Cloud composer orchestrating Spark jobs on GKE using the Spark Operator - PoC

Introduction

This page is document the setup involved in getting the GCP cloud composer to orchestrate Spark applications on a remote GKE cluster. The composer uses the SparkKubernetesOpertaor to perform the job submission to the tenant GKE cluster. For this handshake to happen the composer environment needs to provide a kubernetes secret to the Kube API server/GKE control plane to get the request authenticated and authorised. Such sensitive information needs a dedicated secrets engine and Composer uses the Google Secret Manager as a secret's backend in this setup.

This proof of concept exercise designs the topology in a way that isolates the composer and GKE environments as much as possible to avoid any overlap of networking and security constructs that could be configured as default and stealthily working under the hood to oversimply the setup. To achieve this the following considerations were made.

- Composer and Spark GKE cluster are isolated over two separate GCP projects (in two different regions)

- An imaginary IaC and Configuration management tool performing the resource provisioning and bootstrapping the secrets across these environments.
- To keep it simple, the traffic between the two projects traverses the internet (however it is encrypted) .

Overview

The use case is where the developer produces Kubernetes ready Spark applications either using the inline coding technique or based on templates and need them orchestrated using a cloud composer environment. This orchestration is documented as Directed Acyclic Graph (DAG) that the composer understands and actions each step by engaging the appropriate operators to perform the task.

SparkKubernetesOperator is a purpose built Composer (Airflow) operator that is capable of encapsulating the mechanics of submitting Spark Applications to a Kubernetes environment. which has Spark Operator up and running and configured to

1. Developers write Kubernetes-ready Spark code and DAGs specifying the orchestration requirements for Spark jobs.
2. The Spark code and DAGs are stored in a Google Cloud Storage (GCS) bucket.
3. Cloud Composer, configured to use Google Secret Manager as its secrets backend, reads the DAGs from the GCS bucket.
4. For Spark Kubernetes operators, Cloud Composer packages job requests and fetches secrets from Google Secret Manager to connect to the Kubernetes cluster.
5. The Kubernetes cluster's API Server receives the job request and engages the Spark operator to spin up a Spark application on the Kubernetes cluster.

Additionally there is the infrastructure provisioning aspect where a jenkins like host will take the responsibility of (as shown in the animation below)

