***Chapter 8*: Running a Serverless Application on Oracle Cloud Infrastructure**

Serverless computing is *the next frontier for modern app development!* The serverless framework abstracts the computing resources from the developers so that they don't have to deal with compute or other components to run their code.

But wait – let me tell you a secret: there are still servers involved. Lots of servers and networking gear and storage, too. We just don't want to have to think about them anymore.

In the last two decades, we have shifted our focus from Waterfall model development for a big fat monolithic application and deploying it to a physical server in our on-premises IT, to a DevOps model of delivery for our microservices-based application, running in small containers on the public cloud. This is a big change, not only in terms of technology but also in terms of a quick *go to market* strategy and continuous delivery for new features.

But agility is still not an easy problem to solve. There are various business challenges that we are constantly trying to solve, such as estimating the number of servers in the cloud, paying for idle workloads, and so on. We are also facing growing code and scaling applications, and then our nightmare chasing us in terms of vendor lock-in, dependencies, and so on.

A **Functions as a Service** (**FaaS**) platform is the beating heart of a serverless architecture. A function is a simple piece of code that does one job well. It takes some input (usually on standard in) and, optionally, writes some output, usually to standard out. We are not talking about functions in the *functional programming* sense here as these functions can have side effects; rather, they are self-contained units of work.

Similar to other OCI services, Oracle Function is also a multi-tenant managed service. Developers just focus on their business logic rather than figuring out how the infrastructure components work together. Oracle Function has been designed to be highly available, scalable, and secure. Developers can also run their code based on cloud events that come from OCI, and they pay only for the resources that were consumed during the execution of their code.

Oracle Function is based on the open source Fn project. This gives us a strong multi-cloud (no lock-in), on-premises, and local development story.

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

* Understanding the notion of serverless computing
* Understanding Oracle functions
* Deep diving into Oracle Function
* Understanding the event-based usage of Oracle functions

**Understanding the notion of serverless computing**

Functions are deployed as a single unit to a FaaS platform. This platform then deals with provisioning the underlying infrastructure, deploying your function code, scaling up and down, resilience and reliability, billing, and security (authentication, authorization, and isolation). It has to do all this blazingly fast, at huge scale and for any language/platform that you wish to use.

The primary focus of a serverless platform is your code. The FaaS platform encourages you to break up your application into small, isolated parts. This is great for developers as small, isolated functions are easier to reason about and manage.

Your FaaS platform will enable your app to scale organically per request, without you having to write any special code to handle it. And not just the gentle seasonal changes in demand that a retail business might experience but also sudden, dramatic surges in demand caused by your app going viral.

With this, we have thickened the plot of what serverless and functions are. Now, we will look at the various distributions of serverless platforms.

**Understanding the importance of Oracle Function**

Serverless platforms attract and retain application developers by making it extremely easy to consume the underlying resources. Application developers love serverless platforms because they enable them to focus on their own business problems, instead of managing infrastructure. Let's divide these important benefits into different categories:

* **Infrastructure abstraction**: The serverless platform enables application developers to build, deploy, and scale their apps without thinking about the infrastructure underneath. This frees developers to focus on their own business needs by removing a whole set of challenges, such as compute node management, OS management and patching, load balancing traffic, designing apps to scale across multiple nodes and availability domains, and much more.

The following diagram shows an overview of how functions have evolved over the last two decades, going from bare metal computing environment to running a fraction of the code:

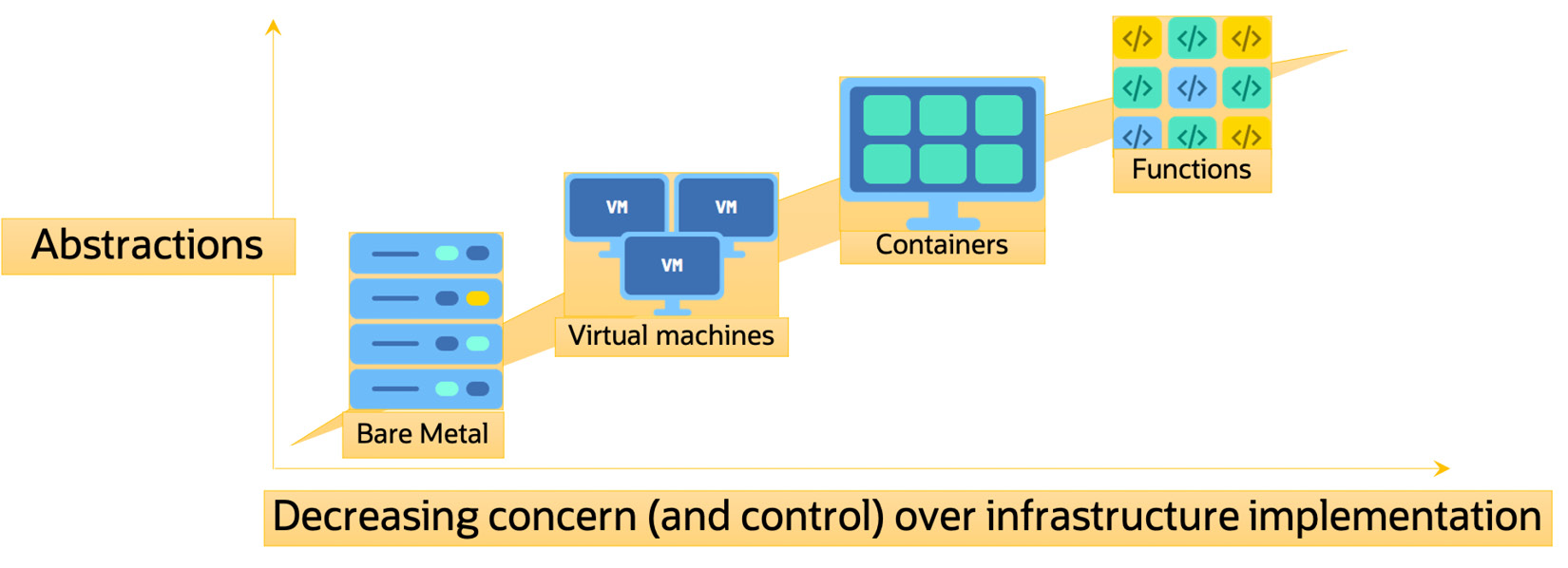


Figure 8.1 – Abstraction of running code in a function

* **Architecture simplification**: Adopting a serverless architecture can greatly simplify application complexity, particularly if the application was already designed in an event-driven fashion.

