**8**

**Observability**

Cloud native architectures and paradigms introduce new challenges and opportunities related to systems management and observation. Monolithic applications are being broken into microservices, applications and their runtime environment and configurations are getting packaged as containers, and infrastructure is becoming ephemeral. Just as these general paradigms are changing, so should the techniques and tools to observe and monitor them. The distributed nature of these application architectures requires the ability to observe a multitude of systems in isolation and correlate these metrics to get a complete view. The polyglot nature of these systems also give us opportunities to gather metrics and performance data from individual systems using tools and techniques that are optimal for each service or unit, without needing to manage dependencies, choosing tools that work across a wide range of technologies, or deal with cascading effects from other systems. The same notions apply to infrastructure. With ephemeral infrastructure, it can be daunting to track infrastructure changes across time and other dimensions, such as the correlated impact on applications, cost, and fault tolerance.

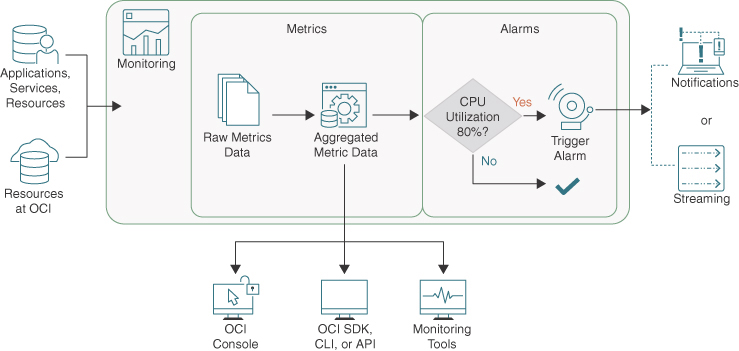
Observability can be defined as the capability to track and trace the happenings within your systems, at both the application and infrastructure levels. This data helps you understand and reason about behaviors you see within the system. In many cases, it is not enough to just observe and reason about observed behavior. You also need to be able to predict future behavior.

Oracle Cloud Infrastructure (OCI) offers several tools and solutions to observe your applications and services. In keeping with OCI’s open approach to the platform, you can also bring in your own tools that you might be more comfortable with. These include tools and platforms that gather metrics or logging, use agents or ingest log data, or use tools such as Extended Berkeley Packet Filter (eBPF) to gain insights into your workloads. This chapter takes a look at the various services and some popular open-source tools that you can use to observe your cloud native applications deployed to OCI.

**OCI Monitoring**

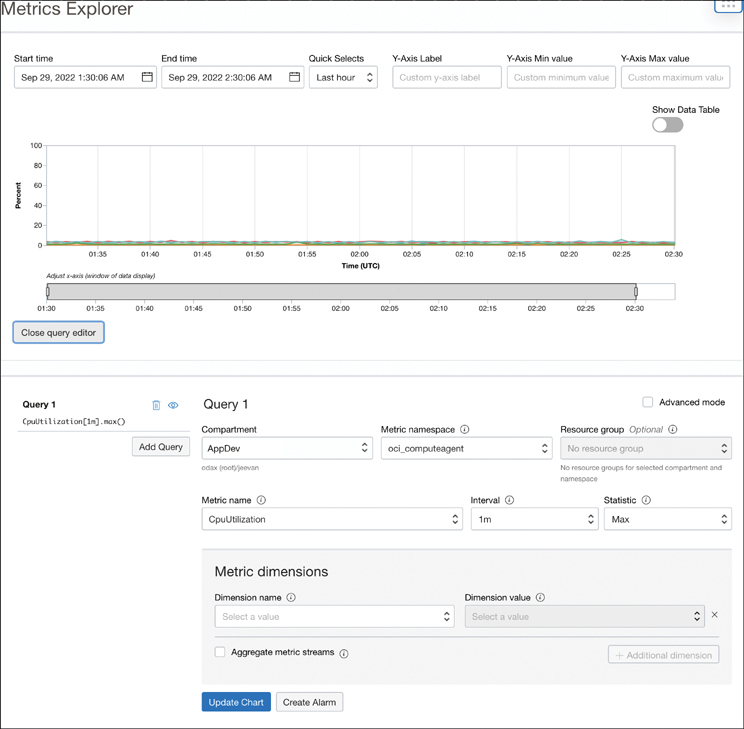
OCI Monitoring is a service that keeps track of your resources across OCI using resource metrics, and then alerts you to various conditions using alarms. A metric is a data point related to the health, capacity, or performance of any resource in OCI. These data points have time stamps, dimensions, and other metadata. Metric data can originate from OCI services or can be published by your own applications. The most common and trivial example of a metric is the CPU and memory utilization of a compute instance. Metrics can be unique as well, such as the number of unschedulable pods in an OKE cluster or the number of vulnerabilities found by a vulnerability scan. Several OCI services report metrics out of the box; the most common of these are the compute instance metrics. Every compute instance reports metrics such as its own CPU and memory utilization. In this example, the CPU utilization value for an instance is the metric (a measured value); it is accompanied by the time stamp for that measurement. A metric definition adds data to the metric, such as metadata that indicate the unit of measure for the value in the metric, and dimensions that are attributes for that measurement. The compute instance OCID, its availability domain, and its compartment are examples of dimensions that can be added to the metric and its metadata to form a metric definition.

Additionally, a metric definition includes a metric namespace that acts as a grouping construct for identifying and grouping the class of resources where the metric originates. The raw metric data emitted from the various sources, such as the services in OCI and the various applications, is consumed and aggregated by the monitoring service. [Figure 8-1](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch08.xhtml#ch08fig01) shows how the metric data flows through the monitoring service.



**Figure 8-1** Typical Flow of Observability Data in OCI

Once the raw metrics have been ingested by the monitoring service, it can be queried to get the aggregated data. These queries are the primary mechanisms through which data is extracted from the service and are the most fundamental way to monitor your resources in OCI. Metric queries are written in Monitoring Query Language (MQL), an intuitive query language that is similar to natural language. The OCI console includes Metrics Explorer, an intuitive query builder that helps you construct the queries in a visual environment (see [Figure 8-2](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch08.xhtml#ch08fig02)).



**Figure 8-2** The Metrics Explorer in OCI Offers an Intuitive Visual Query Builder for Querying Metrics

The Metrics Explorer can help you construct most common queries visually. Besides the visual query builder, the Metrics Explorer offers an Advanced mode that enables you to write your own MQL queries. Understanding MQL can be useful: With it, you can build advanced queries and use the monitoring service from the CLI, SDKs, and APIs. MQL queries have components such as the metric to be queried (such as CpuUtilization); an interval that determines the aggregation window of the raw metric data (CPU utilization for every 1 minute); and a statistic, which is the aggregation function applied (such as max). Taken together, a query of this nature would be expressing a request to get the mean CpuUtilization in intervals of 1 minute. The monitoring service would aggregate the raw data points for the CpuUtilization metric in windows of 1 minute and get the max value from each 1-minute window. The resulting aggregated data for the metric that is returned from the service is called a metric stream. The MQL syntax for this query follows:

CpuUtilization[1m].max()