a) Provisiong the Composer

b) Bootstrapping the Composer with the Secret Manager

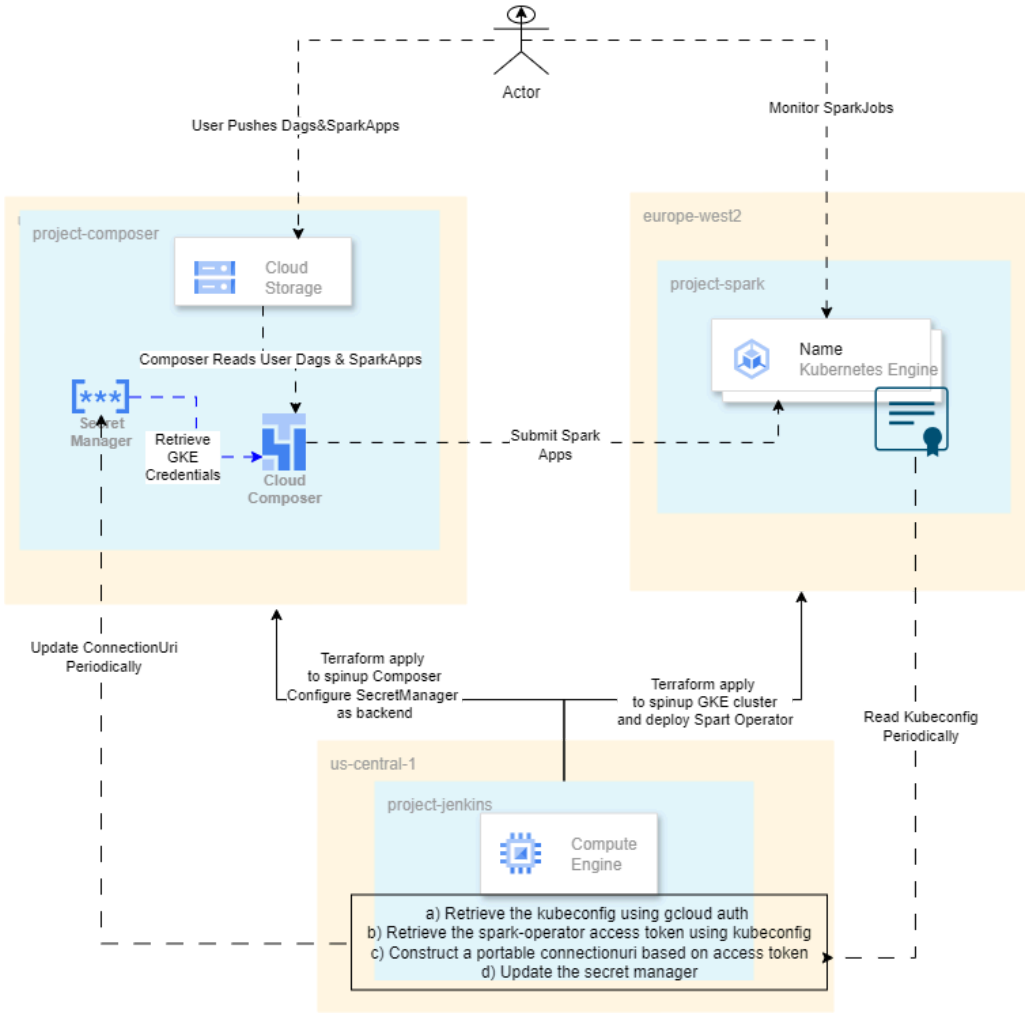
c) Deploying the Tenant Kubernetes Cluster

d) Deploying the Spark Applications and associated security configuration.

e) Derive the connectionUri that the composer need to authenticate with GKE and make it availabe in the Secrete manager.

(Note the connectionUri is based on Tokens that are short lived, so there needs to be a recurring process that rotates the secrets and reprovisions the ConnectionUri

Also consideration on the RACE conditions and dependency where the connectionUri creation doesn’t impact the SparkJobSubmissions inflight)



Technologies Stack

product	Relevance
Cloud Composer	Google Manager Airflow orchestration tool
Google Secret Manager	A fully managed Secret store and secret engine by Google
Google Kubernetes Engine	Google's Managed Kubernetes Clusters
Spark Operator	Intially a Google curated project to develop an execution platform for Spark jobs on Kubernetes Environment. This is now being managed by the kubeflow community

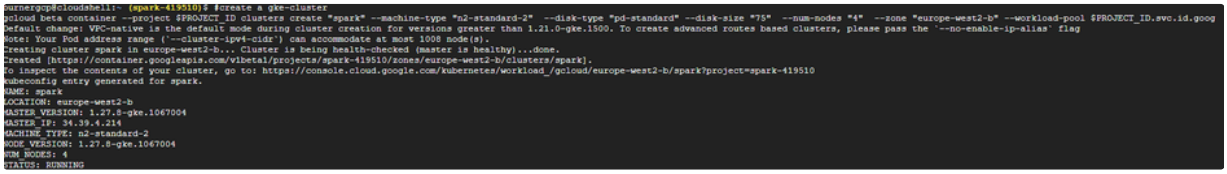
Installation of GKE Cluster and Spark Operator

```
burnergcp@cloudshell:~ (spark-419510)$ gcloud config list
[accessibility]
screen_reader = True
[component_manager]
disable_update_check = True
[compute]
gce_metadata_read_timeout_sec = 30
[core]
account = burnergcp@gmail.com
disable_usage_reporting = True
project = spark-419510
[metrics]
environment = devshell

Your active configuration is: [cloudshell-31147]
burnergcp@cloudshell:~ (spark-419510)$
```

Deploy a kubernetes Cluster

```
export PROJECT_ID=`gcloud config get project`

#create a gke-cluster
gcloud beta container --project $PROJECT_ID clusters create "spark" --machine-type "n2-standard-2"  --disk-type "
```

Wait for the cluster to be provisioned and runnning.



