Getting Started with Falco and Cloud-Native Distributed SQL on Google Kubernetes Engine

Falco is an incubating CNCF project that provides cloud-native, open source runtime security for applications running in Kubernetes environments. Falco monitors process behaviors to detect anomalous activity and help administrators gain deeper insights into process execution. Behind the scenes, Falco leverages the Linux-native extended Berkeley Packet Filter (eBPF) technology to analyze network traffic and audits a system at the most fundamental level, the Linux kernel. Flaco then enriches this data with other input streams, including container and Kubernetes metrics, to provide even deeper insights.

Because YugabyteDB is a cloud-native, distributed SQL database that is designed to run in Kubernetes environments, it can interoperate with Falco and many other CNCF projects right out-of-the-box.

What's YugabyteDB? It is an open source, high-performance distributed SQL database built on a scalable and fault-tolerant design inspired by Google Spanner. Yugabyte's SQL API (YSQL) is PostgreSQL wire compatible.

Why Falco and YugabyteDB?

When it comes to running YugabyteDB in a Kubernetes environment, implementing the recommended database security controls are a great start. However, it is also important to go one level deeper and put in place controls and monitoring to detect unexpected behavior that could be malicious.

For example, a malicious company insider can often find ways to eavesdrop on sensitive data like credit card information, social security numbers, or health records by connecting on different ports or reading the sensitive data files directly from disk or by copying backup files, completely bypassing the database engine. To safeguard against these types of vulnerabilities it makes sense to deploy multiple security layers. By using Falco with a YugabyteDB deployment, it can help create this additional layer of defense in Kubernetes environments. In this blog post we'll show you how to get up and running with YugabyteDB and Falco on Google Cloud Platform plus implement and test some basic security policies.

Prerequisites

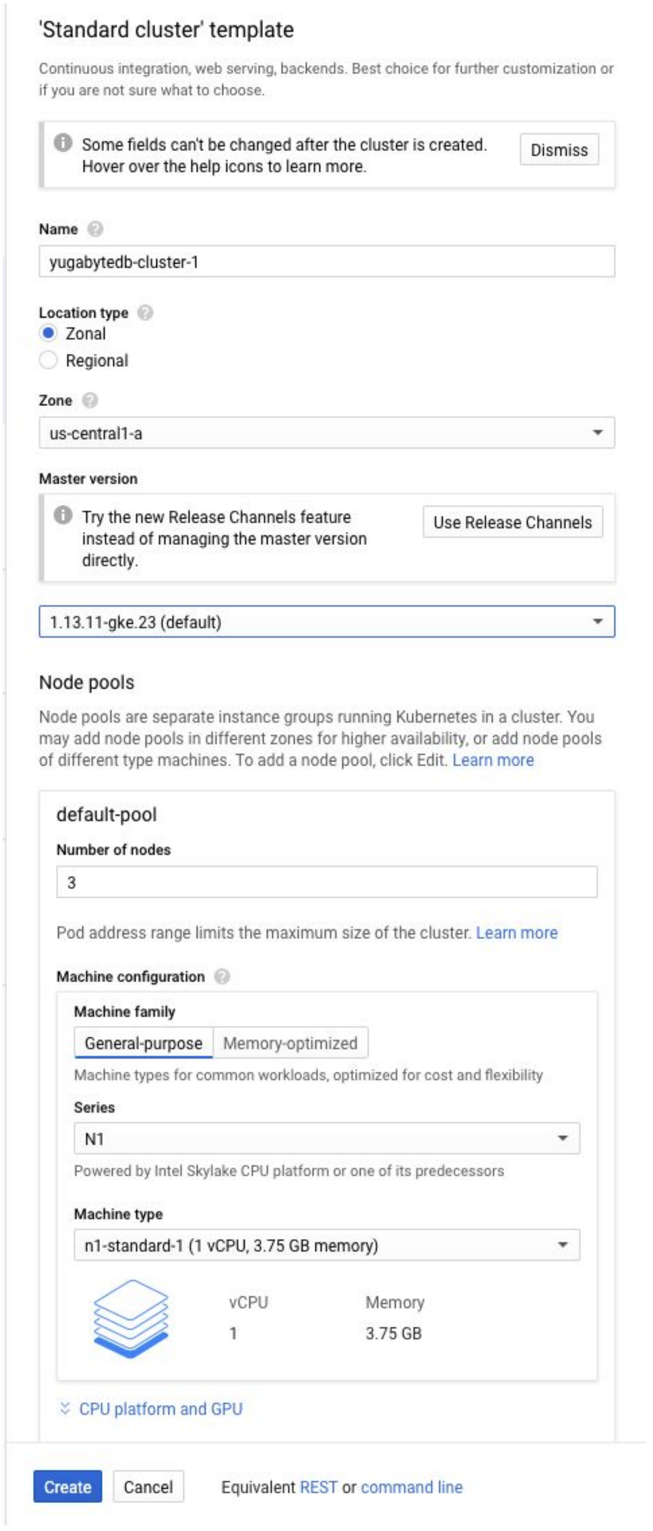
Below is the environment which we'll use to run a YugabyteDB cluster on top of a Google Kubernetes cluster integrated with Falco.