This simple query shows the required parts of an MQL query—the metric, interval, and statistic. Different resources, such as the various OCI service or custom applications, emit different metrics; therefore, the metric you can query from the monitoring service depends on what has been ingested. The OCI documentation[**1**](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch08.xhtml#ref8_1a) covers every metric that is emitted by the various OCI services. The interval can be specified in minutes (as in 1m), hours (as in 2h), or days (as in 5d). The valid intervals in a query also change, depending on the time range for the query. For instance, a query pulling data for the past 30 days requires the minimum interval to be 1d. The service also supports several statistics, including counts, min, max, mean, various percentiles, and rates (rate of an occurrence). Beyond these required components, MQL syntax includes features that are much more powerful in targeting specific resources, grouping them and aggregating metrics. The general form of an MQL query that includes these optional components is as follows:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch08_images.xhtml#f0334-01a)

metric[interval]{dimensionname=dimensionvalue}.groupingfunction.statistic

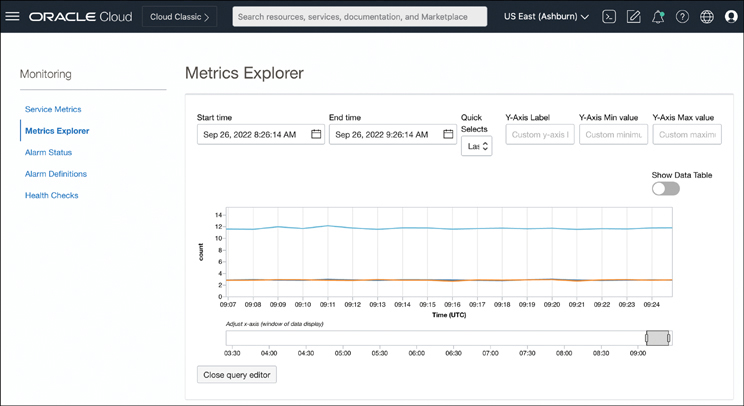
As you can see, MQL can include dimensions that you can use as a filter in your queries and grouping functions to determine how the metrics are aggregated. Dimensions are attributes from the metric data that you can use to filter your query and restrict it to certain data points with these attributes. For example, to look at the CpuUtilization in a single region or a single availability domain, you could use the dimension filter to restrict the query. Similarly, you might want to group the data in a way that makes it easy to work with. MQL supports grouping functions such as groupBy or grouping. The groupBy grouping function groups the metrics by one or more attributes, to aggregate them. The grouping function, on the other hand, aggregates all the metric streams from the query into a single aggregation.

To understand these components a bit better, let’s look at an example. Imagine that you want to track underutilized or abandoned OKE clusters in development environments. Reclaiming underutilized or abandoned cloud resources is a common cost optimization approach. Here, the intent behind the notion of *utilization* is fairly vaguely defined, and it could mean several things. It could refer to the cluster whose nodes exhibit the most/least CPU utilization, but a critical workload that experiences sudden volatility in usage or traffic could lead to false positives. Another way to define *utilization* could be to look for clusters that see very few API server calls. Clusters running production workloads generally see fewer API server calls than clusters running in a development environment. For this example, let’s use this metric to identify clusters that see comparatively fewer API server requests. The OKE service exposes a metric named APIServerRequestCount that you can use to accomplish this. Consider the following query:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch08_images.xhtml#f0334-02a)

APIServerRequestCount[1m].groupBy(clusterId).rate()

The query tracks the rate of Kubernetes API server requests in 1-minute intervals. Then it groups the data by cluster ID and produces one metric stream per cluster. On the console, this can appear as shown in [Figure 8-3](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch08.xhtml#ch08fig03).



**Figure 8-3** The Metrics Explorer Generating a Graph for a Custom Query to Track API Requests Across Multiple Kubernetes Clusters

In this example, you can see three lines, corresponding to the three metric streams. Each metric stream represents a single cluster because the query groups the metrics data by clusterId. If there were four clusters, the groupBy grouping function would group the data into four metric streams; the console then would show four lines representing the four metric streams. From the figure, it is clear that one cluster is seeing more API server requests than the other two. Having this query on a dashboard can help users and infrastructure admins quickly identify potentially underutilized resources on which to focus cost optimization efforts.

A *dimension* filter can filter the data points based on the dimensions available in the raw data. Consider the following example:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch08_images.xhtml#f0335-01a)

APIServerRequestCount[1m]{clusterId = "ocid1.cluster.oc1.iad.

aaaaaaaadcnof56t6ijrxbzjluujrjlwwvxz7u3guqxi7cyy5cemaalmxtlq"}.rate()

The preceding query demonstrates a dimension filter that filters out the metric data for all but one cluster, identified by its OCID, and gets the rate of Kubernetes API server requests. Dimension filters can use fuzzy filters to match multiple filter values and conditions. If you had a set of clusters and you wanted to track the same APIServerRequestCount metric across this set of clusters, you could use a fuzzy filter such as the one that follows:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch08_images.xhtml#f0335-02a)

APIServerRequestCount[1m]{clusterId =~ "\*cemaalmxtlq|\*jluujrjlwwv"}.

groupBy(clusterId).rate()

The query has three noteworthy aspects:

1. The =~ comparison operator specifies a fuzzy match.
2. The \* wildcard in the dimension filter matches zero or more characters. This wildcard is used to create a match with a partial OCID, much like a regular expression.
3. The | acts as the *OR* operation for the dimension filter values, which causes the dimension filter to do a fuzzy match on either of the two partial values.

The combined effect of this query is that you track the APIServerRequestCount metric in 1-minute intervals, and the query uses a dimension filter that uses fuzzy matching to filter the metric data that matches either of the two OCIDs. Then the results are grouped by the clusterId. This grouping generates two metric streams because the dimension filter also uses the clusterId to filter data that matches either of two clusters. Finally, the rate of requests is the statistic that is reported by the query for both metric streams.

Apart from the metrics that OCI services emit, your own applications can emit metrics to be ingested by the OCI Monitoring service. To use the service from within your applications, the easiest method is to use the OCI SDKs. The SDK includes code examples that demonstrate how this API can be integrated with your applications; it can be found on GitHub.[**2**](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch08.xhtml#ref8_2a) Metrics are secured by IAM, so to use the service, you also need to give access to your applications through policies such as the one that follows:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch08_images.xhtml#f0336-01a)

Allow dynamic-group ObservableApps to use metrics in tenancy where target.

metrics.namespace='AppMetricsNamespace'

Here, it is assumed that ObservableApps is a dynamic group that includes the instances where your application is running.

**Alarms**

Closely related to, and often used in conjunction with, metrics, are alarms. Alarms are simply notifications triggered by a specified condition. In the case of OCI, the monitoring service uses the notifications service to send a message when a condition is met. Alarms are also based on the same MQL queries. In addition to the query structure that has been examined so far, an alarm query has a condition. Consider the following example:

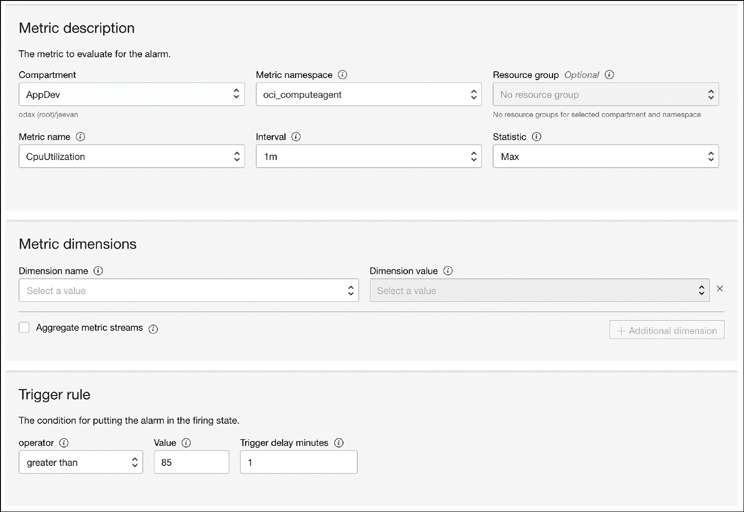
[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch08_images.xhtml#f0336-02a)

UnschedulablePods[1m].grouping().max() >= 1

This line tracks the number of unschedulable pods across clusters. If there is at least 1 unschedulable pod, it raises an alarm. The MQL syntax is the same as before, except for the addition of a condition, >= 1. This is the condition that the alarm continuously evaluates; if it becomes true, the alarm starts to fire.

