures that your app is highly available, scalable, secure, and monitored. You can then deploy your code, call it directly, or trigger it in response to events, and get billed only for the resources consumed during the execution. [Figure 7-6](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch07.xhtml#ch07fig06) illustrates how the OCI Functions service integrates with other systems and services to provide a model that responds to events and other triggers.



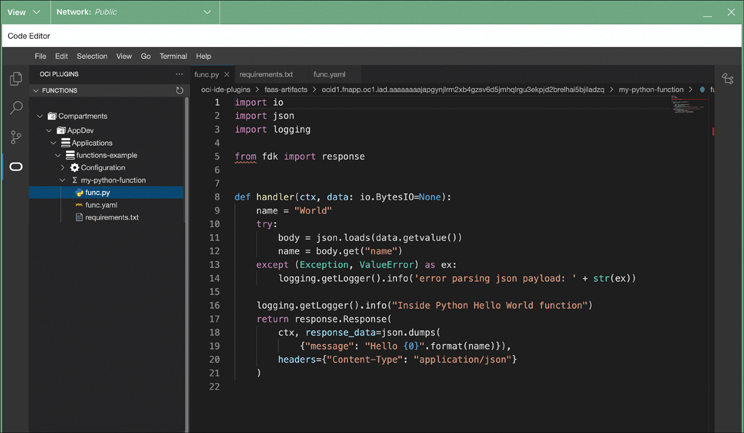
**Figure 7-6** The Functions Service Integrates with Other Systems and Can Respond to External Triggers

**Using OCI Functions**

An *application* is the most fundamental resource you create when you start working with OCI Functions. An application resource can be considered as a logical container for grouping functions. An application can have multiple functions inside it. Functions within an application can share configuration variables and resources, which are allocated to the application. For instance, networking resources, such as subnets to run the functions in, and the logging configurations for functions are configured at the application level when defining the application. When functions from different applications are invoked simultaneously, the application construct acts as the isolation boundary, ensuring that the functions are executed in isolation from each other. Ideally, similar and closely related functions should be grouped into a single application, for better efficiency and performance.

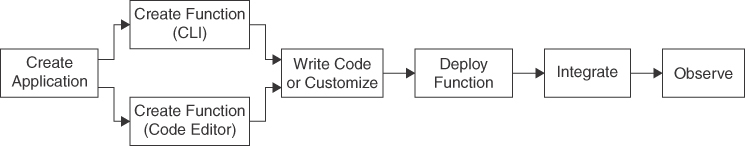
The function itself is a piece of code that you write using the Functions FDK. This code is built and packaged as a container image. When working with the OCI Functions service, developers can use the Fn CLI tool or the OCI Code Editor built into the OCI Console to generate the scaffolding for your functions code, build your function, deploy it to OCI, and manage the lifecycle of the function. The OCI Cloud Shell also comes with the Fn CLI preinstalled, if you need to experience it without installing and setting up a local development environment. Each function is part of an application and contains metadata that is stored on the OCI Functions service that tells the service how to create the execution environment for the function and execute it.

To create an OCI function, you start by creating a new application or choosing an existing application within which to create the new function. Then the function is created using the CLI or the OCI Code Editor. Creating the function includes generating the code scaffolding and the function metadata. [Figure 7-7](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch07.xhtml#ch07fig07) shows the scaffold generated within the OCI Code Editor.



**Figure 7-7** The Code Editor in the OCI Console Comes Preconfigured with Plug-ins for Functions That Can Generate Code Scaffolding and Deploy Functions

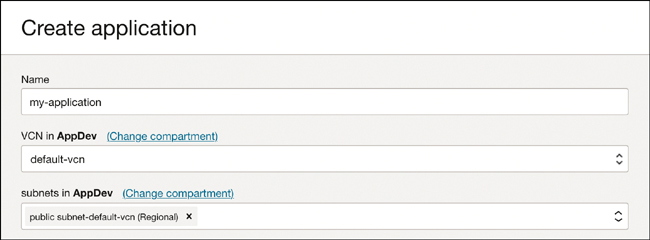
The OCI Functions tooling also enables you to customize the scaffolding it creates for you, such as using your own Dockerfile to build the final container image or using your own custom image to base the function on. When you have generated the scaffold and written the code to accomplish the task you want your function to perform, you can deploy the function. Deploying the function using the tooling provided by OCI Functions will build your code, package it as a container image, and push the container image to the image repository. This deployment process also pushes the function metadata to the OCI Functions service, which identifies the image to use for the function, and properties for the function, such as the version of the function and the runtime. After the function is deployed, it can be invoked directly or in response to events. In most circumstances, functions are invoked in response to a cloud event. The OCI Functions service manages the lifecycle of the infrastructure used for function invocations and scales it up or down in real time, based on invocations. After a function has been deployed, you can gather metrics and other observability data from it, as well as manage the lifecycle and security controls for it. [Figure 7-8](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch07.xhtml#ch07fig08) outlines the overall process.



**Figure 7-8** The High-Level Developer Workflow for Building and Deploying OCI Functions

**Building Your First Function**

To get started with OCI Functions, you first create an Application. This is easily done though the console, by choosing a name for your application, the VCN, and the subnet to use for the function (see [Figure 7-9](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch07.xhtml#ch07fig09)). With an Application to group your functions, you can now start with creating your first function.



**Figure 7-9** The Basic Configuration Elements for Creating an Application

The essential prerequisites are the OCI Functions CLI and a configuration profile for the OCI CLI. The Fn CLI uses the OCI CLI configuration to authenticate itself with the OCI Functions service. The OCI Cloud Shell and the OCI Code Editor both support building functions out of the box; this is the easiest way to get started with OCI Functions.

**Note**

You can also configure a local development environment to use with OCI Functions. The steps required for this setup are covered in the OCI Functions documentation.[**2**](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch07.xhtml#ref7_2a)

Start by opening the OCI Cloud Shell. When using the OCI Cloud Shell, the Fn CLI is already installed for you. The Fn CLI configuration is grouped into *contexts*. A context specifies the OCI Functions endpoints, the compartment to which deployed functions will belong, and the container image registry to use for the functions you build. The OCI Cloud Shell comes preconfigured with two contexts, a default context and a region-specific one. The regional context should be set as the default, to ensure that this uses the following commands:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch07_images.xhtml#f0309-01a)

fn list context # Shows the contexts that are available.