1. YugabyteDB (using Helm Charts) - Version 2.0.11
2. Falco (using Helm Charts) - Version 0.19.0
3. A Google Cloud Platform account

Setting Up a Kubernetes Cluster on Google Cloud Platform

To deploy YugabyteDB on the Google Cloud Platform (GCP), we first have to set up a cluster. To create the cluster in Google Kubernetes Engine (GKE):

Go to *Kubernetes Engine> Clusters > Create Cluster > Standard cluster.* For the purposes of this example we'll name the cluster *yugabytedb-cluster-1* and use the default options.


'Standard cluster' template

Continuous integration, web serving, backends. Best choice for further customization or if you are not sure what to choose.

Some fields can't be changed after the cluster is created. Hover over the help icons to learn more. Dismiss

Name

yugabytedb-cluster-1

Location type

● Zonal

○ Regional

Zone

us-central1-a

Master version

Try the new Release Channels feature instead of managing the master version directly. Use Release Channels

1.13.11-gke.23 (default)

Node pools

Node pools are separate instance groups running Kubernetes in a cluster. You may add node pools in different zones for higher availability, or add node pools of different type machines. To add a node pool, click Edit. Learn more

default-pool

Number of nodes

3

Pod address range limits the maximum size of the cluster. Learn more

Machine configuration

Machine family

General-purpose | Memory-optimized

Machine types for common workloads, optimized for cost and flexibility

Series

N1

Powered by Intel Skylake CPU platform or one of its predecessors

Machine type

n1-standard-1 (1 vCPU, 3.75 GB memory)

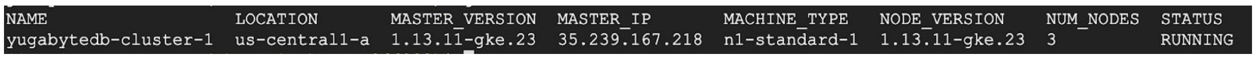
vCPU	Memory
1	3.75 GB

CPU platform and GPU

Create | Cancel | Equivalent REST or command line

Connect to the ***Google Cloud Shell*** and verify that the nodes are setup and running by using the command:

```
$ gcloud container clusters list
```



```
NAME                  LOCATION       MASTER_VERSION  MASTER_IP       MACHINE_TYPE   NODE_VERSION    NUM_NODES  STATUS
yugabytedb-cluster-1  us-central1-a  1.13.11-gke.23  35.239.167.218  n1-standard-1  1.13.11-gke.23  3          RUNNING
```

Note that in this case, we have configured the cluster with *n1-standard-1* machine type.

Installing YugabyteDB

We will be using Helm charts to install YugabyteDB and Falco. However, before we dive into the steps to install YugabyteDB, let's make sure that the Helm prerequisites are available.

Verify and upgrade Helm

First, check to see if Helm is installed by using the Helm version command:

```
$ helm version

Client: &version.Version{SemVer:"v2.14.1",
GitCommit:"5270352a09c7e8b6e8c9593002a73535276507c0",
GitTreeState:"clean"}
Error: could not find tiller
```

If you run into issues associated with Tiller, such as the error above, you can initialize Helm with the upgrade option:

```
$ helm init --upgrade --wait

$HELM_HOME has been configured at /home/jimmy/.helm.

Tiller (the Helm server-side component) has been installed into your
Kubernetes Cluster.

Please note: by default, Tiller is deployed with an insecure 'allow
unauthenticated users' policy.

To prevent this, run `helm init` with the --tiller-tls-verify flag.

For more information on securing your installation see:
https://docs.helm.sh/using_helm/#securing-your-helm-installation
```

You should now be able to install YugabyteDB using a Helm chart.