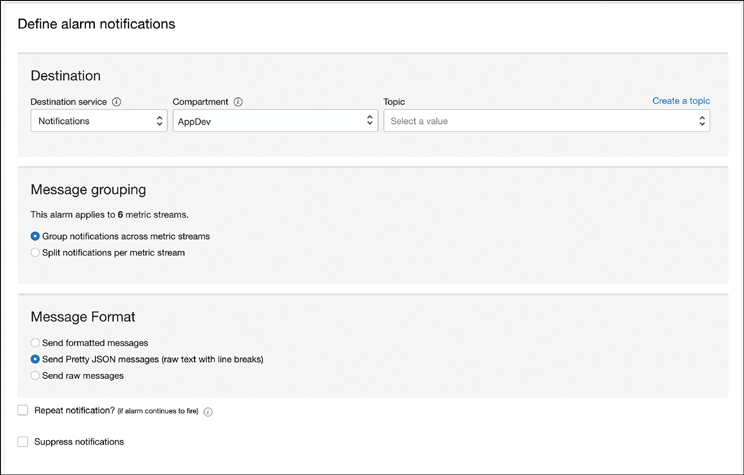
Every alarm definition also includes a severity that you can set, based on the criticality you assign to your query. The available severity levels are Critical, Error, Warning, and Info. Severity is an arbitrary value that the user creating an alarm can set, based on the perceived importance of the alarm. For example, an unschedulable pod might be an Info-level alarm in some applications but a Critical-level alarm for other applications. In practice, alarms often require tuning and these severity levels can change over time. For instance, when you start with a new application, you might set the alarms and severity levels based on an educated guess. Over time, however, you might notice that some alarms are noisy and get triggered too often without much impact to the application, so you could then adjust the trigger conditions and severity. You might also notice that you missed creating some alarms, as you find better predictors of trouble for your applications. This makes tuning your alarms an activity worth revisiting periodically.

Much like the Metrics Explorer, the OCI console has a visual interface to define and manage alarms. This interface enables you to create MQL queries both visually and with straight MQL (Advanced mode). Unlike the Metrics Explorer, however, here you can add conditions to the query and specify what actions to take when the alarm is firing. The actions could raise the alarm by using the notifications service or publishing it to the streaming service. [Figure 8-4](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch08.xhtml#ch08fig04) shows this interface to define alarms that are triggered based on metrics being collected continuously from OCI resources or your own applications.



**Figure 8-4** Creating Alarms That Are Triggered by Resource Metrics That Meet Desired Conditions

The notification service is a low-latency PubSub messaging service within OCI that supports durable messages with delivery guarantees. It provides topics and subscriptions, in which a service or application can send messages to a topic via the Notifications service. When a message is published to a topic, the Notifications service sends the message to all of the topic’s subscriptions. When the alarm destination is the notification service, you can choose to publish formatted text, pretty printed JSON, or raw JSON to the destination, as illustrated in [Figure 8-5](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch08.xhtml#ch08fig05). The choice depends on how you want to process the notification. For instance, if you want to directly create a PagerDuty subscription, perhaps formatted text is the most appropriate. On the other hand, if you need to send it to a function to perform some action, such as to trigger the creation of a compute instance to automatically resolve an alarm, the JSON-formatted message is likely more appropriate.



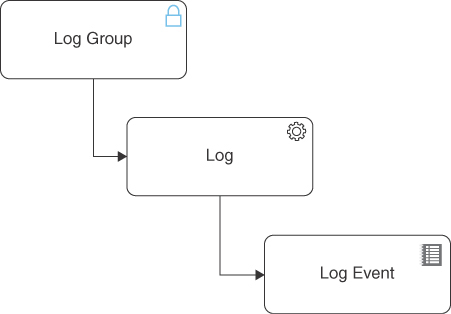
**Figure 8-5** Alarms Can Be Sent to the Notifications Service, Which Can Trigger Other Systems for Corrective Actions.

Similar to the Notification service, the delivery destination can be a stream in the OCI Streaming service. The streaming service supports only JSON message format.

**OCI Logging**

Logs are a fundamental source of insight into your applications and the services you use. Log data often includes diagnostic information that can help you troubleshoot issues and understand the performance characteristics of your application. Logs can also provide intelligence about the security posture of your cloud resources and keep track of changes to your infrastructure, its configuration, and who made them. The OCI Logging service provides a fully managed and scalable way to manage logs across applications and OCI services in your tenancy. The OCI Logging service provides features to store, organize, and search log data. Log data can come from OCI services when you enable them or from your own applications.

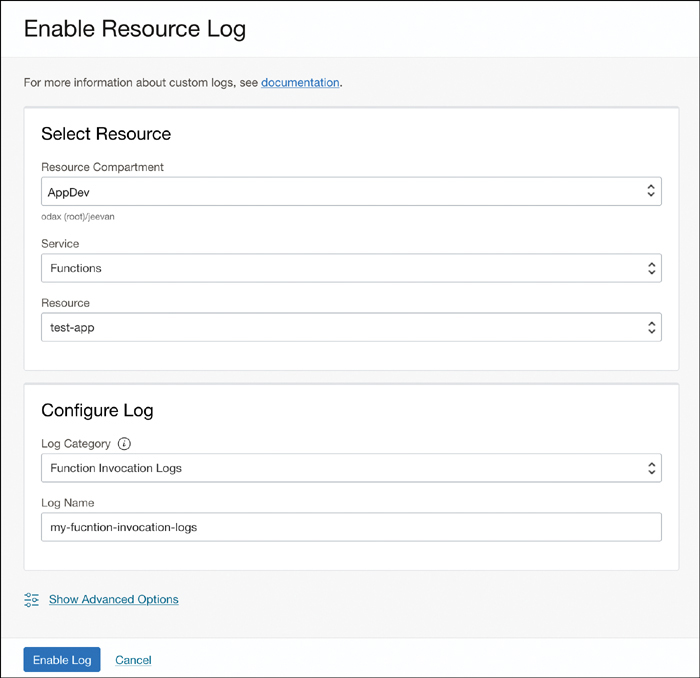
The first thing most people think about when hearing about logs is log files. Although the idea of representing and recording an event in an application or service in a file is still relevant, services such as OCI Logging provide a persistence mechanism that is efficient for searching, indexing, and applying data processing techniques. A *log* in this context is an OCI resource with its own OCID. You can think of this resource as a store for event data collected from an OCI service or your own applications. The actual log message that an application or service writes to standard out (STDOUT) is a *log event* that is captured in the log. The logging system captures the entire context of what happens when the service or application emitted the log message; this is why the log entry is called a log event. The information contained in the log event depends on the event type. All OCI services have log events that capture a great deal of contextual data about the event. Logs themselves are organized into *log groups*, logical containers for logs that help you organize similar logs. Log groups also help secure and manage these logs by applying IAM policies to log groups that determine who has access to the logs. For instance, you might want to create log groups based on the sensitivity of the log messages contained within logs. Logs that contain sensitive information can be grouped together into a separate log group. IAM policies can then be set up in such a way that access to this log group requires a specific role. This effectively restricts access to sensitive information and controls access purely through the use of policies. [Figure 8-6](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch08.xhtml#ch08fig06) shows this hierarchy of logging resources.