# This should show two contexts, with the

# regional context set as the current one.

fn use context <ctx\_name> # Explicitly set the context to the regional context.

**Note**

These commands can be found on the application’s Getting Started page, where the relevant OCIDs are prepopulated for you in a manner that is easy to copy and paste into the OCI Cloud Shell.

The preconfigured regional context has already set up the OCI Functions endpoints for the region. You now need to configure the compartment to which the functions will belong, as well as the container image registry. To configure these parameters, use the following commands, substituting the placeholders for the compartment\_ocid and the image registry prefix:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch07_images.xhtml#f0309-02a)

fn update context oracle.compartment-id <compartment\_ocid> # sets the compartment

under which functions will be created.

fn update context registry <region\_key>.ocir.io/<tenancy-namespace>/[repo-name-

prefix]

When you deploy a function, the Fn CLI builds a container image and pushes the container image to the registry that is configured. Therefore, you should also log into the registry. For OCIR, you need to generate an Auth token from your User Settings page (which is accessed using your logged-in profile icon on the top right of the screen), which you can use to log in to your registry.

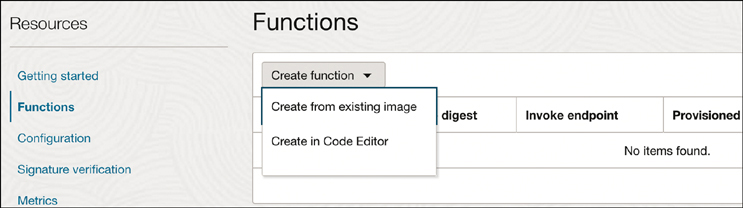
[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch07_images.xhtml#f0309-03a)

docker login -u '<tenancy-namespace>/<username>' <region\_key>.ocir.io

**Note**

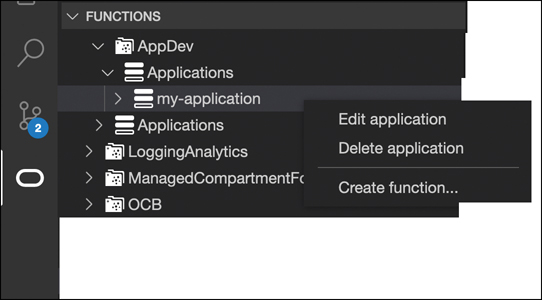
If you use the Oracle Identity Cloud service to federate your user account, the username is in the form <tenancy-namespace>/oracleidentitycloudservice/<username>.

With the CLI configured, you can now create a function using the OCI Cloud Shell. On the OCI Functions page, choose to create a new function in the OCI Code Editor, as demonstrated in [Figure 7-10](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch07.xhtml#ch07fig10).



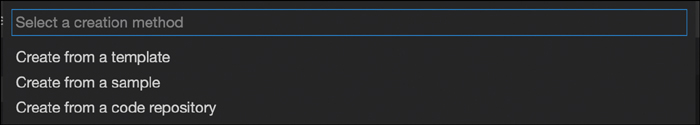
**Figure 7-10** Accessing the OCI Code Editor for Creating a Function

When the OCI Code Editor launches, it walks you through creating a scaffold for your application. Alternatively, you can open the OCI Code Editor, navigate to the application object within the OCI Function plug-in, and create a function directly from within the OCI Code Editor, as shown in [Figure 7-11](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch07.xhtml#ch07fig11).



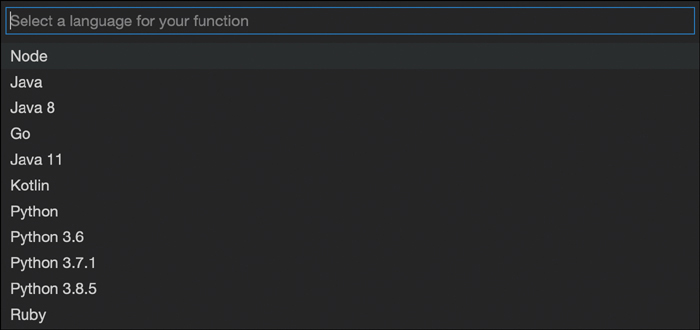
**Figure 7-11** The OCI Code Editor Has Built-in Support for Working with OCI Functions

The OCI Code Editor walks you through a workflow to start your function. It offers the capability to start from a template, an existing Git repository, or a sample (see [Figure 7-12](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch07.xhtml#ch07fig12)).



**Figure 7-12** The OCI Code Editor Offers Multiple Options to Bootstrap a New Function

For your first function, you can start from a template and choose the language you want to develop your function in. The example in [Figure 7-13](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch07.xhtml#ch07fig13) shows Python. You are prompted for the function name, and the OCI Code Editor then generates a scaffold for the function that you can expand upon.



**Figure 7-13** When Starting from an OCI Functions Template, You Can Build Your Function Using a Wide Range of Languages and SDKs

When the scaffold is created, it will have files named func.yaml, func.py, and requirements.txt. The func.py and the requirements.txt are part of the scaffold, with starting code and dependencies. The func.yaml is the function definition. It looks similar to [Listing 7-1](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch07.xhtml#list7_1).

**Listing 7-1** An Example of a Function Definition Generated by the OCI Code Editor

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch07_images.xhtml#f0311-01a)

schema\_version: 20180708

name: myfunction

version: 0.0.1

runtime: python

build\_image: fnproject/python:3.9-dev

run\_image: fnproject/python:3.9

entrypoint: /python/bin/fdk /function/func.py handler

memory: 256

This function definition identifies the function and several of its properties, including the runtime to use, the container images to use for building the function container, and the base image for the runtime. It also shows the entry point to use when the container image is generated and the amount of memory to request for the function. The OCI Functions service uses this definition to set up the runtime environment for the function at execution time.