The following diagram shows an overview of how functions have been instrumental in simplifying the DevOps strategy of computing:

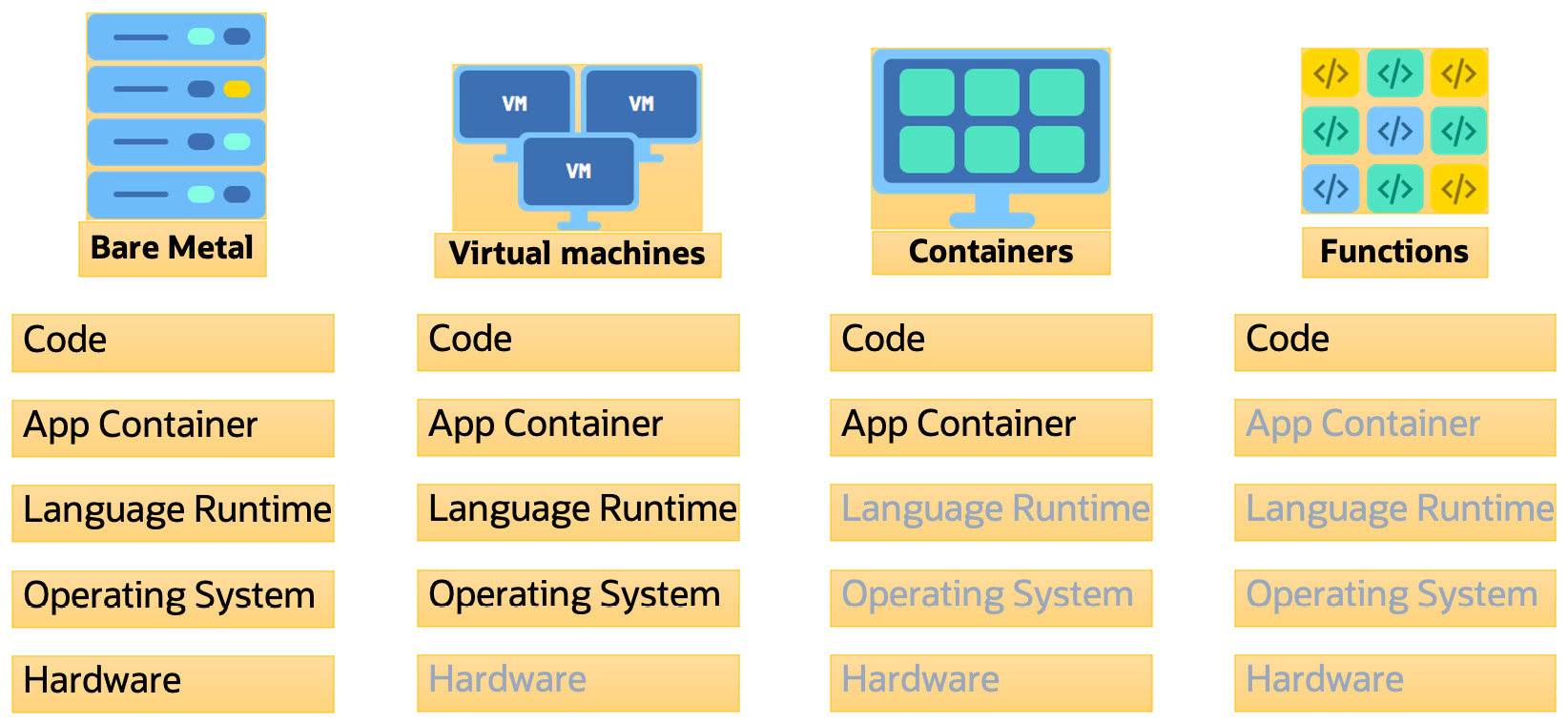


Figure 8.2 – Simplifying DevOps using functions

* **Cost control**: Application developers will only pay for the resources they've consumed, such as CPU cycles, RAM consumed, and I/O. This is billed in sub-second increments, with no costs for idle time.
* **Run anywhere**: Customers want to avoid vendor lock-in by adopting a multi-cloud strategy. Oracle's adoption of Fn as the underlying serverless platform means that their customers can run the exact same code in another cloud, or on-premises, similar to how you can run Oracle functions.
* **High-fidelity local developer experience**: Enable rapid and reliable software engineering by enabling developers to develop and test locally, without the time/cost of deploying to a cloud environment during dev/test cycles.
* **Run arbitrary code**: Run code in any language and any framework.
* **Consistent, isolated, repeatable**: Code runs in the same isolated execution environment every time.
* **Logging and metrics built in**: Oracle functions further simplify the design and operation of an application by providing a pre-defined set of metrics and centralized logging, both of which require no code.

In this section, we discussed the importance of Oracle Function and its architecture, but we haven't discussed why you should use Oracle Function. We'll do that in the next section.

**Understanding the use cases of Oracle Function**

One of the main use cases of the serverless platform is to be able to run event-driven code based on cloud events; that is, *run a function in response to an event occurring in OCI*.

Integrating Oracle Function into the broader OCI ecosystem provides added value to OCI customers. Oracle Function has been integrated with OCI Event Service to allow customers to trigger their functions in response to events happening in OCI.

An example for this use case is a customer configuring Oracle Function to run a processing function every time a new object arrives in their object store bucket.

Another important use case of Oracle Function is event queue processing; that is, *run a function in response to an item in a queue*.

This use case for Oracle Function is the ability to run a function in response to an item being queued on a queue.

Another use case of running Oracle Function is against a webhook event. With the rapid increase of DevOps tools, it is more common to run code against a webhook event; that, is *run a function in response to a webhook event from an external web service, such as GitHub*.

This use case focuses on the ability to execute a function in response to a webhook. The user of a service can send an outgoing HTTP webhook request to execute functions (for example, GitHub, PagerDuty, and so on).

So far, we have discussed the basics of serverless computing, the importance of it, and why and where you should apply serverless functions. We'll look at how to create and use Oracle functions in the next section.

**Creating and using Oracle functions**

Oracle's serverless platform is pretty simple and due to its adoption of open source Fn as the base underline platform, it avoids any vendor lock-in as well. It runs in five different steps:

1. Developing a function either locally or on an OCI cloud shell
2. Building and packaging the code using a container
3. Pushing the function image to the OCI registry
4. Configuring how this function will be triggered, either based on HTTP calling or an Event, Stream, or Timer
5. Paying for the code execution time only

The preceding workflow can be seen in the following diagram:

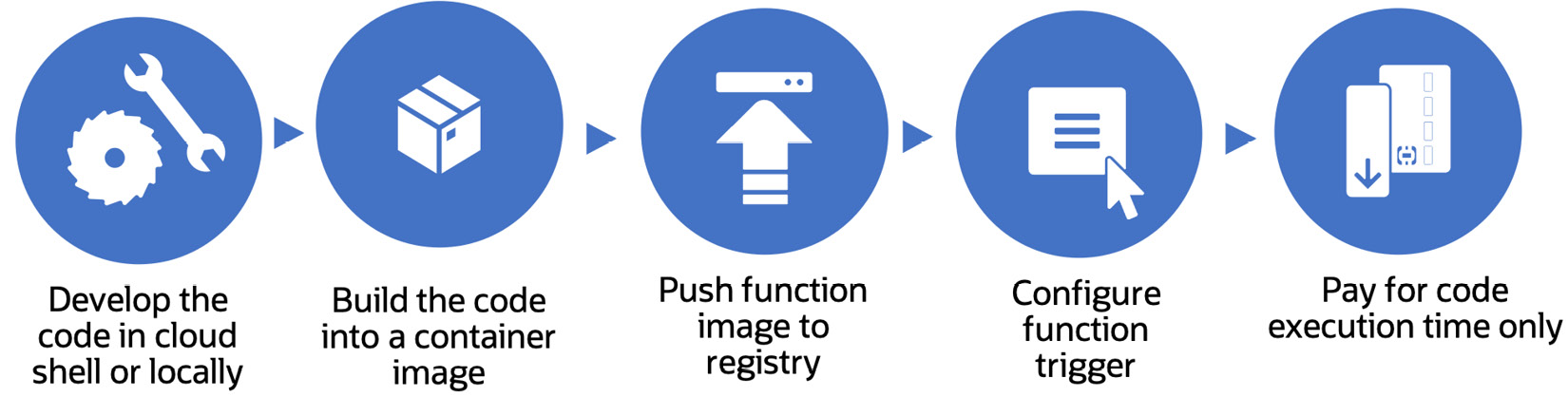


Figure 8.3 – Life cycle of Oracle Function

Once you've created the function and uploaded the image to the OCIR, you can invoke it in four ways:

* Developers can use the Fn CLI to invoke a function.
* Developers can use OCI SDKs to invoke a function.
* Developers can send a signed HTTP request to the invoking point of a function.
* Developers can use other OCI services to invoke a function as well, such as the OCI Events service.

OCI uses IAM principles such as an IAM policy to verify whether the invoked command entity is entitled to use Oracle functions or not. If there are adequate permissions in place, then the request will be passed to the backend infrastructure that runs the Fn project. The Fn project will use the function definition that a user has written and identify which Docker image it has to pull to run that code. Once it pulls the image from the Docker registry, it compiles the code and runs the container on a dedicated instance that is connected to the particular tenancy.

As this function runs on the subnet that was specified to run, it can read and write to other OCI resources that are connected to the same subnet. It can access other shared OCI resources as well, such as OCI object storage. Since it has long-running functions, OCI can run a function for as long as 5 minutes. But that is something you have to specify in the **func.yaml** file before deploying the function.

You can either use the OCI logging service and service connector to store the function logs within the OCI itself, or you can send the logs to another centralized log server.

Oracle Function uses Docker containers to run the code inside a dedicated Docker host. However, if the container is idle for some time, then that container will be removed and a new container will be started.

Oracle Function will remove the Docker container once it has finished running the code inside it, but only after a brief period of idle time. However, if a request is received by the Oracle function within the idle time threshold, then Oracle Function will use the same Docker container to run that code again, instead of creating a new one. If there is a parallel request that comes in, then Oracle Function will scale this Docker container to run the same code in parallel.

Using security first measures, Oracle Function has been tightly integrated with OCI **Identity and Access Management** (**IAM**) principles. This means that to send a call to an OCI API endpoint, you can use an **instance principal**. Let's create a function application that will list the compartments that you have in your tenancy.

In this exercise, you will create a dynamic group, add a policy that must send signed calls to the OCI API endpoint, which is an Oracle Function application, and trigger the function from the Function command line. You will use the OCI Cloud Shell to perform this whole lab.

First, we'll create the dynamic group and then create the policies for the functions. To do so, we need the compartment OCID, which is where we will deploy our function:

1. Sign in to the OCI console.
2. Open the navigation menu, select **Identity**, and then **Compartments**.
3. Find your desired compartment from the list, hover over the cell in the OCID column, and click **Copy** to copy the compartment OCID to your clipboard.
4. Store the compartment OCID as you will use it soon.

Now, let's create the dynamic group:

1. To create a dynamic group, open the navigation menu, select **Identity**, and then **Dynamic Groups**.
2. Click **Create Dynamic Group**.
3. For **Name**, enter **functions-dynamic-group**.
4. For **Description**, enter a group with all the functions in a compartment.
5. To select the functions that belong to the dynamic group, write the following matching rule, which includes all the functions within the compartment you are going to create your application in:

All {resource.type = 'fnfunc', resource.compartment.id = 'ocid1.compartment.oc1..example'}

*NOTE*

*Make sure you replace the preceding value with the compartment OCID that you stored earlier.*

At this stage, we will define some policies that are required for functions:

1. Under **Governance and Administration**, go to **Identity** and click **Policies**.
2. Choose the root compartment from the compartment menu.
3. Click on **Create Policy**.
4. Provide a name and description.
5. On the **Policy Builder** screen, select **Customize (Advanced)**.
6. Add the following policy statements:

Allow group func-pol to use cloud-shell in tenancy

Allow group func-pol to manage repos in tenancy

Allow group func-pol to read objectstorage-namespaces in tenancy

Allow group func-pol to read metrics in tenancy

Allow group func-pol to manage functions-family in tenancy

Allow group func-pol to use virtual-network-family in tenancy

1. Click on **Create**, as shown in the following screenshot:

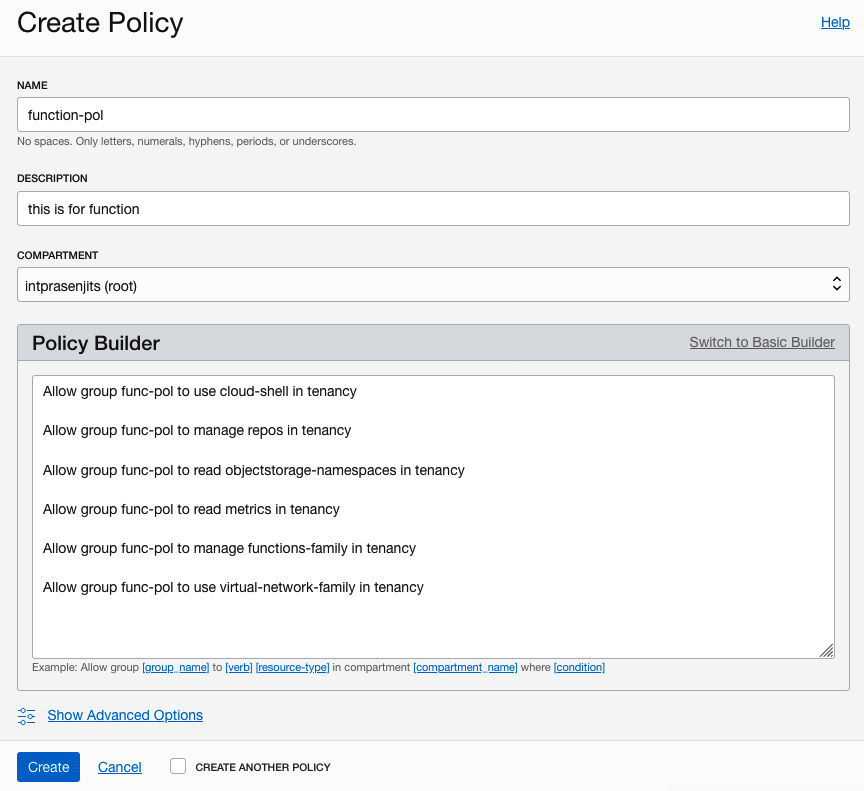


Figure 8.4 – Policy statement for Oracle Function

You've just created the prerequisites. Now, let's create the actual application:

1. Under **Solutions and Platform**, go to **Developer Services** and click on **Functions**.
2. From the compartment dropdown, select the **Compartment** area where you want to create the application.
3. Click on **Create Application**.
4. Provide a name. Then, select the VCN and subnet where you want to deploy this function.
5. Click on **Create**, as shown in the following screenshot:

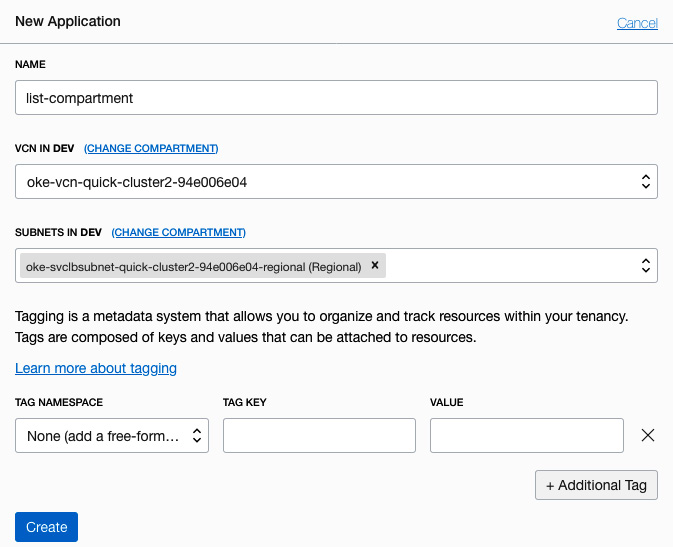


Figure 8.5 – Creating a new function application

Next, we need to launch Cloud Shell:

1. On the **Applications** page, select the application that you have just created.
2. Click on the **Getting Started** link from the **Resources** section.
3. Click on **Cloud Shell Setup**.
4. Click on **Launch Cloud Shell**, as shown in the following screenshot:

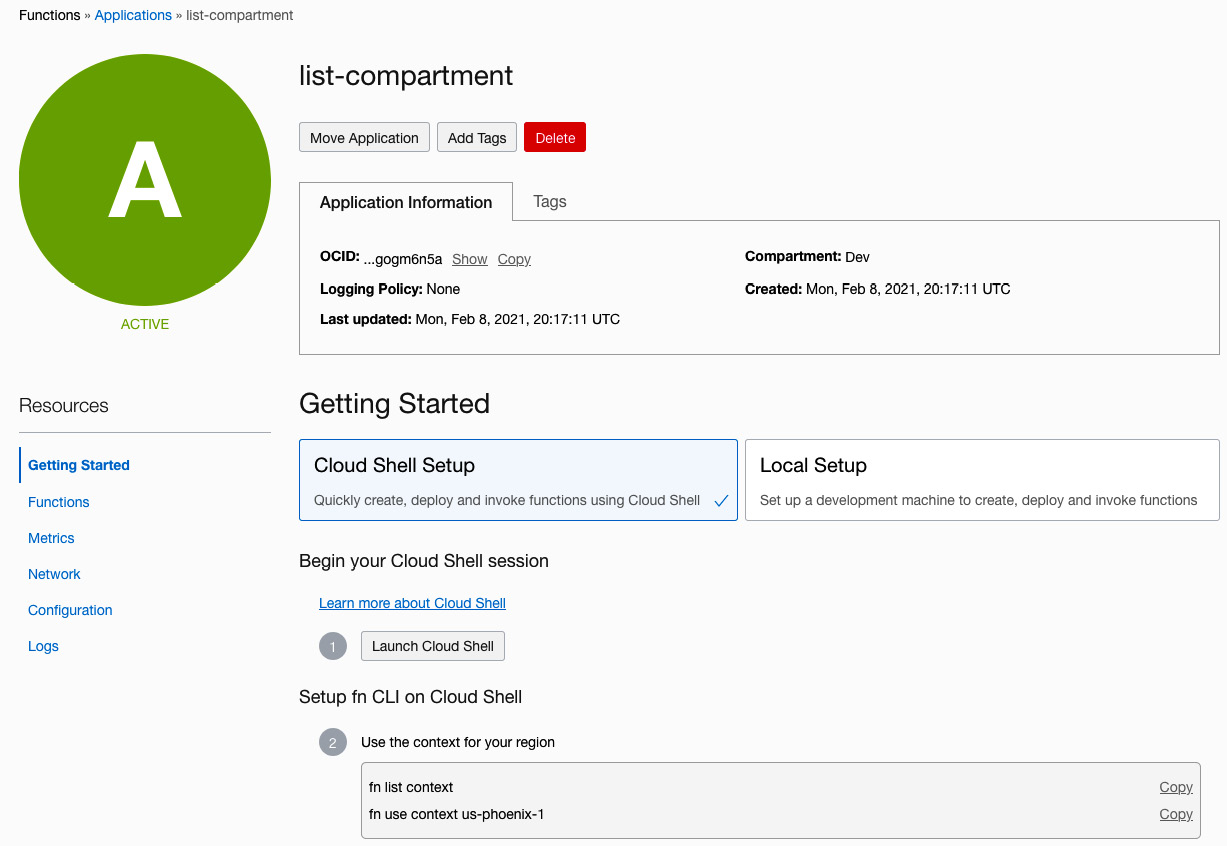


Figure 8.6 – Opening Cloud Shell

1. From this section, run the commands that you can see in the **Setup fn CLI** section of Cloud Shell.
2. These commands are tailor-made for your tenancy, so you can just change some of your parameters, such as your OCI Auth Token, to set up the Cloud Shell environment.
3. Now, we will deploy our application code. Oracle already has some sample code in their GitHub sample code repository for Function, so we will use that.
4. Download the repository by cloning the GitHub repository. To do this, run **git clone**<https://github.com/oracle/oracle-functions-samples>.
5. Go inside the sample code directory by running **cd oracle-functions-samples/samples/oci-list-compartments-python/**.
6. Run **fn -v deploy –app list-compartment**, as shown in the following screenshot:

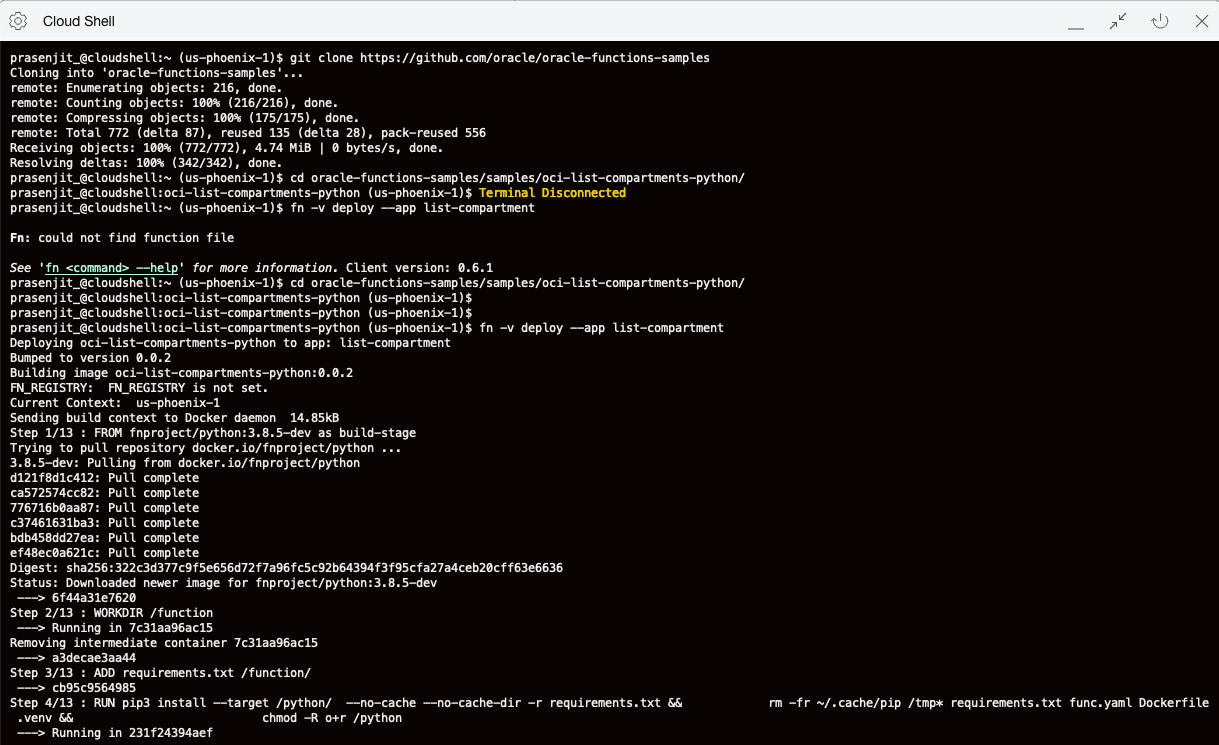


Figure 8.7 – Deploying the function code

1. Once it has been deployed, you can invoke the function by calling it using the same **fn** command line. Run **fn invoke list-compartment oci-list-compartments-python**.
2. This will show you the list of the compartments that you have in your tenancy.

With that, you've learned how to use Oracle functions to run a fraction of your code on demand, as well as how to send signed calls to an OCI API endpoint using an instance principal. In the next section, we will talk about some advanced use cases and deep dive into various Oracle functions.

**Deep diving into Oracle functions**

We will deep dive into a couple of topics before talking about event-based Oracle Function deployment.

First, let's discuss filesystem access within the Docker container where the function runs:

* Function code running inside the Docker container has read access for all the files and directories within the filesystem.
* Function code only has write access to the **/tmp** filesystem.

This means that when you write the code and want to perform a file operation, such as downloading a file inside the docker container, you need to make sure that you only download that file to **/tmp**.

Although this **/tmp** is writable, it has a size restriction, and it is proportionate to the amount of memory that you allocate to this function. If you allocate 128 MB of memory to the function, then you can get 32 MB of allowed space on your **/tmp**. For 256 MB of allocated memory, it will be 64 MB, for 512 MB of allocated memory, it will be 128 MB, and for 1,024 MB of allocated memory, you can get 256 MB of allowed space inside **/tmp**.

Another thing that you have to be careful about is the Docker container permissions when you bring in your own container to run as a function on the OCI. To run your code, Oracle Function creates a container. The container's functionality depends on the Linux permission schema using a **user ID** (**UID**) and a **group ID** (**GID**). So, when you create a container outside of Oracle Function's perimeter and do not provide a **USER** parameter within the Dockerfile while creating the container, Docker will allow this container to run all the processes as root and add all the default capabilities of a root user. This is a dangerous combination for your security postures.

To mitigate this security flaw, Oracle Function adds a UID and a GID, both named **fn** and both with an ID of **1000** . As you can imagine, there is no privilege attached to this UID and GID. As a result, if your code depends on the elevated permissions to perform any operation, such as **sudo** or **setuid**, then it won't be able to run.

As a best practice for bringing your own container to Oracle Function, you should add the **fn** UID and GID while creating the container to avoid any other pitfalls. You can see an example of this in the following screenshot, where the **RUN** parameter is adding this to the end, after running all the commands:

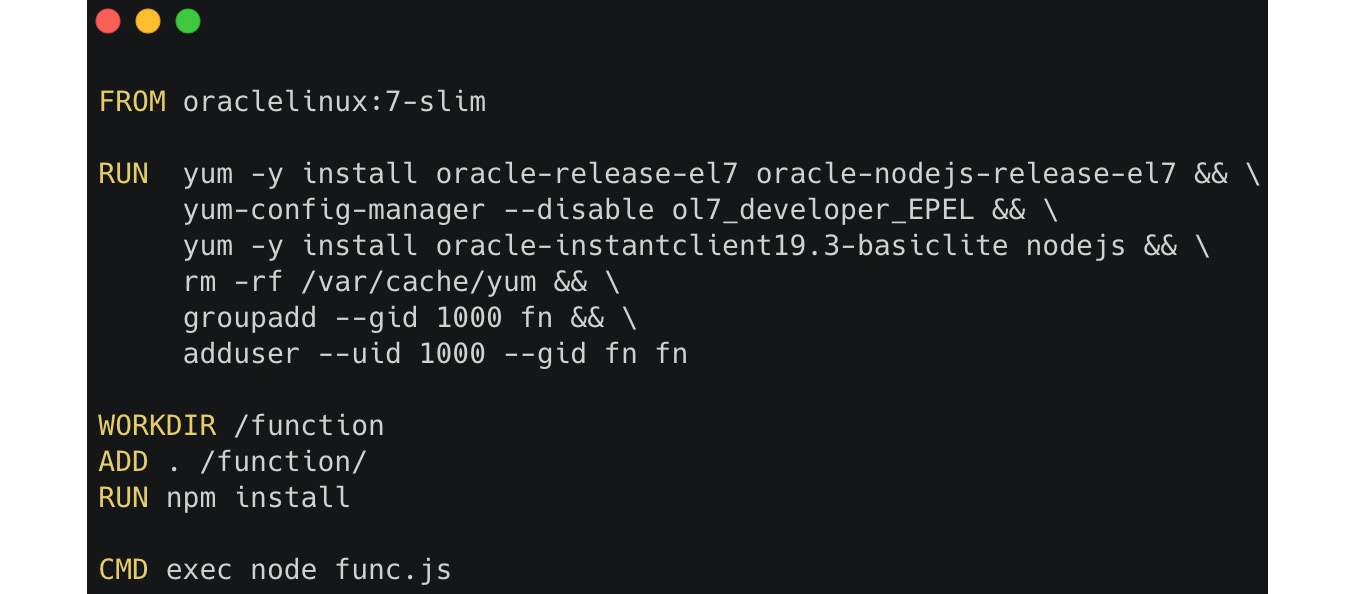


Figure 8.8 – Deploying the function code

Oracle Function uses **Resource Principal** to allow your code to use other OCI resources. Resource Principal works the same as Instance Principal, which we looked at in [*Chapter 2*](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9781800566460/B16798_02_Final_NM_ePub.xhtml#_idTextAnchor028), *Understanding Identity and Access Management*. You need to create a dynamic group and add the function to it; then, you must write a policy to grant access to other OCI resources.

Within your code, you must call a **resource principal provider**. This uses a **resource principal session token** (**RPST**), which gives you permission to access other OCI resources. This token is only valid for the resources that you have granted through the policy.

This token is cached for 15 minutes. This means that if you want to make any changes to the policy, you need to wait 15 minutes before that change will take effect.

Oracle integrates this resource principal provider inside all its SDKs, so use any of the given SDKs to authenticate against OCI API within your function code. However, if you don't use any of the supported SDKs, then you need to create a custom resource principal provider.

The following diagram shows a logical representation of this. Here, a REST token is being provided to the function by RPST, so that it can authenticate itself against other OCI API endpoints:

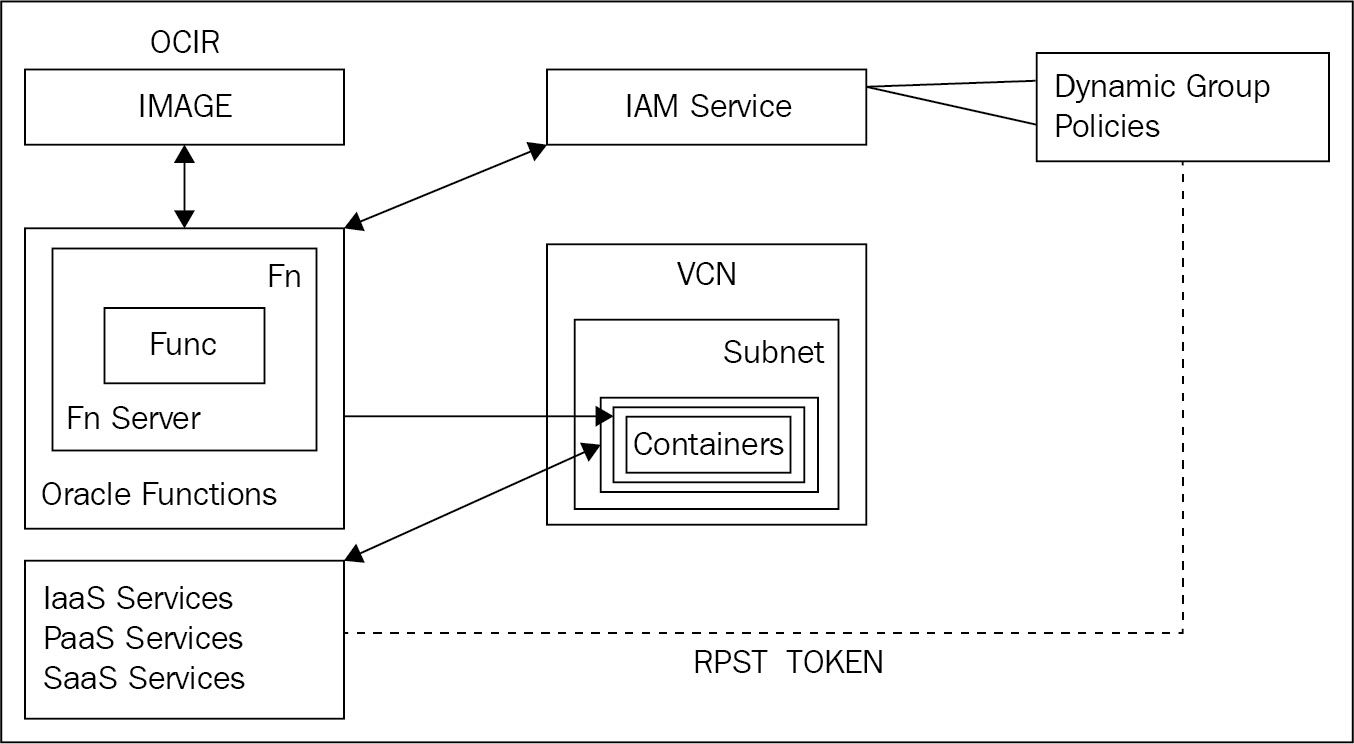


Figure 8.9 – Accessing other OCI resources from Oracle Function

In this section, you have learned about various topics surrounding Oracle Function. In the next section, we will talk about event-driven functions.

**Understanding event-based usage of Oracle functions**

Oracle functions have been carefully integrated with other OCI services, such as OCI Notification Service, Events Service, API Gateway, Object Storage, Autonomous Database, Key Management System, Compute, VCN, and others. The following diagram shows how Oracle Function can be integrated with other OCI services:

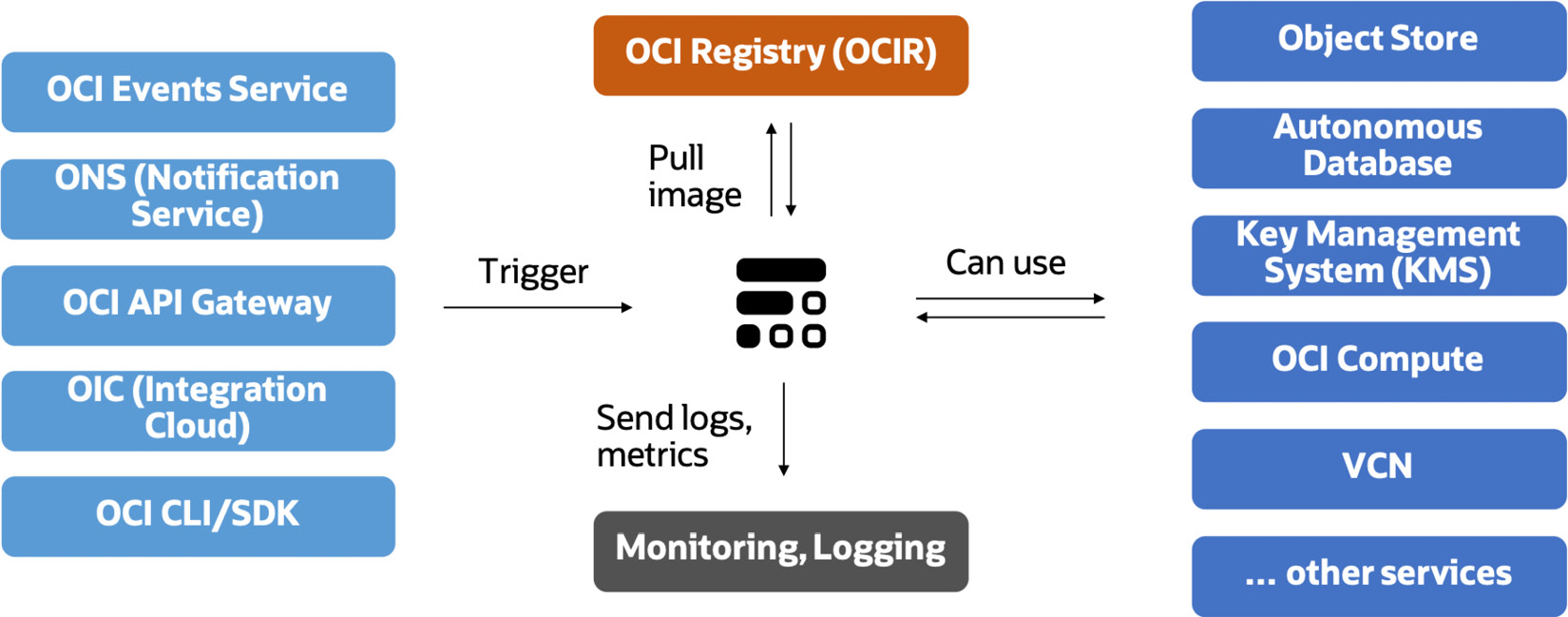


Figure 8.10 – Oracle Function integration with other OCI services

We have been looking at event-driven applications more since the rise of distributed systems. OCI Events Service is real time and uses event alerts and rules to track changes to cloud resources using open standard cloud events. OCI Events Service tracks user-initiated, system, and resource life cycle changes and allows admins to define rules and actions for event types. OCI Events Service can be integrated with OCI Streaming Service, **OCI Notifications Service** (**ONS**), and Oracle functions.

Let's look at a classic use case of OCI Events Service and integrating Oracle functions with it. The following diagram shows a high-level overview of this use case:

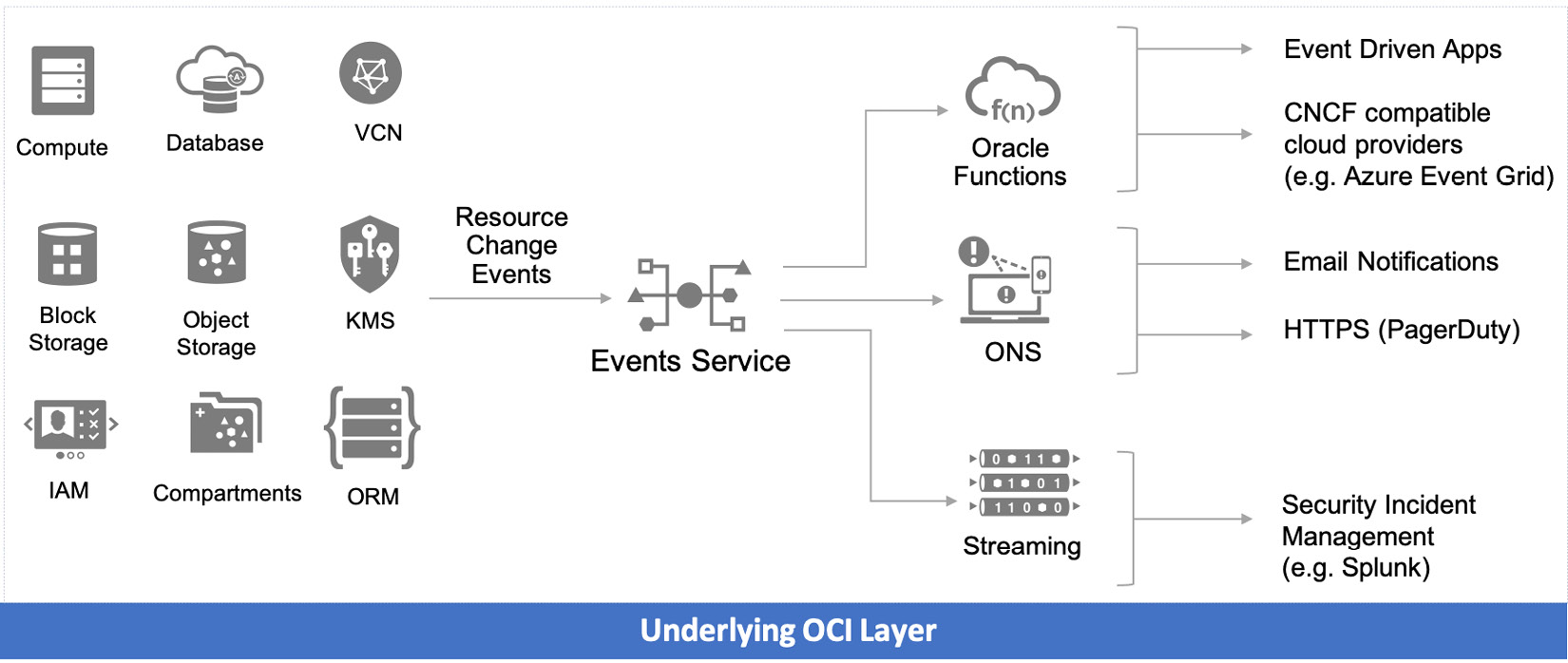


Figure 8.11 – OCI Events Service and Oracle Function integration

In the preceding diagram, you can see that how OCI Events Service allows you to easily respond to **Autonomous Transaction Processing** (**ATP**) via functions. In this use case, we are creating a new ATP instance, which will generate an event when the instance has been provisioned. This event is processed by Events Service to trigger a function, as well as send a notification. The function uses the data in the event to create database schemas, tables, rows, and other **data definition language** (**DDL**) constructs. Notifications Service can send event notifications via email and PagerDuty. Let's look at some other common use cases of event-based functions.