**Figure 8-6** Hierarchy of Logging Resources

**Service Logs**

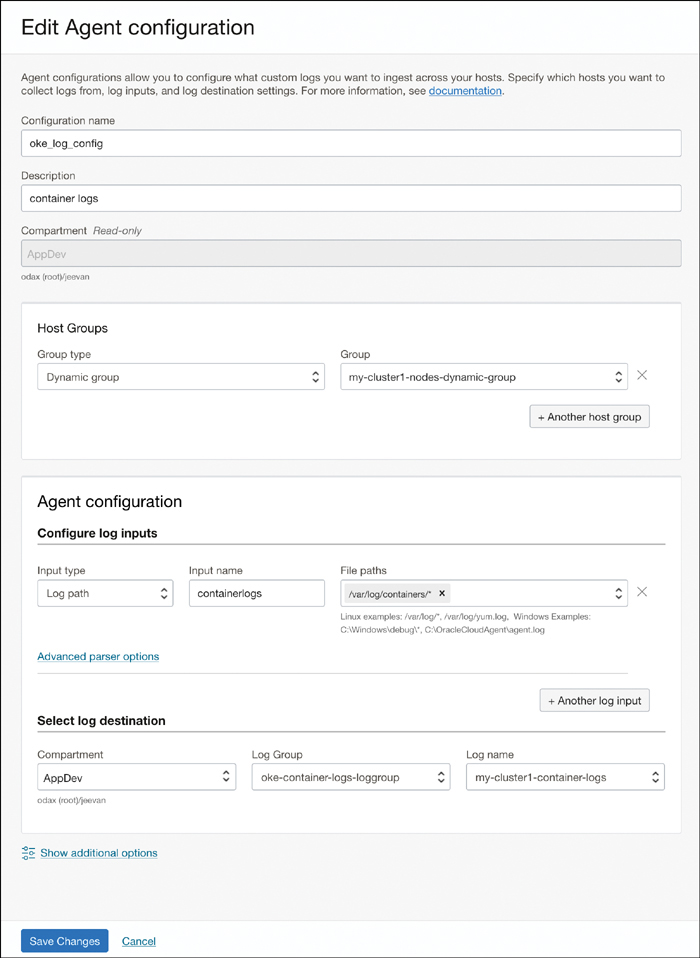
*Service logs* are logs from the various OCI services, and they are the most common type of logs in OCI. Service logs provide insight into the utilization, health, and performance of these services. These logs are collected by the OCI Logging service, which is made available to you and can be queried. Service logs are *enabled* in OCI Logging. This is simply because the services themselves emit log events that are discarded if they are not captured by enabling the log for a service. Users can easily enable logs for any service using the console. When you enable a service log, you must add it to a log group (see [Figure 8-7](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch08.xhtml#ch08fig07)); you can select the service and a resource provided by the service when enabling the service log. When the log is enabled, the log events pertaining to the resource are collected and indexed in the log from where it can be queried. Service logs can also be ingested into other services, where they can be combined with other operational data to provide more insight into your business and any applications running on OCI.



**Figure 8-7** Enabling Logging for a Service—in This Case, a Specific OCI Function

**Custom Logs**

The logs that are created by your applications can be ingested into the OCI Logging service. These are called *custom logs*. The main reason to ingest custom logs is to put your application logs in the same context as the infrastructure logs, to draw correlations between them and derive deeper insights. Unlike service logs that are enabled, custom logs are ingested into the OCI Logging service. The log resource is the same; however, the source behaves differently here. Service logs create high-fidelity log events with detailed contextual information. Custom logs, on the other hand, typically originate from a log file. OCI provides an agent to extract, parse, and ingest directly to the OCI Logging service. Typically, each line in a log file is considered to be a separate log event. Depending on the log format, the information in the log line can be tokenized or parsed to generate the fields of the log event to index it and make it searchable. Some logging formats, such as a web server, might be common, so parsers and tokenizers might exist for these; custom applications, on the other hand, might have a custom log format that requires a custom tokenizer. To set up custom logs, you create an agent configuration, as shown in [Figure 8-8](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch08.xhtml#ch08fig08).



**Figure 8-8** Agent Configuration for Custom Logs—the Image Shows the Log Inputs but Does Not Show the Parser Configuration (Which Can Be Selected for Each Log Input)

The agent configuration starts with determining the group of hosts from which to get the logs. This is typically a *dynamic group*. The dynamic groups are listed so that you can easily pick one. In the example shown in [Figure 8-8](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch08.xhtml#ch08fig08), the agent configuration is set up to ingest container logs from all containers running on a given OKE cluster. For this, the user first defines a dynamic group that contains all nodes that make up the cluster. It is not necessary to keep the granularity at the cluster level: The user could have created a dynamic group that picked a select subset of nodes in the cluster (say, GPU nodes used for a machine-learning workload) or all nodes across an availability domain or region, regardless of whether they were OKE nodes. The dynamic group has to be given access to the OCI Logging service so that the agents running on these hosts can push the log files to the service. Next, you configure the various log sources. These can be one or more paths where the log files of interest to you are located on the hosts. In the example, you see a single path for /var/log/containers/\*, which ingests all log files in this location. You can set up multiple paths to ingest from multiple sources simultaneously. For every input, you also set up the parser options to tokenize the log files to convert them into log events. More than 200 predefined parsers that cover popular log file formats such as Kubernetes Audit Logs, etcd logs, Apache logs, and syslog are supported, apart from generic formats such as JSON and XML. You can also create custom parsers used to tokenize your application’s log files that use a custom format for log data. Finally, you choose the destination log resource in the OCI Logging service to which the events from the log files should be ingested. After they are ingested, the log events are indexed and become searchable like service logs.

**Audit Logs**

OCI also tracks how and when each service is being accessed, and by whom. It captures metadata about the access or action performed to the resource, to provide a complete view into how your resources are being accessed and how. This is called the *audit log* and is separate from service logs. The audit service calls to all OCI API endpoints as log events. Because the service operates at the API level, log events are generated for all operations, regardless of whether they originate from within OCI or externally, and regardless of the clients (such as the CLI, the SDK, or custom applications) used to make the API calls.

The audit event is structured data and uses a well-defined schema called the audit schema to publish audit events. Using the standardized message structure provided by the schema makes it easy for consumers to listen to audit events, process them, and build automation code that can react to certain events. It is also worthwhile to note that OCI offers mechanisms to listen to events and take actions based on those using services, such as Cloud Guard, that do not involve custom code development. Event data in the audit schema is structured into a payload and an envelope. The payload is the data about the specific API call that is provided by the service that was the target of the API call. This can include the OCID for the resource that is targeted by the API call; information about the request, response, and identity of the caller; any resource state changes; and more. This payload is wrapped in an envelope that uses the CNCF CloudEvents[**3**](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch08.xhtml#ref8_3a) standard. [Listing 8-1](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch08.xhtml#list8_1) shows an example audit log event.