With the scaffold generated, you can now build and test the function. The OCI Code Editor offers tools to commit the generated resources to a Git repository, including public repositories such as GitHub, and directly use the cloud editor to deploy and invoke the function. You can also use the terminal to quickly deploy and test your function using the Fn CLI. To do this, right-click func.py and choose Open in Terminal. The terminal window opens in the location where your function code is located. From here, you can build and deploy the function to the application in one step using the command in [Listing 7-2](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch07.xhtml#list7_2).

**Listing 7-2** An Example Showing a Function Deployment in Progress

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch07_images.xhtml#f0312-01a)

$ fn deploy --app my-application

Deploying myfunction to app: my-application

Bumped to version 0.0.11

Using Container engine docker to push

Pushing iad.ocir.io/xxxxxxxxx/functions/myfunction:0.0.11 to docker registry...

The push refers to repository [iad.ocir.io/xxxxxxxxx/functions/myfunction]

466fd30b96ba: Pushed

7464153686f5: Pushed

3ffc4f8259bf: Pushed

7656fdef3d98: Pushed

23ca6a735cc7: Layer already exists

b939f13c738d: Layer already exists

606f5e26329f: Layer already exists

0.0.11: digest: sha256:01e409b27ddb01c810fbe705b541dbfb6142485eabc9607f1a4bcdb0131

40cc2 size: 1781

Updating function myfunction using image iad.ocir.io/xxxxxxxxx/functions/

myfunction:0.0.11...

When the function is deployed, you can invoke it. On the OCI Functions page in the console, you can see the invoke endpoint (see [Figure 7-14](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch07.xhtml#ch07fig14)).



**Figure 7-14** The OCI Console Displays the Invoke Endpoint for a Function After It Has Been Deployed

To invoke a function directly using the invoke endpoint, you need to sign your requests. This authenticates the request with your identity. The most common way to do this is to use the raw-request feature of the OCI CLI. This enables you to use the OCI CLI to directly send requests to OCI resources such as your function, and the OCI CLI will handle the request signing. To use this to invoke the function directly, use the command in [Listing 7-3](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch07.xhtml#list7_3). The example here shows a POST request because the default scaffolding code generated by the OCI Code Editor can parse a JSON request body, if one is provided. [Listing 7-3](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch07.xhtml#list7_3) shows the typical response.

**Listing 7-3** An Example of How to Invoke a Function Manually Using the OCI CLI

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch07_images.xhtml#f0313-01a)

$ oci raw-request --http-method POST --target-uri <invoke\_endpoint> --request-

body '{"name":"user"}'

{

"data": {

"message": "Hello user"

},

"headers": {

"Content-Length": "25",

"Content-Type": "application/json",

"Date": "Tue, 14 Mar 2023 04:53:57 GMT",

"Fn-Call-Id": "01GVF7FGZB1BT0S38ZJ00E8QXM",

"Fn-Fdk-Runtime": "python/3.9.13 final",

"Fn-Fdk-Version": "fdk-python/0.1.51",

"Opc-Request-Id": "21D2172E020D43EFB212A3EB4B15A02E/xxxxxxxxxxxxxxxx/

xxxxxxxxxxxxxxxx"

},

"status": "200 OK"

}

When a function is invoked for the first time, the OCI Functions service pulls the function’s container image from the specified container registry, runs it as a container, and executes the function. If there are subsequent requests to the same function, OCI Functions directs those requests to the same container. If there are concurrent requests to the function, the OCI Functions service creates more containers, as needed, to scale up; after a period of being idle, the containers are scaled down as well.

To prevent the initial delay in serving a function call, you can also set up provisioned concurrency. Provisioned concurrency is a feature of OCI Functions by which the service always maintains the execution infrastructure for at least a certain minimum number of concurrent function invocations. Enabling provisioned concurrency ensures that your functions will have sub-second latencies from the very first invocation.

**Adding an API Gateway**

In most situations, the function endpoints are not directly exposed to external users. Instead, the function is wrapped by an API Gateway to provide some API shaping and control. The quickest way to create a gateway is through the OCI console, as shown here:

Step 1.Click **Developer Services -> API Gateway** from the sidebar on the left.

Step 2.Click the **Create Gateway** button.

Step 3.Enter the following values (you can use a different name if you’d like):

* **Name:** function-gateway
* **Type:** Public
* **Virtual Cloud Network:** Pick one from the dropdown
* **Subnet:** Pick the subnet from the dropdown

Step 4.Click **Create**.

Step 5.When the gateway is created, click the **Deployments** link from the sidebar on the left.

Step 6.Under Deployments, click the **Create Deployment** button. Make sure the **From Scratch** option is selected at the top, and enter the following values (you can leave the other values as they are—no need to enable CORS, Authentication, or Rate Limiting):

* **Name:** functions
* **Path prefix:** /functions
* **Compartment:**
* **API Logging Policies:** Information

Step 7.Click **Next** to define authentication, and choose **No Authentication**.

Step 8.Click **Next** to define the route. Enter the following values for Route 1:

* **Path:** /my-function
* **Methods:** POST
* **Type:** Oracle Functions
* **Application:** my-application (or other, if you used a different name)
* **Function name:** myfunction

Step 9.Click **Next** and review the deployment.

Step 10.Click **Create** to create the gateway deployment.

When deployment completes, navigate to it to get the URL for the gateway. Click the Show link next to the Endpoint label to reveal the full URL for the deployment. It should look like this:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch07_images.xhtml#f0314-01a)

https://j2fd2x25qkrtupcrtxvienbywy.apigateway.us-ashburn-1.oci.customer-

oci.com/functions

You can now use the following command to test your function through the gateway:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch07_images.xhtml#f0315-01a)

curl -X POST -d '{"name": "user"}' https://j2fd2x25qkrtupcrtxvienbywy.

apigateway.us-ashburn-1.oci.customer-oci.com/functions/myfunction

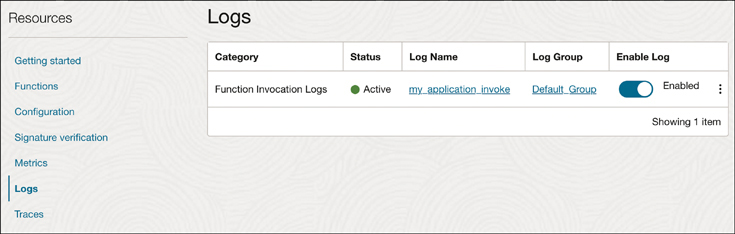
**Function Logs and Distributed Tracing**

After you have created a function, you want to observe the performance metrics of the function. Occasionally, when a function is not performing or behaving as expected, you also need to troubleshoot and debug it. As a serverless service, this can initially sound challenging because developers will not have access to the “servers” that are running these functions. However, the OCI Functions service provides multiple ways to enable developers to productively troubleshoot their code.