Create a service account

Before you can create the cluster, you need to have a service account that has been granted the *cluster-admin* role. Use the following command to create a *yugabyte-helm* service account granted with the ClusterRole of *cluster-admin*.

```
kubectl create -f
https://raw.githubusercontent.com/yugabyte/charts/master/stable/yugab
yte/yugabyte-rbac.yaml

serviceaccount/yugabyte-helm created
clusterrolebinding.rbac.authorization.k8s.io/yugabyte-helm created
```

Initialize Helm

```
$ helm init --service-account yugabyte-helm --upgrade --wait

$HELM_HOME has been configured at /home/jimmy/.helm.
Tiller (the Helm server-side component) has been installed into your
Kubernetes Cluster.

Please note: by default, Tiller is deployed with an insecure 'allow
unauthenticated users' policy.
To prevent this, run `helm init` with the --tiller-tls-verify flag.
For more information on securing your installation see:
https://docs.helm.sh/using_helm/#securing-your-helm-installation
```

Create a namespace

```
$ kubectl create namespace yb-demo

namespace/yb-demo created
```

Add the charts repository

```
$ helm repo add yugabytedb https://charts.yugabyte.com

"yugabytedb" has been added to your repositories
```

Fetch updates from the repository

```
$ helm repo update

Hang tight while we grab the latest from your chart repositories...
...Skip local chart repository
...Successfully got an update from the "yugabytedb" chart repository
...Successfully got an update from the "stable" chart repository
Update Complete.
```

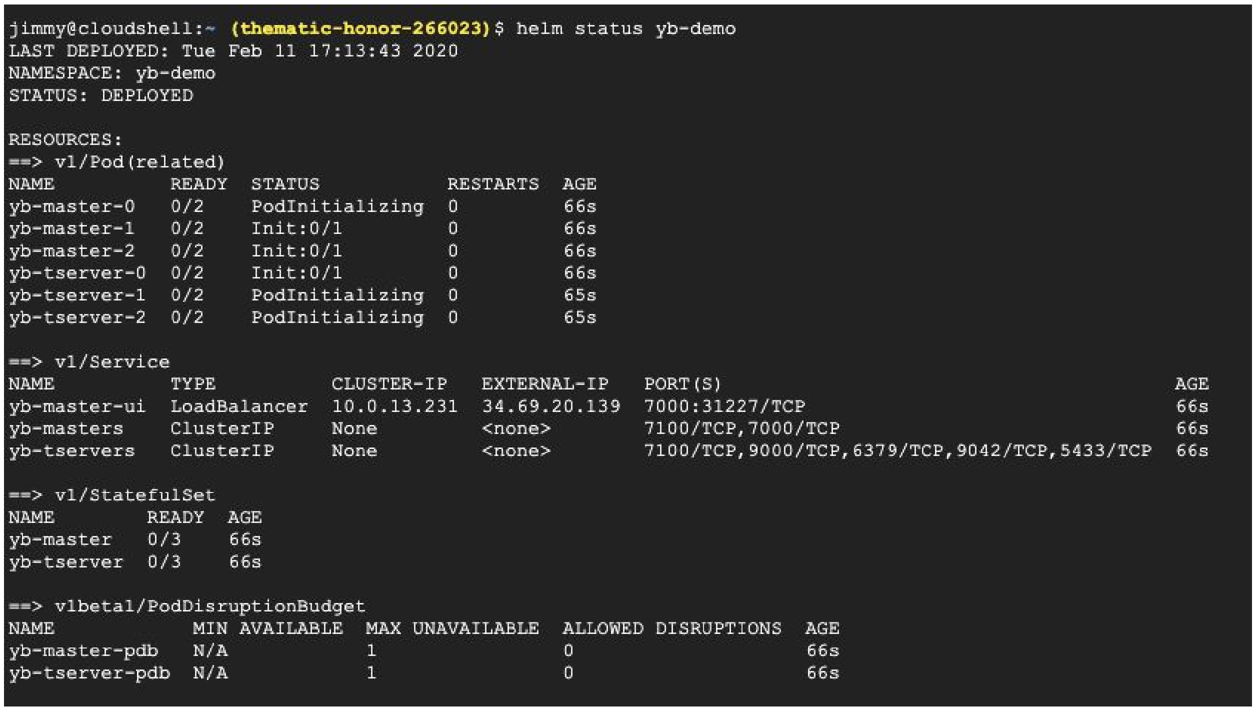
Install YugabyteDB

By default, the YugabyteDB Helm chart will expose only the master UI endpoint using the LoadBalancer. For the purposes of this blog post, we also want to expose the YSQL service. Additionally, since we used *n1-standard-1* type servers in our cluster, we will use the Helm resource options for low resource environments.

```
helm install yugabytedb/yugabyte --set
resource.master.requests.cpu=0.1,resource.master.requests.memory=0.2G
i,resource.tserver.requests.cpu=0.1,resource.tserver.requests.memory=
0.2Gi --namespace yb-demo --name yb-demo
```