**Listing 8-1** Example Audit Log Event

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch08_images.xhtml#f0343-01a)

{

"datetime": 1667373839959,

"logContent": {

"dataschema": "2.0",

"id": "346d7f82-ea45-48fe-b7b4-a6cb3fb56dad",

"oracle": {

"enantednted": "ocid1.compartment.oc1..xxxx",

"ingestedtime": "2022-11-02T07:24:08.990Z",

"loggroupid": "\_Audit",

"enanted": "ocid1.tenancy.oc1..xxxx"

},

"source": "oke-k8sApiEndpoint-subnet-xxxx",

"specversion": "1.0",

"time": "2022-11-02T07:23:59.959Z",

"type": "com.oraclecloud.virtualNetwork.GetSubnet",

"data": {

"additionalDetails": {

"X-Real-Port": 59226

},

"availabilityDomain": "AD2",

"compartmentId": "xxxxxxxx",

"compartmentName": "AppDev",

"resourceId": "ocid1.subnet.oc1.iad.xxxx",

"definedTags": {},

"eventGroupingId": "xxxx/xxxx",

"eventName": "GetSubnet",

"freeformTags": {},

"identity": {

"authType": null,

"callerId": null,

"callerName": null,

"consoleSessionId": null,

"credentials": "xxxx,

"ipAddress": "10.240.2.7",

"principalId": "xxxx",

"principalName": "oke",

"tenantId": "xxxx",

"userAgent": "Oracle-JavaSDK/2.11.1 (Linux/4.14.35-2047;Java 64-Bit Server

VM GraalVM EE)"

},

"message": "oke-k8sApiEndpoint-subnet-xxxx GetSubnet succeeded",

"request": {

"action": "GET",

"headers": {},

"id": “xxxx/xxxx/xxxx",

"parameters": {},

"path": "/20160918/subnets/ocid1.subnet.oc1.iad.xxxx"

},

"response": {

"headers": {},

"message": null,

"payload": {},

"responseTime": "2022-11-02T07:23:59.959Z",

"status": “200”

},

"stateChange": {

"current": null,

"previous": null

}

}

}

}

The logcontent contains the overall envelope for the event. The data element within the envelope contains the resource-specific audit information. The envelope structure is the same for all events, regardless of the service; however, the payload contained in the data section is dependent on the type of the event.

**Auditing OKE Activity**

Audit logging captures activity within a service that can be used to create an audit trail of events and occurrences. It can be used to answer questions such as, “Who did what and when ?” Audit logging is an important part of several security and compliance programs because it maintains an immutable log of events and changes that occur within the services. In the case of cloud native execution environments such as OKE, these events can occur on two different levels:

* The infrastructure level, where some infrastructure configuration for your cluster is changing, such as the number of nodes in a node pool, the shape of the instances, or the version of Kubernetes being used. These changes are captured by the audit service along with metadata such as who made the change, when the change was made, whether the change was successful, and the previous and current states of the resources affected by the change.
* Within the execution environment itself. In the case of a Kubernetes cluster, these could be changes to the Kubernetes resources or actions and events that are performed on the cluster objects. For instance, if a user deploys a new pod onto the cluster, that event needs to be audited. In fact, for platforms such as OKE, these activities are more common and frequent than infrastructure events.

OKE captures changes to the Kubernetes objects in the audit log to provide a full audit trail of cluster events that occur within the cluster. These include all interactions with the Kubernetes API, including those made by nonhuman users such as service accounts. For example, consider an OKE cluster that deploys its workloads in a GitOps model using ArgoCD. ArgoCD runs as a workload on the cluster and periodically checks a Git repository for changes to the application deployment YAMLs. When a change is detected, ArgoCD deploys the change and updates the Kubernetes objects. This is an example of when there is no human interaction in the deployment process; the Kubernetes objects are updated by an automated system that identifies itself with a service account. [Listing 8-2](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch08.xhtml#list8_2) shows a PATCH event in which a change is deployed to the cluster.

**Listing 8-2** Example PATCH Event: Change Deployed to Cluster

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch08_images.xhtml#f0345-01a)

{

"datetime": 1667528786186,

"logContent": {

"data": {

"additionalDetails": null,

"availabilityDomain": null,

"compartmentId": "ocid1.compartment.oc1..

aaaaaaaagup7orev5wck2z3nh5hd6kgoaubksrundbrndocucev5dzct7wsq",

"compartmentName": "AppDev",

"definedTags": null,

"eventGroupingId": "c45ef9f5-5ec2-4dc8-a83b-2f9f2ba3ab64",

"eventName": "io.argoproj.v1alpha1.applications.patch",

"freeformTags": null,

"identity": {

"authType": "Native",

"callerId": null,

"callerName": null,

"consoleSessionId": null,

"credentials": "",

"ipAddress": null,

"principalId": null,

"principalName": "system:serviceaccount:argocd:argocd-application-

controller",

"tenantId": "",

"userAgent": "Go-http-client/2.0"

},

"message": "io.argoproj.v1alpha1.applications.patch succeeded",

"request": {

"action": "PATCH",

"headers": null,

"id": "c45ef9f5-5ec2-4dc8-a83b-2f9f2ba3ab64",

"parameters": null,

"path": "/apis/argoproj.io/v1alpha1/namespaces/argocd/applications/

wordpress"

},

"resourceId": "ocid1.cluster.oc1.iad.

aaaaaaaadcnof56t6ijrxbzjluujrjlwwvxz7u3guqxi7cyy5cemaalmxtlq",

"response": {

"headers": null,

"message": null,

"payload": null,

"responseTime": "2022-11-04T02:26:26.201Z",

"status": "200"

},

"stateChange": {

"current": {

"responseObject": null

},

"previous": {

"requestObject": null

}

}

},

"dataschema": "2.0",

"id": "c45ef9f5-5ec2-4dc8-a83b-2f9f2ba3ab64",

"oracle": {

"compartmentid": "ocid1.compartment.oc1..

aaaaaaaagup7orev5wck2z3nh5hd6kgoaubksrundbrndocucev5dzct7wsq",

"ingestedtime": "2022-11-04T02:26:35.914Z",

"loggroupid": "\_Audit",

"tenantid": ""

},

"source": "",

"specversion": "1.0",

"time": "2022-11-04T02:26:26.186Z",

"type": "io.argoproj.v1alpha1.applications.patch"

}

}

**Advanced Observability in OCI**

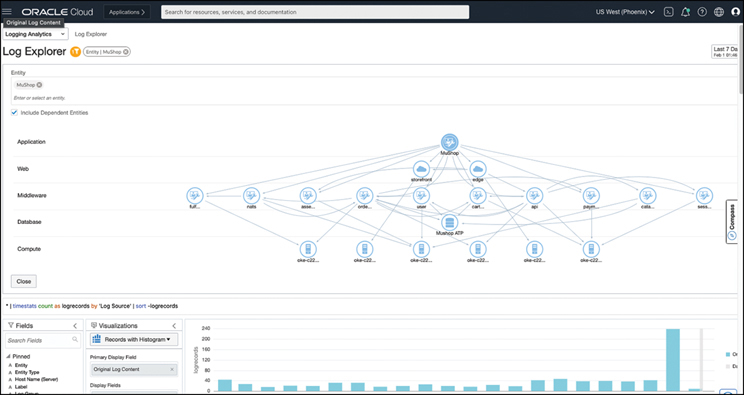
Advanced observability in OCI refers to a set of services that provide observability across several services and correlate information across structured and unstructured data. This includes, for example, the capability to derive insights by correlating data across logs and metrics from various services to form a more complete view of the systems and processes that are under observation. The advantage of using this set of advanced observability tools is that users can derive deep business insights and drive root cause analysis instead of focusing on surface-level problems. As an example, consider a scenario in which the metrics from an application indicate degraded performance for one of its service endpoints. The metrics from the application can point to the endpoint that is performing poorly and help you identify the issue. However, this issue could have been occurring in some other component, and that component might have been logging warnings about degraded performance for a while. In this instance, the capability to correlate the performance degradation on the service with the logs that described a potential warning on another service could have helped you to find the root cause for this issue and address it more quickly. In practice, however, the root cause for an issue could be a result of multiple separate systems interacting with each other in unexpected ways, with several smaller inefficiencies having a cumulative effect. The advanced tools in OCI are designed to observe a wide range of systems, from databases and file systems to Kubernetes clusters and application logs. This gives the advanced tools in OCI the capability to correlate metrics and log data across several layers in the stack, to create a complete 360-degree view of your workloads across applications, infrastructure, and external systems.