OCI Functions shows information about function invocations in metric charts. These are available on the console, by default, and include several metrics, such as the number of times a function is invoked, the duration for which the function runs, and invocations that resulted in a throttle (HTTP 429, “Too many requests”) or an error.

If you notice that your function metrics indicate an error or that the result of an invocation provided an unexpected response, you should enable logging for the function. Enabling logging automatically gathers the logs that a function emits from its code into the OCI Logging service. These log events capture the actual log line and metadata that can be searched through and analyzed further in the Logging service.

Logging can be enabled for an application and all functions within it. The logs are sent to a Log resource in a LogGroup; if these resources do not exist, they are created for you. [Figure 7-15](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch07.xhtml#ch07fig15) illustrates the logging controls at the application level.



**Figure 7-15** Logging Controls for an Application

Consider the code in [Listing 7-4](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch07.xhtml#list7_4) in a function handler.

**Listing 7-4** A Function Demonstrating Logging

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch07_images.xhtml#f0315-02a)

def handler(ctx, data: io.BytesIO=None):

name = "World"

logging.getLogger().info("Inside Python function")

try:

body = json.loads(data.getvalue())

name = body.get("name")

except (Exception, ValueError) as ex:

logging.getLogger().info('error parsing json payload: ' + str(ex))

return response.Response(

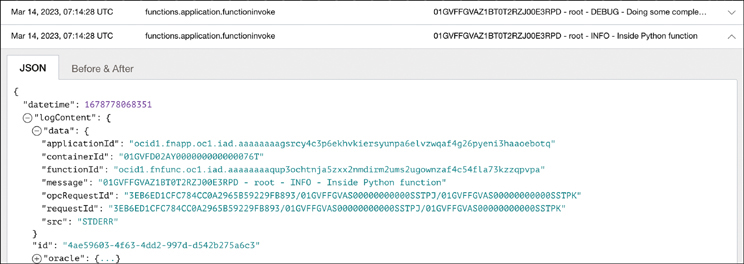
ctx, response\_data=json.dumps(

{"message": "Hello {0}".format(name)}),

headers={"Content-Type": "application/json"}

)

As the handler is invoked, the code logs the line Inside Python Function. When logging is enabled, these log lines are captured and collected within the Logging service, as demonstrated in [Figure 7-16](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch07.xhtml#ch07fig16). Developers can add more logs for specific conditions that help follow the code execution path and identify runtime conditions and data to troubleshoot a function.



**Figure 7-16** Logs from the Function Are Sent to the Logging Service, Where They Can Be Queried, Analyzed, or Exported to Other Systems

Beyond metrics and logging, the OCI Functions service also integrates with the Log Analytics service to provide deep performance analysis and tracing. As with logging, tracing can be enabled for an application; the traces are sent to an application performance monitoring (APM) domain in the Log Analytics service. If an APM domain does not exist, one is created for you.

With tracing enabled, you can add tracing spans directly in your code. Consider the example in [Listing 7-5](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch07.xhtml#list7_5).

The code adds the Zipkin tracing libraries to the application. Using these, every function in the code adds metadata such as service\_name and span\_name. The main handler function sets the span name Function Handler. From within this function, it invokes two other functions, do\_work and do\_more\_work. Both these functions add their spans and sleep for a short duration (150ms). Note that the second function, do\_more\_work, also throws an exception.

**Listing 7-5** An Example Function Illustrating the Use of Zipkin Libraries for Tracing

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch07_images.xhtml#f0317-01a)

import io

import json

import logging

import requests

import time

from collections import namedtuple

from py\_zipkin.zipkin import zipkin\_span

from py\_zipkin.encoding import Encoding

from fdk import response

def handler(ctx, data: io.BytesIO=None):

tracing\_context = ctx.TracingContext()

with zipkin\_span(

service\_name=tracing\_context.service\_name(),

span\_name="Function Handler",

transport\_handler=(

lambda encoded\_span: transport\_handler(

encoded\_span, tracing\_context

)

),

zipkin\_attrs=tracing\_context.zipkin\_attrs(),

encoding=Encoding.V2\_JSON,

binary\_annotations=tracing\_context.annotations()

):

name = "World"

logging.getLogger().info("Inside Python function")

try:

body = json.loads(data.getvalue())

name = body.get("name")

except (Exception, ValueError) as ex:

logging.getLogger().info('error parsing json payload: ' + str(ex))

do\_work(ctx)

do\_more\_work(ctx)

return response.Response(

ctx, response\_data=json.dumps(

{"message": "Hello {0}".format(name)}),

headers={"Content-Type": "application/json"}

)

# transport handler, needed by py\_zipkin

def transport\_handler(encoded\_span, tracing\_context):

return requests.post(

tracing\_context.trace\_collector\_url(),

data=encoded\_span,

headers={"Content-Type": "application/json"},

)

def do\_work(ctx):

with zipkin\_span(

service\_name=ctx.TracingContext().service\_name(),

span\_name="Doing Error Prone work",

binary\_annotations=ctx.TracingContext().annotations()

) as example\_span\_context:

try:

logging.getLogger().debug("Doing some complex task")

time.sleep(0.15)

except (Exception, ValueError) as error:

example\_span\_context.update\_binary\_annotations(

{"Error": True, "errorMessage": str(error)}

)

else:

FakeResponse = namedtuple("FakeResponse", "status, message")

fakeResponse = FakeResponse(200, "OK")