To check the status of the YugabyteDB cluster, execute the command below:

```
helm status yb-demo
```


```
jimmy@cloudshell:~ (thematic-honor-266023)$ helm status yb-demo
LAST DEPLOYED: Tue Feb 11 17:13:43 2020
NAMESPACE: yb-demo
STATUS: DEPLOYED

RESOURCES:
==> v1/Pod(related)
NAME           READY  STATUS           RESTARTS  AGE
yb-master-0    0/2    PodInitializing  0         66s
yb-master-1    0/2    Init:0/1         0         66s
yb-master-2    0/2    Init:0/1         0         66s
yb-tserver-0   0/2    Init:0/1         0         66s
yb-tserver-1   0/2    PodInitializing  0         65s
yb-tserver-2   0/2    PodInitializing  0         65s

==> v1/Service
NAME           TYPE          CLUSTER-IP   EXTERNAL-IP   PORT(S)                                       AGE
yb-master-ui   LoadBalancer  10.0.13.231  34.69.20.139  7000:31227/TCP                                66s
yb-masters     ClusterIP     None         <none>        7100/TCP,7000/TCP                             66s
yb-tservers    ClusterIP     None         <none>        7100/TCP,9000/TCP,6379/TCP,9042/TCP,5433/TCP  66s

==> v1/StatefulSet
NAME        READY  AGE
yb-master   0/3    66s
yb-tserver  0/3    66s

==> v1beta1/PodDisruptionBudget
NAME            MIN AVAILABLE  MAX UNAVAILABLE  ALLOWED DISRUPTIONS  AGE
yb-master-pdb   N/A            1                0                    66s
yb-tserver-pdb  N/A            1                0                    66s
```

Congrats! At this point you have a three node YugabyteDB cluster running on GKE.

Installing Falco

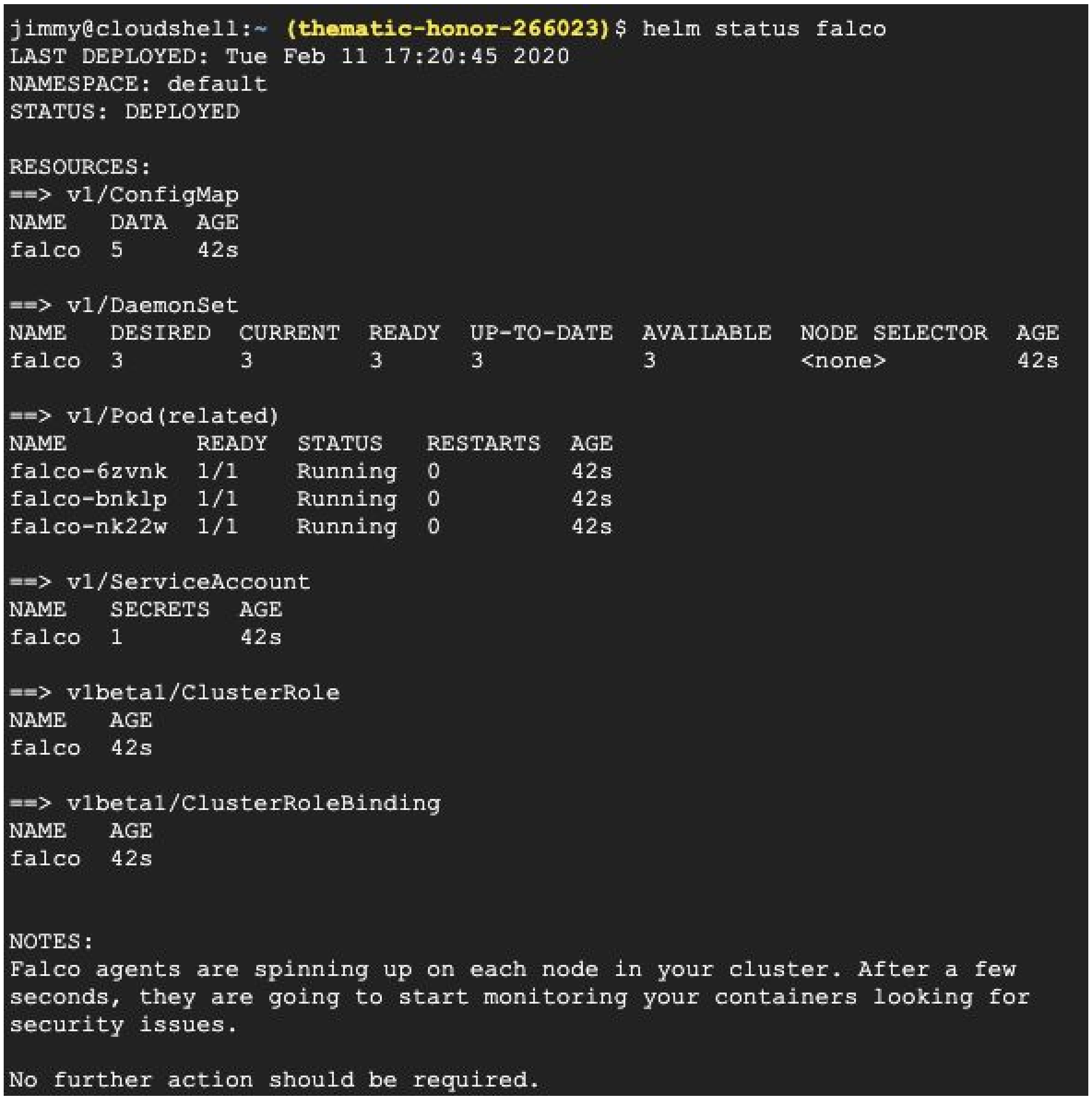
Now, let's proceed with installing Falco. For the purposes of this blog, we'll be using Helm charts version 1.1.0.