**Logging Analytics**

The biggest challenge with observability when operating at scale is the sheer amount of noise that it generates under normal operating conditions. Millions of events can be happening every second in a large enough set of distributed applications. Sifting through the noise to identify real events and signals and then correlate them to form insightful and actionable information is a real challenge. For instance, it would be trivial to monitor the network latencies and set up an alarm. This might be useful with a single application, but an application development team is less likely to be taking on infrastructure or network management responsibilities so that it can remediate a problem. An infrastructure or network team, on the other hand, would have multiple applications to service; when operating at scale, these alarms and events simply become noise. Another common example involves a security team. Security events can include file access, network connections, and process spawning, all of which can lead to a lot of noise in the raw data.

When discussing observability in this context, a common phrase used is “single pane of glass” for monitoring and observing systems and processes. The true intention of this phrase is to indicate visibility into all aspects of a system so that you can fully visualize and observe it from all angles and vantage points. In practice, however, this is harder to define (not implement) than it looks. This is because, as you saw in the earlier example, the view that is relevant to an operations and site reliability engineer might be very different from what is relevant to a developer or a penetration tester. Medium and large organizations have various personas, ranging from DevOps teams, to DBAs, to developers, to security teams. When you need observability in a “single pane of glass” across these teams, you need observability across your IT landscape, not just a few applications. It is also desirable to limit the number of tools used, to keep maintenance and dependencies in check. Log analytics is the general approach to solving this at scale. Industry solutions include Splunk and App Dynamics, among others.

OCI Logging Analytics is an OCI native service that provides these advanced analytic capabilities so that you can derive insights from across your entire IT landscape. OCI Logging Analytics can ingest data from various sources and correlate this to automatically generate complete application topologies and other visualizations. This makes it easy for every persona to have a relevant view, without having to invest time in capturing every metric and building every possible visualization. [Figure 8-9](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch08.xhtml#ch08fig09) shows an example in which OCI Logging Analytics has constructed a topology diagram for an application based on the log analysis.



**Figure 8-9** Logging Analytics Can Create Visual Representations of Applications and Systems, Along with How They Communicate, Based on Logs from the Various Systems—the Image Shows the Microservices That Make Up the MuShop Application Being Visualized, Based on the Logs Each Service Generates

These advanced analytical abilities can automatically learn patterns from the constant flow of metric and log data and can be used to identify unusual usage patterns that might indicate a security event or predict problems. These tools can be customized with additional log parsers as well.

**Enabling and Using Logging Analytics**

Logging Analytics works with OCI Logging and uses OCI IAM to control access, as with other services. The process of configuring the service includes setting up relevant IAM policies, followed by configuring how data is ingested into the Logging Analytics service. Log ingestion can ingest logs from both your tenancy and your on-premises resources. Additionally, the service can be configured to ingest audit logs from OCI, for added fidelity. Logs from compute instances and on-premises servers are collected by the compute management agent and sent over to the log group that is created for it. Although this host-level data is necessary, in the case of cloud native applications deployed to a Kubernetes cluster, you also need cluster-level metrics and logs to fully analyze the state of your workloads. To enable this, Logging Analytics uses fluentD to collect Kubernetes system/service logs, Linux system logs, and application pod/container logs. A preconfigured fluentD container can be deployed to your Kubernetes cluster as a DaemonSet, which continually collects data and ingests it into Log Analytics.

OCI Logging Analytics also includes built-in knowledge about well-known large-scale enterprise systems such as EBS. This is particularly useful when Oracle E-Business Suite (EBS) or a similar enterprise application suite is extended with custom bespoke applications that are now moving into a cloud native model. In these scenarios, Log Analytics can automatically discover your EBS or similar enterprise application deployment, including all its components and layers, and correlate it with systems that interact with it, such as a microservice running on a Kubernetes cluster.

**Prometheus and Grafana with OKE**

Prometheus and Grafana are some of the most widely used tools for monitoring metrics in Kubernetes. The kube-prometheus project provides a comprehensive experience for deploying Prometheus and Grafana on Kubernetes. It is based on the Prometheus Operator for Kubernetes, which uses the operator pattern to manage Prometheus deployments. The project provides pre-built Grafana dashboards and Prometheus rules, making it a complete monitoring solution for Kubernetes clusters. This project is an ideal starting point for most Kubernetes users. [Chapter 5](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch05.xhtml#ch05) includes the complete steps to deploy and manage this stack on OKE.

**Using the OCI DataSource Plug-ins for Grafana**

OCI provides data source plug-ins for Grafana that enable you to create panels and dashboards that query OCI directly. Data sources for both OCI Logging[**4**](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch08.xhtml#ref8_4a) and OCI Metrics[**5**](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch08.xhtml#ref8_5a) are available. The capabilities and query methods of each data source are different; in the case of the OCI plug-ins, you construct queries based on the region, compartment, metric namespace, and other OCI-specific dimensions. The OCI data source plug-in for Grafana has a customized Query Editor UI that is tuned for the features and capabilities that the plug-in exposes. In Grafana, you can combine data from multiple data sources onto a single dashboard, but each panel is tied to a specific data source. You can also create alerts based on OCI metrics using Alertmanager. You can download and install these plug-ins from the Grafana marketplace (see the “Resources” section at the end of the chapter).

**eBPF-Based Monitoring with Tetragon on OKE**

Security applications often have requirements and scenarios that are very broad in nature, such as being able to watch for all programs that are opening a sensitive file and perhaps even terminating programs that are exhibiting behaviors you are not expecting. The deepest you can get is down to the operating system kernel itself because very little activity in the OS escapes the kernel. Implementing these security protocols broadly across a system often leads to performance degradation of unacceptable levels because the tools typically run outside the kernel’s privileged execution context. This leads most users to settle for monitoring specific applications or smaller parts of a system. Building this functionality directly into the kernel or as kernel modules that execute within the kernel space often lets a program have very low overhead, but traditionally this comes at the cost of security and maintenance of these kernel modules. eBPF fundamentally changes this equation by providing a method of introducing new functionality that can execute in a sandboxed and privileged context without changing kernel source code or loading a kernel module. It essentially functions by creating a paradigm similar to a programming language virtual machine, such as Java. Modern Java programs are compiled into platform agnostic bytecode. This bytecode is consumed by the Java Virtual Machine (JVM), which uses a built-in Just-In-Time (JIT) compiler to convert the bytecode into native machine code to get native performance. Similarly, eBPF programs have a bytecode representation. BPF is deeply tied to the Linux kernel and can be considered a virtual machine inside the kernel. The in-kernel JIT compiler consumes an eBPF program in the eBPF bytecode and compiles it into native code that can execute in the kernel space.

eBPF uses an event-based model to load programs, and eBPF programs are written to “hook” into network events, systems calls, and more. When an event that an eBPF program hooks into is called, the eBPF program is loaded into the kernel after verification and JIT compilation. The verification step ensures that the program is safe to run, has the right privileges, and can run to completion; the JIT compilation ensures native performance. In many cases, eBPF programs are written in higher-level languages and compiled into the bytecode representation. These are then loaded into a running kernel after JIT compilation, based on the events that the programs are hooked into.