# how to update the span dimensions/annotations

example\_span\_context.update\_binary\_annotations(

{

"responseCode": fakeResponse.status,

"responseMessage": fakeResponse.message

}

)

def do\_more\_work(ctx):

with zipkin\_span(

service\_name=ctx.TracingContext().service\_name(),

span\_name="Do more work",

binary\_annotations=ctx.TracingContext().annotations()

) as example\_span\_context:

try:

logging.getLogger().debug("Doing some complex task")

time.sleep(0.15)

# throwing an exception to show how to add error messages to spans

raise Exception('do\_more\_work - failed. Handling error.')

except (Exception, ValueError) as error:

example\_span\_context.update\_binary\_annotations(

{"Error": True, "errorMessage": str(error)}

)

else:

FakeResponse = namedtuple("FakeResponse", "status, message")

fakeResponse = FakeResponse(200, "OK")

# how to update the span dimensions/annotations

example\_span\_context.update\_binary\_annotations(

{

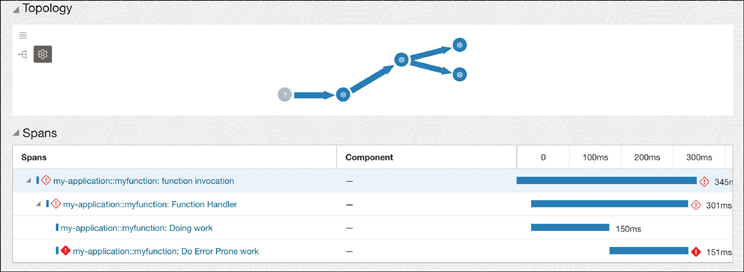
"responseCode": fakeResponse.status,

"responseMessage": fakeResponse.message

}

)

The trace for a function invocation shows the topology of the calls, as well as the spans, their duration, and their status (see [Figure 7-17](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch07.xhtml#ch07fig17)).



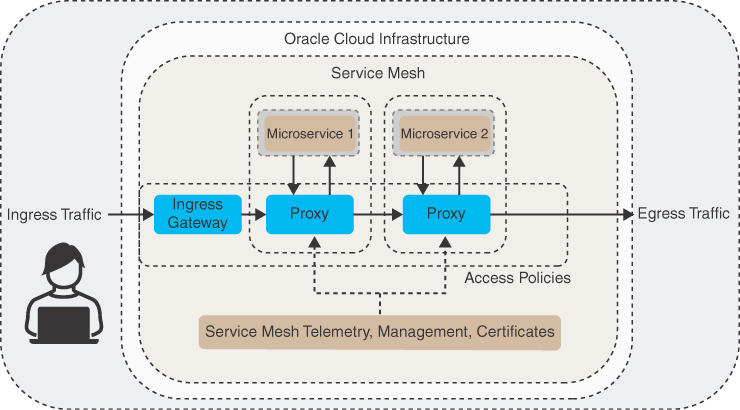
**Figure 7-17** Traces Can Be Visualized to Quickly Identify Performance Bottlenecks

Tracing enables developers to visually see the code execution flow and quickly identify where code execution time is spent. This allows developers to identify bottlenecks and address them quickly.

**Service Mesh**

As application services evolve into smaller and more focused services that scale independently, they get more distributed. This creates a challenge to manage the constantly evolving set of services and the communication between them. Common challenges include securing the communication between the services, managing traffic between services, and implementing observability across a range of services. A service mesh is an infrastructure layer for facilitating these service-to-service communications between services or microservices. A proxy-based model typically is used to accomplish this.

The OCI Service Mesh uses such a proxy-based model based on the Envoy proxy. The proxy runs alongside each microservice, which receives configuration information from a managed control plane. These proxies are separate from the services themselves, so the services and the application code do not need to change or even know about the proxy to use a service mesh. In a Kubernetes environment, these proxies run within the same pod as the application, but as separate containers and separate processes; thus, they are often called sidecars. The service mesh uses the proxies to implement security, observability, and traffic management on behalf of your application because the proxy acts as a facade to the actual application. [Figure 7-18](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch07.xhtml#ch07fig18) shows how the OCI Service Mesh introduces new resources into an existing application and transparently provides new features and functionality.

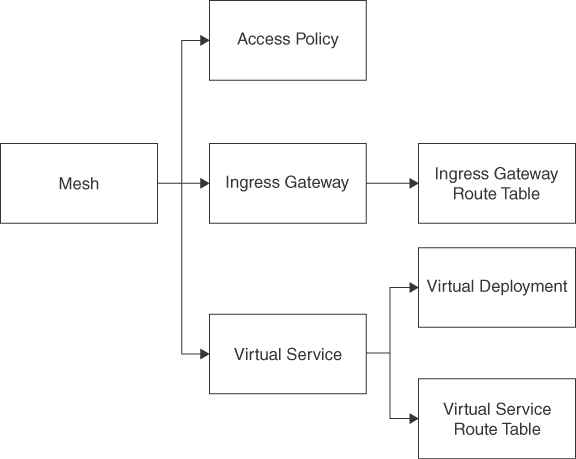


**Figure 7-18** The OCI Service Mesh Can Transparently Add New Capabilities and Rules to an Existing Application Without Code Changes, Using Its Proxy-Based Model

**Using the Service Mesh**

When the service mesh has been installed in your cluster, you can access the OCI Service Mesh using the OCI console, OCI CLI, REST APIs, and the Kubernetes CLI tool kubectl. However, you can manage your mesh resources either through the OCI APIs (Console, CLI, or Terraform) or through Kubernetes. When creating the mesh resources, you need to pick an approach. If you create the mesh resources with the OCI APIs, these resources will exist only on the control plane, with no corresponding resource definitions (CRDs) created in the Kubernetes cluster. However, if you choose to create these resources as CRDs in your Kubernetes cluster, the Kubernetes operator creates their definitions on the mesh control plane. The Kubernetes resources that you create using the CRDs become the source of truth, in this case, and you should continue to use this approach for modifying and managing the resources. Mesh resources managed by the Kubernetes operator cannot be modified using the OCI Console, CLI, or Terraform. In most cases, it might be more natural to manage these resources as Kubernetes resources. Mesh resources are closely associated with your application resources, and this is the most common way users interact with mesh resources.