As previously mentioned, Falco requires eBPF, and by default eBPF is not enabled in GKE's Cloud OS. The Helm install command that enables eBPF and installs Falco chart version 1.1.0 can be found below:

```
$ helm install --name falco --set ebpf.enabled=true stable/falco
--version=1.1.0
```

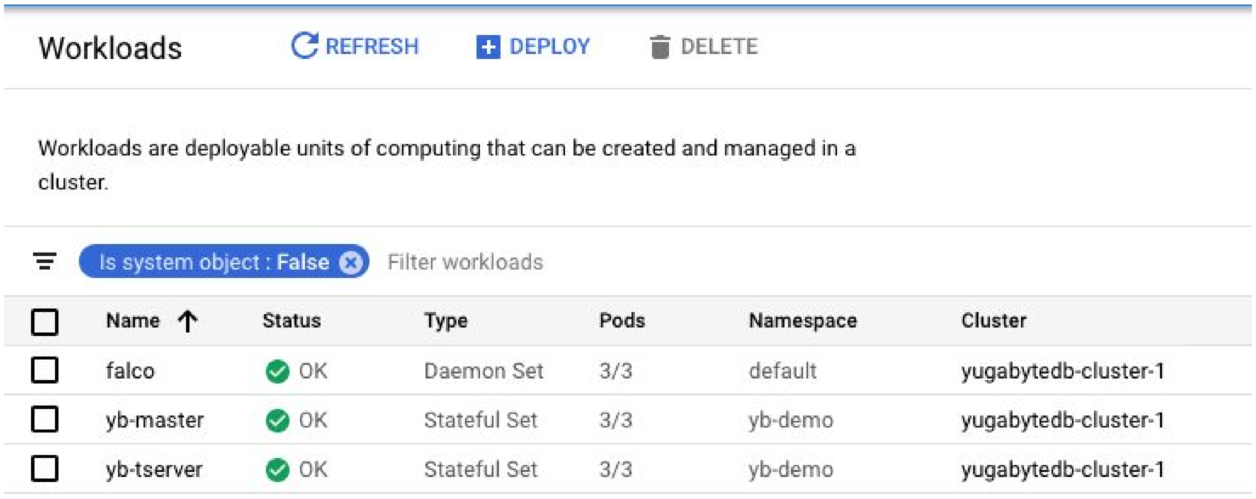
We can now check the status of the Falco pod by executing:

```
$ helm status falco
```



```
jimmy@cloudshell:~ (thematic-honor-266023)$ helm status falco
LAST DEPLOYED: Tue Feb 11 17:20:45 2020
NAMESPACE: default
STATUS: DEPLOYED

RESOURCES:
==> v1/ConfigMap
NAME   DATA  AGE
falco  5     42s

==> v1/DaemonSet
NAME   DESIRED  CURRENT  READY  UP-TO-DATE  AVAILABLE  NODE SELECTOR  AGE
falco  3        3        3      3           3          <none>         42s

==> v1/Pod(related)
NAME         READY  STATUS   RESTARTS  AGE
falco-6zvnk  1/1    Running  0         42s
falco-bnklp  1/1    Running  0         42s
falco-nk22w  1/1    Running  0         42s

==> v1/ServiceAccount
NAME   SECRETS  AGE
falco  1        42s

==> v1beta1/ClusterRole
NAME   AGE
falco  42s

==> v1beta1/ClusterRoleBinding
NAME   AGE
falco  42s


NOTES:
Falco agents are spinning up on each node in your cluster. After a few
seconds, they are going to start monitoring your containers looking for
security issues.

No further action should be required.
```

Alternatively, the “Workloads” tab in GKE should also indicate that Falco is running.