**Tetragon: eBPF-Based Security Observability and Enforcement**

Tetragon is a cloud native eBPF-based tool that performs security observability and enforcement. It is a component of the Cilium project. Using eBPF, Tetragon filters and observes events and applies policies in real time without sending events to an agent that is running outside the kernel. Tetragon can address numerous security and observability use cases by filtering for events such as a workload opening a network connection, accessing a file, or even starting a process inside a container. For instance, a shell process being started inside an application container could be considered a security event. Someone could be trying to troubleshoot an issue, or this could be some malicious activity—either way, it should trigger a security check to rule out an attack on the system. The same could be said about network connections being opened or files being read. Tetragon can trace and filter these activities while introducing little to no overhead, usually at the earliest stage that these events can be detected in software.

Tetragon is ideally suited for all Kubernetes workloads, and it runs as a DaemonSet in each node on the cluster. Tetragon can then pull metadata from the Kubernetes API server and correlate that metadata with the events observed within the kernel of each node. Tetragon makes it easy to set up real-time filters for these activities and more using TracingPolicies. A TracingPolicy is a custom resource created by Tetragon that enables admins and DevSecOps teams to create and deploy filters for kernel events as Kubernetes resources. A TracingPolicy can match system calls, process attributes and arguments, and also trigger an action on matches.

**Running Tetragon on Oracle Container Engine for Kubernetes (OKE)**

Tetragon can be deployed to Kubernetes clusters on OKE using the Helm chart published by the Tetragon project. After it is installed, the TracingPolicy Custom Resource Definition (CRD) is created and Tetragon runs on the cluster nodes as a DaemonSet.

**Prerequisites for Oracle Linux**

OKE uses Oracle Linux, and Tetragon relies on having the BPF Type Format (BTF) support in the kernel. Recent Oracle Linux kernels include this out of the box. For this reason, users should use a kernel that is 5.4.17-2136.305.3.el7uek or newer. Tetragon also does not provide support for ARM (linux/arm64) architecture; at the time of writing, it provides only x86 (linux/amd64) support. If you have ARM nodes in your OKE cluster, the DaemonSet will stay in the Init:CrashLoopBackOff status.

Recent versions of the OKE node images are based on kernels that include BTF support. This caveat for BTF support is applicable only for clusters in which the node OS has not been updated in a while, not for newly created clusters. If you are unsure, the best way to check whether you have BTF support is to log in to the node using SSH and run ls /sys/kernel/btf. You should see the kernel (vmlinux) and modules listed here.

To check the version of the kernel that your nodes are using, run uname -a on the node. If you are running an older version of the kernel, you can upgrade the version on the node pool configuration. However, this affects only newly created nodes; existing nodes are not upgraded automatically to ensure continuity for the workloads that might be running on them. You can follow the node pool upgrade process to bring your existing nodes up to the newer kernel versions.

When you have ensured that you are running on a recent version of the kernel on your nodes, you can get started with Tetragon installation using the Tetragon Helm chart. You can follow the instructions from the Tetragon GitHub page as well. To use the Helm chart Tetragon, follow these instructions:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch08_images.xhtml#f0352-01a)

helm repo add cilium https://helm.cilium.io

helm repo update

helm install tetragon cilium/tetragon -n kube-system

kubectl rollout status -n kube-system ds/tetragon -w

If you see that Tetragon pods are in a CrashLoopBackOff state, this could be caused by one of two reasons. The most likely reason is that this is occurring on ARM-based nodes, if you have them in your cluster. Tetragon does not yet run on ARM as of the time of writing. To confirm that this is the case, use the following line:

kubectl describe pod

You will then see the init container named tetragon-operator. This is likely failing and in a terminated state, with an exit code of 1. You can use the following line to view the init container logs:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch08_images.xhtml#f0352-02a)

kubectl logs <pod\_name> -c tetragon-operator -n kube-system

You might see the reason for the init container to terminate as standard\_init\_linux.go:228: exec user process caused: exec format error, indicating that the binary is not meant for use on ARM CPU architecture.

The second reason Tetragon pods are in a CrashLoopBackOff state could be that you have an older kernel on your node, and BTF support is not included in it. To verify this, get the container logs for the failing container in the pod, as described previously. If the lack of BTF support in the kernel is the issue, you will see an error message similar to the following:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch08_images.xhtml#f0352-03a)

aborting kernel autodiscovery failed: Kernel version BTF search failed kernel is

not included in supported list.

Use --btf option to specify BTF path and/or '--kernel' to specify kernel version

This is expected on nodes that have not had their OS updated for a while. To resolve this, the node pool configuration needs to be updated and the nodes need to be upgraded, following the standard node pool upgrade process.

When the DaemonSet is ready and the Tetragon pods are in the Running state, you can start listening to events on your nodes. Out of the box, Tetragon can monitor process execution. Tetragon emits the events it matches in JSON format, and the logs can be observed with the following command (assuming that you have jq installed):

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch08_images.xhtml#f0353-01a)

kubectl logs -n kube-system -l app.kubernetes.io/name=tetragon -c export-stdout

-f | jq

Depending on what activity is occurring on your cluster, you will see a stream of JSON objects that represent these events. [Listing 8-4](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch08.xhtml#list8_4) shows sample output from a cluster that was running ArgoCD, where it was cloning a Git repository.

**Listing 8-4** Example Logs from Tetragon Showing Activities That It Is Tracking

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch08_images.xhtml#f0353-02a)

{

"process\_exec": {

"process": {

"exec\_id": "MTAuMC4xMC4yMTg6OTE0MTQ2NjAzODU0MDcwOjEwNDA4Ng==",

"pid": 104086,

"uid": 999,

"cwd": "/tmp/\_argocd-repo/83c509d8-f9ba-48c3-a217-a9278134963e/",

"binary": "/usr/bin/git",

"arguments": "rev-parse HEAD",

"flags": "execve clone",

"start\_time": "2022-06-07T17:03:42.519Z",

"auid": 4294967295,

"pod": {

"namespace": "argocd",

"name": "argocd-repo-server-7db4cc4b45-cpvlt",

"container": {

"id": "cri-o://1c361244fcb1d89c02ef297e69a13bd80fd4d575ae965a92979deec74

0711e17",

"name": "argocd-repo-server",

"image": {

"id": "quay.io/argoproj/argocd@sha256:85d55980e70f8f7073e4ce529a7bbcf6

d55e51f8a7fc4b45d698f0a7ffef0fea",

"name": "quay.io/argoproj/argocd:v2.3.4"

},

"start\_time": "2022-05-31T16:57:53Z",

"pid": 319

}

},

"docker": "1c361244fcb1d89c02ef297e69a13bd",

"parent\_exec\_id": "MTAuMC4xMC4yMTg6MzA4OTk3NTAyODQyMTEzOjExMjQ3",

"refcnt": 1

}

},

"node\_name": "10.0.10.218",

"time": "2022-06-07T17:03:42.519Z"

}

The event stream as JSON output is verbose and hard to understand, but it is information dense. You have several ways of ingesting this JSON data and deriving analytical information from it. The obvious one is to use the Tetragon CLI tool. Isovalent, the company behind Cilium and Tetragon, also offers a full-featured commercial product that can analyze and visualize this data, to make it more actionable and easier to assimilate.