You can enforce corporate security policies and governance rules via Events Service and integrate this with Oracle functions. If you want to ensure that all your user-provisioned cloud infrastructure complies with corporate security policies, then define a set of audit functions that validate the configuration of different OCI resources (VM, network, and database). Then, define OCI Events Service rules to trigger audit functions on creation and configuration changes. This deployed audit function will tear down resources that do not meet corporate security policies; for example, if a compute instance has a public IP, then terminate the instance. The following is a logical diagram of how this can be achieved:

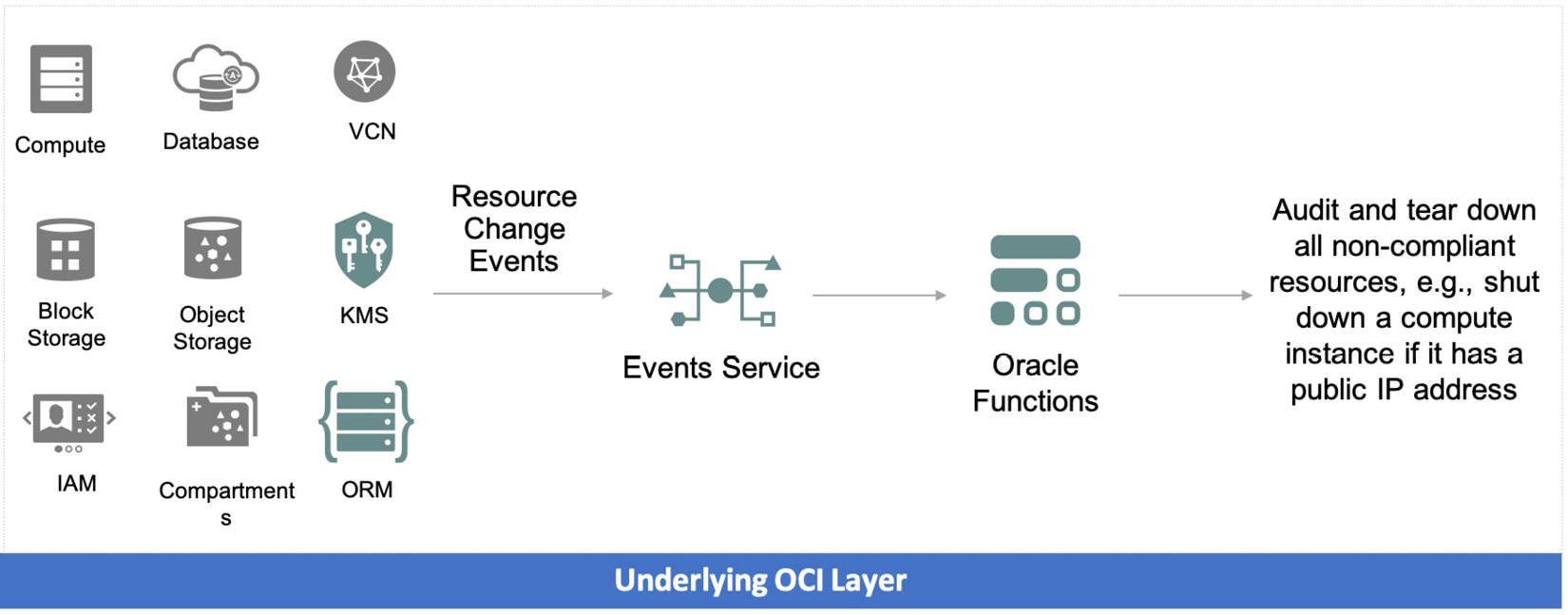


Figure 8.12 – OCI Events Service and Oracle Function integration for enforcing governance rules

Another use case could be ingesting access logs in security incident management. If you want to ensure that all your access and audit logs are sent to a central security incident management system such as Splunk, then define a function to parse the log files and post the log data to the customer's Splunk instance. You also need to define OCI Events Service rules to trigger the function on creation and develop the functions that parse the log file and send data to Splunk. The following diagram shows how this can be achieved:

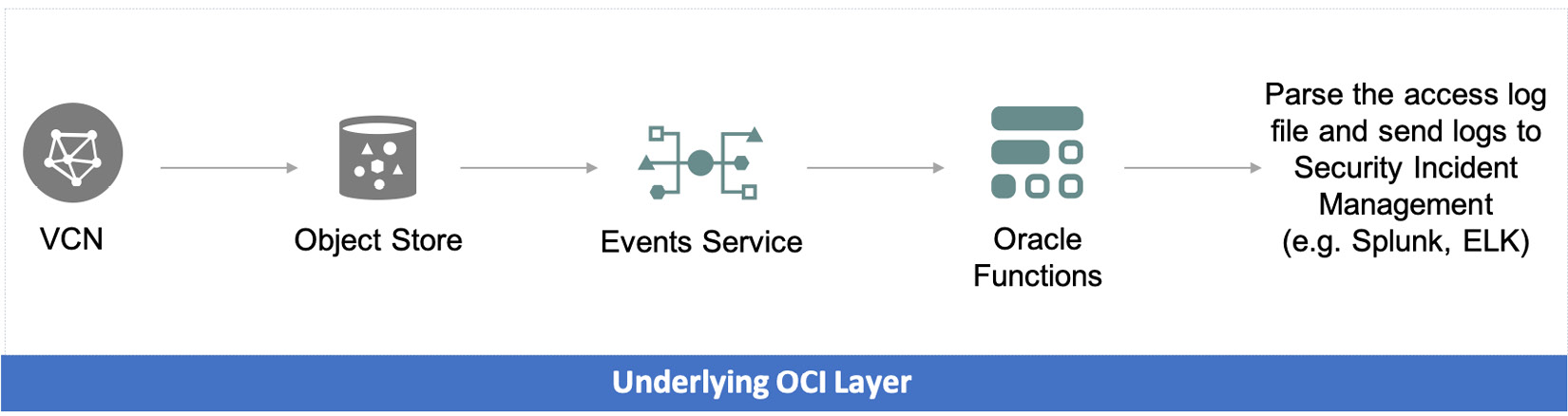


Figure 8.13 – OCI Events Service and Oracle Function integration for sending access logs to security incident management systems

With that, you've seen how Oracle functions can be integrated with OCI Events Service and how they make a perfect pair for event-driven applications.

**Summary**

In this chapter, you have learned about the nuances of serverless applications, their use cases, and why serverless platforms are getting popular among enterprises. To reiterate, it is a simple piece of code that does one job. It takes some input (usually on standard in) and, optionally, writes some output, usually to standard out.

You also learned how Oracle has adopted the open source **fn** project to break free from the cloud lock-in of serverless applications. You also looked at the advanced concepts surrounding Oracle functions and how they can be taken further with the integration of OCI Events Service.

In the next chapter, you will learn how to use Terraform and Ansible code to maintain your OCI Infrastructure as Code.