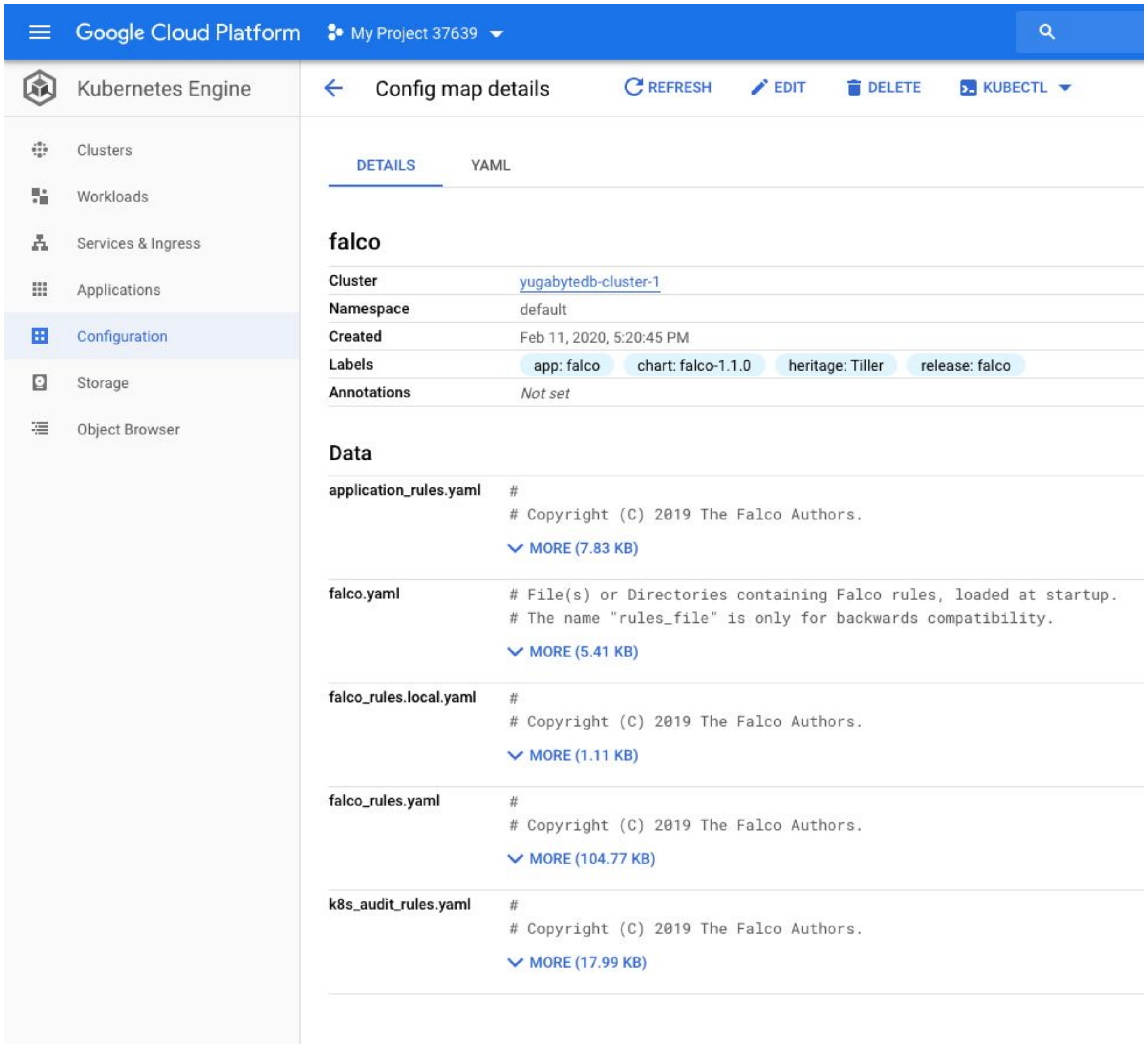
Workloads are deployable units of computing that can be created and managed in a cluster.