As [Figure 7-18](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch07.xhtml#ch07fig18) illustrated, the OCI service mesh consists of resources that are mapped to application components such as *VirtualDeployments*, *VirtualServices*, *IngressGateways*, and more. These resources are instrumental in providing the Service Mesh features. [Figure 7-19](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch07.xhtml#ch07fig19) shows the various resources that make up the service mesh and how they are related.



**Figure 7-19** The Various Resources That Make Up a Service Mesh

The relationships and interactions between these resources can seem quite daunting at first, but these become easier to understand in the context of an example application.

**Adding a Service Mesh to an Application**

Consider this simple deployment of nginx in [Listing 7-6](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch07.xhtml#list7_6). This consists of a *Deployment* resource and a *Service* resource, to keep the example simple. The Deployment is backed by a replicas set with two replicas, as indicated by the replicas property in the deployment spec. The replicas set creates and manages the two pods of nginx. The pods have the label app: nginx, and the service targets all pods with the same label. The service is of type LoadBalancer, which indicates that the Kubernetes cluster will create an appropriate load balancer on the cloud provider and wire it to the pods targeted by the service.

**Listing 7-6** Kubernetes Resources That Describe an Application Without a Service Mesh

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch07_images.xhtml#f0321-01a)

apiVersion: v1

kind: Service

metadata:

name: my-nginx-svc

labels:

app: nginx

spec:

type: LoadBalancer

ports:

- port: 80

selector:

app: nginx

---

apiVersion: apps/v1

kind: Deployment

metadata:

name: my-nginx

labels:

app: nginx

spec:

replicas: 2

selector:

matchLabels:

app: nginx

template:

metadata:

labels:

app: nginx

spec:

containers:

- name: nginx

image: nginx:1.14.2

ports:

- containerPort: 80

To add a service mesh to this application, we can introduce the service mesh resources into the mix that wrap and encapsulate these Kubernetes resources.

The *Mesh* is the top-level resource that includes all mesh resources; it represents the boundary of services and traffic that the service mesh manages. The mesh also identifies a certificate authority that will be used to generate certificates to secure communications among the workloads that are covered by the service mesh. [Listing 7-7](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch07.xhtml#list7_7) shows the definition of a mesh resource.

**Listing 7-7** Mesh Resource Definition

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch07_images.xhtml#f0322-01a)

kind: Mesh

apiVersion: servicemesh.oci.oracle.com/v1beta1

metadata:

name: nginx

spec:

compartmentId: ocid1.compartment.oc1..

aaaaaaaahgeqvnooufd5efvafgiobngb2xfcih62h3u7o2sq2wfhei5ddgoa

certificateAuthorities:

- id: ocid1.certificateauthority.oc1.iad.

amaaaaaab5uyggqan7pi3iozlv2g7godjk5jclf3tlbrtlfar4c7sk3f72uq

displayName: nginx-mesh

mtls:

minimum: PERMISSIVE

*IngressGateways* are the entry point of traffic into all resources managed by the mesh. All incoming traffic passes through the ingress gateway, so it implements customized security policies on how to manage this ingress and transparently provide observability and traffic shaping for the services within the mesh. For example, the ingress gateway can be configured to enable encryption on all incoming traffic using TLS or to keep track of an access log. An ingress gateway can be configured with a set of DNS hostnames and listener ports that clients use to make their requests to. Clients communicate with the application through the ingress gateway using one of these hostnames and ports where the ingress gateway is listening. The ingress gateway is an optional resource and is not mandatory for all situations. The ingress gateway handles traffic as it enters the application, which provides the capability to track the traffic from the start. Without an ingress gateway, you would not have service mesh control over this segment in the traffic flow. [Listing 7-8](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch07.xhtml#list7_8) shows a typical ingress gateway. Notice that it is associated with a mesh resource.

**Listing 7-8** An Example IngressGateway Illustrating Hostnames and Listeners

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch07_images.xhtml#f0323-01a)

kind: IngressGateway

apiVersion: servicemesh.oci.oracle.com/v1beta1

metadata:

name: nginx-ingress-gateway

spec:

compartmentId: ocid1.compartment.oc1..

aaaaaaaahgeqvnooufd5efvafgiobngb2xfcih62h3u7o2sq2wfhei5ddgoa

mesh:

ref:

name: nginx

hosts:

- name: nginxHost

hostnames:

- nginx.example.com

listeners:

- port: 8080

protocol: HTTP

tls:

mode: DISABLED

accessLogging:

isEnabled: true

The ingress gateway resource is supported by resources such as an *IngressGatewayDeployment* and an *IngressGatewayRouteTable*. The ingress gateway resource defines the properties and configuration of the ingress gateway, which is an OCI resource. In a Kubernetes cluster, the ingress gateway is manifested as a set of Kubernetes resources, such as pods, that implement the configuration defined in the ingress gateway. This is accomplished by an *IngressGatewayDeployment* resource. This resource exists only on the Kubernetes cluster; it uses the configuration from the ingress gateway it is attached to and configures the Kubernetes resources in the cluster. This can be seen as a manifestation of the ingress gateway resource from the mesh service within a Kubernetes context and is analogous to the Kubernetes *Ingress* resource. When using the service mesh, the *IngressGatewayDeployment* resource replaces the *Ingress* resource. [Listing 7-9](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch07.xhtml#list7_9) shows an example ingress gateway deployment. Notice that the ingress gateway deployment has options to autoscale the number of pods handling the ingress traffic. The ingress gateway deployment creates a Kubernetes service, and the service.type can be used to control what type of service is created.