**Installing the Tetragon CLI**

The Tetragon CLI is useful to filter events by pod, host, namespace, or process. The CLI can be downloaded from the GitHub releases page. Simply download the tool based on your operating system and CPU architecture, and untar it to a standard location such as /usr/local/bin, or add the path to the binary to your PATH variable for your shell. Alternatively, if you have go installed on your workstation where you want to run the CLI, you can download and install it with the commands in [Listing 8-5](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch08.xhtml#list8_5).

**Listing 8-5** Installing the Tetragon CLI

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch08_images.xhtml#f0354-01a)

GOOS=$(go env GOOS)

GOARCH=$(go env GOARCH)

curl -L --remote-name-all https://github.com/cilium/tetragon/releases/download/

tetragon-cli/tetragon-${GOOS}-${GOARCH}.tar.gz{,.sha256sum}

sha256sum --check tetragon-${GOOS}-${GOARCH}.tar.gz.sha256sum

sudo tar -C /usr/local/bin -xzvf tetragon-${GOOS}-${GOARCH}.tar.gz

rm tetragon-${GOOS}-${GOARCH}.tar.gz{,.sha256sum}

With the Tetragon CLI installed, the events from the log files can be pretty printed simply by sending the JSON output to the CLI command tetragon observe, as shown here:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch08_images.xhtml#f0354-02a)

kubectl logs -n kube-system ds/tetragon -c export-stdout -f | tetragon observe

**TracingPolicies for FileAccess and Network Observability**

TracingPolicies are custom resources that make it easy to set up real-time filters for kernel events. A TracingPolicy can not only match and filter system calls for observability, it can also trigger an action on these matches. Tetragon offers a few examples that showcase this capability, to inspire your own TracingPolicies.

Apply the example tracing policies for file access and network observability, as shown here:

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch08_images.xhtml#f0354-03a)

kubectl apply -f https://raw.githubusercontent.com/cilium/tetragon/v0.8.0//crds/

examples/sys\_write\_follow\_fd\_prefix.yaml

kubectl apply -f https://raw.githubusercontent.com/cilium/tetragon/v0.8.0/crds/

examples/tcp-connect.yaml

With these additional TracingPolicies enabled, Tetragon starts tracing file access and network activity, as demonstrated in [Listing 8-6](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch08.xhtml#list8_6).

**Listing 8-6** An Example Showing the Tetragon CLI Displaying Events Such as File Access and Network Calls

[Click here to view code image](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch08_images.xhtml#f0355-01a)

$ kubectl logs -n kube-system ds/tetragon -c export-stdout -f | tetragon observe

...[output truncated]

Images process default/xwing /bin/bash

Images open default/xwing /bin/bash /etc/passwd

Images close default/xwing /bin/bash

Images open default/xwing /bin/bash /etc/terminfo/x/xterm

Images close default/xwing /bin/bash

Images process default/xwing /bin/cat /etc/passwd

Images open default/xwing /bin/cat /etc/passwd

Images close default/xwing /bin/cat

Images exit default/xwing /bin/cat /etc/passwd 0

Images process default/xwing /usr/bin/curl -Lv https://cloud.oracle.com

Images open default/xwing /usr/bin/curl /etc/ssl/openssl.cnf

Images close default/xwing /usr/bin/curl

Images open default/xwing /usr/bin/curl /etc/hosts

Images close default/xwing /usr/bin/curl

Images open default/xwing /usr/bin/curl /etc/resolv.conf

Images close default/xwing /usr/bin/curl

Images connect default/xwing /usr/bin/curl tcp 10.244.1.152:65175 -> 23.212.250.69:443

Images open default/xwing /usr/bin/curl /etc/ssl/certs/ca-certificates.crt

Images close default/xwing /usr/bin/curl

Images sendmsg default/xwing /usr/bin/curl tcp 10.244.1.152:65175 -> 23.212.250.69:443

bytes 517

Images sendmsg default/xwing /usr/bin/curl tcp 10.244.1.152:65175 -> 23.212.250.69:443

bytes 126

Images sendmsg default/xwing /usr/bin/curl tcp 10.244.1.152:65175 -> 23.212.250.69:443

bytes 109

Images sendmsg default/xwing /usr/bin/curl tcp 10.244.1.152:65175 -> 23.212.250.69:443

bytes 31

Images close default/xwing /usr/bin/curl tcp 10.244.1.152:65175 -> 23.212.250.69:443

Images exit default/xwing /usr/bin/curl -Lv https://cloud.oracle.com 0

Images exit default/xwing /bin/bash 0

...[output truncated]

These events are monitored directly from within the kernel, so very little can be obfuscated or masked by a malicious actor.

The primary downside to this approach is that actions you can take (such as killing a process that reads a file) are reactionary: You know about the event as it is happening, not beforehand. Still, it is extremely powerful to be able to have a low-overhead solution for filtering and matching events at the kernel level and being able to create policies that can help you observe and act on them.

**Summary**

This chapter covered several services and choices that enable you to observe both your infrastructure and your applications in OCI. OCI services always emit metrics and logs that can be captured and analyzed. The services that enable this functionality can also do it for your applications. OCI Monitoring can capture, store, and search through the metric data that OCI services and your applications generate. Monitoring is also integrated with alarms that can trigger other systems by creating notifications. OCI Logging can capture, index, and search through logs generated by OCI services and your application. This can help you understand patterns in application use and behavior. OCI Logging also includes audit logging and support for custom logging for your applications. Apart from these services, the OCI Logging Analytics service has features that can ingest metrics and logs from a multitude of services and provide deep insights into your applications and infrastructure, with minimal configuration. You can also bring in your own tools and processes to monitor your OCI resources and assets. For most cloud native applications running on Kubernetes, the Prometheus and Grafana stack is a popular choice for gathering and visualizing metrics. This chapter included an overview of setting up this popular set of tools on OKE, including how to set up OCI plug-ins for Grafana. eBPF is an emerging technology that is often used for observability, and this chapter also covered its installation and use on OCI with the popular open-source project Tetragon.

**References**

[1](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch08.xhtml#ref8_1) Supported monitoring services: <https://docs.oracle.com/en-us/iaas/Content/Monitoring/Concepts/monitoringoverview.htm#SupportedServices>

[2](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch08.xhtml#ref8_2) SDK example for posting metric data for your application to OCI Monitoring: <https://github.com/oracle/oci-java-sdk/blob/master/bmc-examples/src/main/java/MonitoringMetricPostExample.java>

[3](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch08.xhtml#ref8_3) CloudEvents: <https://cloudevents.io/>

[4](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch08.xhtml#ref8_4) OCI Logs data source for Grafana: <https://grafana.com/grafana/plugins/oci-logs-datasource/>

[5](https://learning.oreilly.com/library/view/oracle-cloud-infrastructure/9780137902835/ch08.xhtml#ref8_5) OCI Metrics data source for Grafana: <https://grafana.com/grafana/plugins/oci-metrics-datasource/>