Name	Status	Type	Pods	Namespace	Cluster
falco	OK	Daemon Set	3/3	default	yugabytedb-cluster-1
yb-master	OK	Stateful Set	3/3	yb-demo	yugabytedb-cluster-1
yb-tserver	OK	Stateful Set	3/3	yb-demo	yugabytedb-cluster-1

Falco Rules and Configmap

Out of the box, Falco comes with a rich set of predefined rules that you can edit for flagging abnormal behaviors. The rules are essentially yaml files (**_rules.yaml*) that contain the checks that Falco uses to generate alerts (shells being opened, files being modified, incoming connections, etc.). In addition to the rules, there are also configuration files (such as *falco.yaml*) that have deamon settings such as output type, ports, etc.

In Kubernetes, *ConfigMaps* allow you to decouple configuration artifacts from image content to keep containerized applications portable. You can view the rules and configuration files of Falco through the “Configuration” tab in GKE, and then by selecting the Falco *ConfigMap*. This can be very useful when writing custom rules for Falco.



falco

Cluster	yugabytedb-cluster-1
Namespace	default
Created	Feb 11, 2020, 5:20:45 PM
Labels	app: falco, chart: falco-1.1.0, heritage: Tiller, release: falco
Annotations	*Not set*

Data

application_rules.yaml
```
#
# Copyright (C) 2019 The Falco Authors.
```
MORE (7.83 KB)

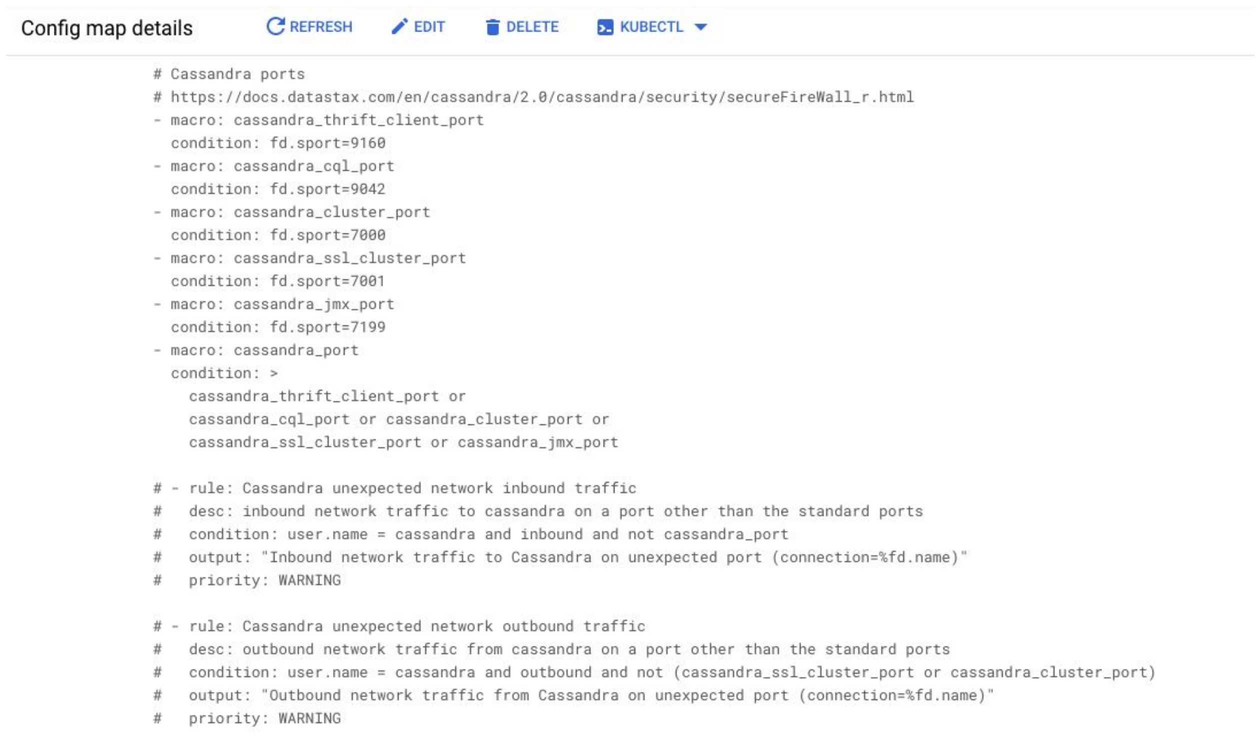
falco.yaml
```
# File(s) or Directories containing Falco rules, loaded at startup.
# The name "rules_file" is only for backwards compatibility.
```
MORE (5.41 KB)

falco_rules.local.yaml
```
#
# Copyright (C) 2019 The Falco Authors.
```
MORE (1.11 KB)

falco_rules.yaml
```
#
# Copyright (C) 2019 The Falco Authors.
```
MORE (104.77 KB)

k8s_audit_rules.yaml
```
#
# Copyright (C) 2019 The Falco Authors.
```
MORE (17.99 KB)