**Listing 7-9** An Example IngressGatewayDeployment Resource That Wraps an IngressGateway

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch07_images.xhtml#f0324-01a)

apiVersion: servicemesh.oci.oracle.com/v1beta1

kind: IngressGatewayDeployment

metadata:

name: nginx-ingress-gateway-deployment

labels:

mesh-ingress: nginx

spec:

ingressGateway:

ref:

name: nginx-ingress-gateway

deployment:

autoscaling:

minPods: 2

maxPods: 4

ports:

- protocol: TCP

port: 8080

serviceport: 80

service:

type: LoadBalancer

An *IngressGatewayRouteTable* defines traffic routing rules that are applied to the traffic that is coming through an ingress gateway. These route rules are applied to the specific ingress gateway hosts and ports. For HTTP requests, the ingress gateway route table has route rules that match an incoming request’s path and directs the requests to one of the virtual services. [Listing 7-10](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch07.xhtml#list7_10) shows an ingress gateway route table. It is associated with an ingress gateway, and the routeRules determine how traffic arriving on this ingress gateway gets routed to the various downstream virtual services.

**Listing 7-10** An IngressGatewayRouteTable Illustrating Various Route Rules to Be Applied

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch07_images.xhtml#f0325-01a)

apiVersion: servicemesh.oci.oracle.com/v1beta1

kind: IngressGatewayRouteTable

metadata:

name: nginx-ingress-gateway-route-table

spec:

compartmentId: ocid1.compartment.oc1..

aaaaaaaahgeqvnooufd5efvafgiobngb2xfcih62h3u7o2sq2wfhei5ddgoa

ingressGateway:

ref:

name: nginx-ingress-gateway

routeRules:

- httpRoute:

destinations:

- virtualService:

ref:

name: mesh-nginx

ingressGatewayHost:

name: nginxHost

A *VirtualService* is a logical service definition in a service mesh. In a typical Kubernetes application with pods and services, a virtual service represents the service mesh wrapper around multiple versions of a Kubernetes service. It is typically used to implement traffic flow control and management when deploying two versions of the service alongside each other. This makes it easy to implement canary deployments and have a solid rollback strategy in case the new version is deemed unstable. [Listing 7-11](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch07.xhtml#list7_11) shows a virtual service.

**Listing 7-11** An Example VirtualService

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch07_images.xhtml#f0326-01a)

kind: VirtualService

apiVersion: servicemesh.oci.oracle.com/v1beta1

metadata:

name: mesh-nginx

spec:

mesh:

ref:

name: nginx

defaultRoutingPolicy:

type: UNIFORM

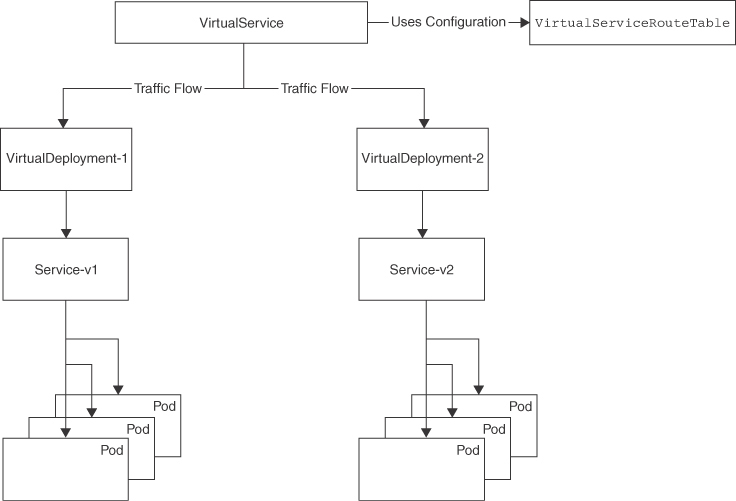
compartmentId: ocid1.compartment.oc1..

aaaaaaaahgeqvnooufd5efvafgiobngb2xfcih62h3u7o2sq2wfhei5ddgoa

hosts:

- mesh-nginx

Virtual services are backed by one or more virtual deployments that represent a single version of the Kubernetes service. In this regard, virtual deployments are associated with individual Kubernetes services. Virtual services and virtual deployments use DNS for service discovery; this allows applications to use consistent DNS names provided by the virtual services, while the service mesh can work with multiple versions of a Kubernetes service and perform traffic management transparently. The virtual service directs traffic to its associated virtual deployment based on a virtual service route table. [Listing 7-12](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch07.xhtml#list7_12) shows a virtual deployment as well as a virtual service route table. Note that the virtual service does not refer to the virtual deployments backing it, but the individual virtual deployments carry a reference to the virtual service. The virtual service route table determines how traffic is distributed to each virtual deployment that is part of the virtual service. To better understand these ServiceMesh objects and their relationships, consider the diagram in [Figure 7-20](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch07.xhtml#ch07fig20).



**Figure 7-20** Traffic Flow Through VirtualServices and VirtualDeployments, Based on the Configuration Contained in VirtualServiceRouteTables

**Listing 7-12** An Example VirtualDeployment

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch07_images.xhtml#f0327-01a)

kind: VirtualDeployment

apiVersion: servicemesh.oci.oracle.com/v1beta1

metadata:

name: nginx-v1

spec:

virtualService:

ref:

name: mesh-nginx

compartmentId: ocid1.compartment.oc1..

aaaaaaaahgeqvnooufd5efvafgiobngb2xfcih62h3u7o2sq2wfhei5ddgoa

listener:

- port: 80

protocol: HTTP

accessLogging:

isEnabled: true

serviceDiscovery:

type: DNS

hostname: nginx-v1

---

apiVersion: servicemesh.oci.oracle.com/v1beta1

kind: VirtualServiceRouteTable

metadata:

name: nginx-route-table

spec:

compartmentId: ocid1.compartment.oc1..

aaaaaaaahgeqvnooufd5efvafgiobngb2xfcih62h3u7o2sq2wfhei5ddgoa

virtualService:

ref:

name: mesh-nginx

routeRules:

- httpRoute:

destinations:

- virtualDeployment:

ref:

name: nginx-v1

weight: 100

isGrpc: false

path: /

pathType: PREFIX

A virtual deployment is a logical resource that is managed by the service mesh control plane. It needs an implementation or a manifestation of that definition on the platform it runs. Within a Kubernetes cluster, this is a *VirtualDeploymentBinding*, which connects the control plane definition of virtual deployment to the pods running on the Kubernetes cluster.