For example, Falco has preconfigured rules for databases like MongoDB, Cassandra, and Elasticsearch. These rules are set up to catch things like inbound/outbound network traffic on a port other than the standard ports. Similar rules could be configured to monitor inbound and outbound YugabyteDB traffic on unauthorized ports.



```
# Cassandra ports
# https://docs.datastax.com/en/cassandra/2.0/cassandra/security/secureFireWall_r.html
- macro: cassandra_thrift_client_port
  condition: fd.sport=9160
- macro: cassandra_cql_port
  condition: fd.sport=9042
- macro: cassandra_cluster_port
  condition: fd.sport=7000
- macro: cassandra_ssl_cluster_port
  condition: fd.sport=7001
- macro: cassandra_jmx_port
  condition: fd.sport=7199
- macro: cassandra_port
  condition: >
    cassandra_thrift_client_port or
    cassandra_cql_port or cassandra_cluster_port or
    cassandra_ssl_cluster_port or cassandra_jmx_port

# - rule: Cassandra unexpected network inbound traffic
#   desc: inbound network traffic to cassandra on a port other than the standard ports
#   condition: user.name = cassandra and inbound and not cassandra_port
#   output: "Inbound network traffic to Cassandra on unexpected port (connection=%fd.name)"
#   priority: WARNING

# - rule: Cassandra unexpected network outbound traffic
#   desc: outbound network traffic from cassandra on a port other than the standard ports
#   condition: user.name = cassandra and outbound and not (cassandra_ssl_cluster_port or cassandra_cluster_port)
#   output: "Outbound network traffic from Cassandra on unexpected port (connection=%fd.name)"
#   priority: WARNING
```

Creating Custom YugabyteDB Rules in Falco

Let's go ahead and create a test rule to audit logins whenever a user named *user1* logs into YugabyteDB.

Click on the *edit* button of the *applications_rules.yaml* file and make the following additions to the yaml file.

```
# YugabyteDB logins
     - rule: Audit logins from user user1
       desc: Audit logins from user user1
       condition: user.name = user1 and inbound and fd.sport = 5433
       output: "Login from user1 to YugabyteDB (connection=%fd.name)"
       priority: WARNING
```

Connecting to Yugabyte DB with a Test Login

Finally, let's do a "quick and dirty" test of our auditing rule by connecting to YugabyteDB, creating the *user1* login and then logging in with that new role.

```
$ kubectl exec -n yb-demo -it yb-tserver-0 /home/yugabyte/bin/ysqlsh
-- -h yb-tserver-0.yb-tservers.yb-demo
```

```
yugabyte=# CREATE ROLE user1 WITH LOGIN SUPERUSER;

\q

$ kubectl exec -n yb-demo -it yb-tserver-0 bash

$ ./bin/ysqlsh -U user1 -h yb-tserver-0 -d yugabyte

ysqlsh (11.2-YB-2.0.11.0-b0)
Type "help" for help.
yugabyte=#
```

Inspecting Falco audit logs

The login of *user1* should have been logged by Falco based on the rule we configured earlier. To view the audit trail we can inspect the logs.

```
$ kubectl logs --selector app=falco | grep user1

02:55:35.245247928: Error File below / or /root opened for writing
(user=root command=ysqlsh -U user1 -h yb-tserver-0 -d yugabyte
parent=bash file=/root/.psql_history program=ysqlsh
container_id=ac8028a3c3bc image=yugabytedb/yugabyte) k8s.ns=yb-demo
k8s
.pod=yb-tserver-0 container=ac8028a3c3bc k8s.ns=yb-demo
k8s.pod=yb-tserver-0 container=ac8028a3c3bc
```

From the above output, we can see that Falco has logged an error saying that **user=root** has run **command=ysqlsh** and connected to our YugabyteDB database.

That's it! At this point you can now start to build more complex rules in Falco to monitor and audit YugabyteDB in order to get an additional layer of defense in Kubernetes environments.