Here you have a typical application made up of several pods that is exposed as a service. Now consider that you deployed a newer version of this application. The diagram in [Figure 7-20](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch07.xhtml#ch07fig20) shows two versions of this application, v1 and v2, each exposed as its own service that is mapped to its own set of resources. You can use the service mesh features to perform some traffic management so that you can slowly transition users from the older version of the application to the newer version. The new version of the application is represented by the new virtual deployment, VirtualDeployment-2; the old version is represented by the virtual deployment VirtualDeployment-1. The virtual service can now direct traffic to either virtual deployments. This traffic management policy is defined by the virtual service route table associated with the virtual service.

Similar to these resources, the mesh supports access policies that define access rules for communication between virtual services and to external services. Access policies exist for the entire mesh and control whether communication is allowed between any given source and a destination. These sources and destinations can be internal or external. One example of internal communication is the common use case of one virtual service communicating with one or more other virtual services within the mesh boundary. In a microservice architecture, this type of control is helpful if you want to ensure that only a select set of services can communicate with a sensitive service. Examples of external communication include use cases in which clients outside the mesh communicate to virtual services through an ingress gateway where you can allow the ingress gateway to communicate with only a subset of virtual services. Another common example is an access policy that allows a virtual service within the mesh to communicate to a limited set of external destinations. [Listing 7-13](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch07.xhtml#list7_13) shows an access policy resource that is allowing communication from the ingress gateway to a virtual service.

**Listing 7-13** An Example Access Policy Controlling Communication Between an IngressGateway and a VirtualService

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch07_images.xhtml#f0328-01a)

kind: AccessPolicy

apiVersion: servicemesh.oci.oracle.com/v1beta1

metadata:

name: nginx-policy

spec:

mesh:

ref:

name: nginx

compartmentId: ocid1.compartment.oc1..

aaaaaaaahgeqvnooufd5efvafgiobngb2xfcih62h3u7o2sq2wfhei5ddgoa

rules:

- action: ALLOW

source:

ingressGateway:

ref:

name: nginx-ingress-gateway

destination:

virtualService:

ref:

name: mesh-nginx

By default, every mesh is secure and denies all communication, which includes ingress, egress, and communication within the mesh. When the service mesh resources are created, you must create an access policy to enable communication. These policies can be fine-grained, identifying exact sources and destinations, or they can be broader and target multiple virtual services or use wildcards for external hostnames.

With the application and the mesh resources deployed, the CRDs for the mesh jump into action to set up the proxies and implement the mesh. You can check the progress and status for the various resources using the command in [Listing 7-14](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch07.xhtml#list7_14).

**Listing 7-14** Interacting with Service MeshResources Using Kubernetes Tooling

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch07_images.xhtml#f0329-01a)

$ kubectl get meshes,virtualservices,virtualdeployments,virtualserviceroutetabl

es,ingressgateways,ingressgatewayroutetables,accesspolicies,virtualdeployment

bindings,ingressgatewaydeployments

NAME ACTIVE AGE

mesh.servicemesh.oci.oracle.com/nginx True 5d1h

NAME ACTIVE AGE

virtualservice.servicemesh.oci.oracle.com/mesh-nginx True 5d1h

NAME ACTIVE AGE

virtualdeployment.servicemesh.oci.oracle.com/nginx-v1 True 5d1h

NAME ACTIVE AGE

virtualserviceroutetable.servicemesh.oci.oracle.com/nginx-route-table True

5d1h

NAME ACTIVE AGE

ingressgateway.servicemesh.oci.oracle.com/nginx-ingress-gateway True 5d1h

NAME ACTIVE AGE

ingressgatewayroutetable.servicemesh.oci.oracle.com/nginx-ingress-gateway-route-

table True 5d1h

NAME ACTIVE AGE

accesspolicy.servicemesh.oci.oracle.com/nginx-policy True 5d1h

NAME ACTIVE AGE

virtualdeploymentbinding.servicemesh.oci.oracle.com/nginx-binding True 5d1h

NAME ACTIVE AGE

ingressgatewaydeployment.servicemesh.oci.oracle.com/nginx-ingress-gateway-

deployment True 5d1h

The first time you deploy the resources, you might notice a time delay for them to move to the ACTIVE state. This is expected because the resources are created in the control plane and reconciled. To look for potential issues or to check on progress, you can check the logs on the controller manager for the OCI service operator:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch07_images.xhtml#f0330-01a)

kubectl logs -n oci-service-operator-system deploy/oci-service-operator-

controller-manager -f

With the service mesh deployed and your application managed by the service mesh, you can now use the mesh resources to implement traffic shaping using virtual service route tables and control both north/south and east/west traffic access using access policies.

**Summary**

This chapter examined three key serverless platforms for application development on OCI. The main characteristic of all these systems is that they involve no infrastructure management and can automatically scale to meet the rising needs of your application. The three services examined in this chapter are each aimed at a different type of use case. Container Instances makes the process of running containers easy by completely removing infrastructure management and letting you focus on your containers, while still providing features such as resource throttling and configuration. OCI Functions provide an even more agile experience, in which the OCI Functions SDKs and CLI directly work with your code to make them fully managed functions that can scale for full elasticity. They also integrate with multiple OCI services and external triggers to help you create reactive applications. Finally, OCI Service Mesh introduces a method to transparently add new capabilities to your existing Kubernetes-based applications with the help of a fully managed service mesh. The OCI Service Mesh, based on the Envoy proxy, lets you easily add features such as mTLS, traffic shaping, and observability to your applications without requiring any code changes. Serverless platforms and services often act as a way to build scalable integrations among multiple systems or to enhance existing systems with new capabilities in a cloud native manner.

**References**

[1](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch07.xhtml#ref7_1) OCI Image Specification: <https://github.com/opencontainers/image-spec/blob/main/config.md>

[2](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch07.xhtml#ref7_2) Functions QuickStart on Local Host: <https://docs.oracle.com/en-us/iaas/Content/Functions/Tasks/functionsquickstartlocalhost.htm>