Cluster create command should get the GKE kubeconfig by default and place it in the default ~.kube/config

```
burnergcp@cloudshell:~ (spark-419510)$ ls -lrt .kube/config
-rw------- 1 burnergcp burnergcp 5525 Apr  8 21:47 .kube/config
burnergcp@cloudshell:~ (spark-419510)$
```

Inspect the kubeconfig file

Pay attention to the user section, there is no x509 crypto like we find in a regular kubernetes kubeconfig file.

The default GKE kubeconfig delegates authorisation to the ‘gke-gcloud-auth-plugin’ and seeks a jwt token once the oauth2 dance is performed successfully.

This implies the client (either REST APIs, or kubectl in most cases) need to have a means to authenticate with the Google IAM. (we will get back to this aspect later in the demo)

Install the Spark Operator

Now is the time to do some preparation

Create a namespace for the spark-operator

```
kubectl create ns spark-operator
```

```
burnergcp@cloudshell:~ (spark-419510)$ #create namespace for spark operator
kubectl create ns spark-operator
namespace/spark-operator created
burnergcp@cloudshell:~ (spark-419510)$ kubectl get ns
NAME              STATUS   AGE
default           Active   22m
gmp-public        Active   21m
gmp-system        Active   21m
kube-node-lease   Active   22m
kube-public       Active   22m
kube-system       Active   22m
spark-operator    Active   14s
burnergcp@cloudshell:~ (spark-419510)$
```

Create a service account and assign it the cluster edit role (which is quite permissive, but we'll keep it simple for the PoC)

```
kubectl create sa -n spark-operator spark
```

```
burnergcp@cloudshell:~ (spark-419510)$ kubectl create sa -n spark-operator spark
serviceaccount/spark created
```

Inspect the service account created

```
burnergcp@cloudshell:~ (spark-419510)$ k get sa -n spark-operator spark
NAME    SECRETS   AGE
spark   0         2m25s
burnergcp@cloudshell:~ (spark-419510)$
```

It appears to be created but there are no associated secrets created by default.

(This auto creation of a token has been disabled since the v1.24)

Create a cluster role binding to assign the cluster editor (This is very permissive but is ok for the PoC)

```
kubectl create clusterrolebinding spark-role --clusterrole=edit --serviceaccount=spark-operator:spark -n spark-op
```

```
burnergcp@cloudshell:~ (spark-419510)$ kubectl create clusterrolebinding spark-role --clusterrole=edit --serviceaccount=spark-operator:spark -n spark-operator
clusterrolebinding.rbac.authorization.k8s.io/spark-role created
burnergcp@cloudshell:~ (spark-419510)$
```

We now have the kubernetes security resource the service account for the pods to inherit which will empower then with the cluster edit role.

clone spark-operator repo

```
git clone https://github.com/kubeflow/spark-operator
```

Note: The spark-operator repository has been moved from GoogleCloud repo and being handled by the kubeflow community

```
burnergcp@cloudshell:~ (spark-419510)$ git clone https://github.com/kubeflow/spark-operator
Cloning into 'spark-operator'...
remote: Enumerating objects: 7487, done.
remote: Counting objects: 100% (352/352), done.
remote: Compressing objects: 100% (232/232), done.
remote: Total 7487 (delta 137), reused 233 (delta 98), pack-reused 7135
Receiving objects: 100% (7487/7487), 24.95 MiB | 24.12 MiB/s, done.
Resolving deltas: 100% (5087/5087), done.
burnergcp@cloudshell:~ (spark-419510)$
```

Install the spark operator using the helm chart from the git repo cloned.

```
#Install helm chart for spark operator
 helm install spark-operator spark-operator/charts/spark-operator-chart --namespace spark-operator
```

This will take a couple of minutes and should install the spark-operator

```
burnergcp@cloudshell:~ (spark-419510)$ #Install helm chart for spark operator
helm install spark-operator spark-operator/charts/spark-operator-chart --namespace spark-operator
NAME: spark-operator
LAST DEPLOYED: Tue Apr  9 07:17:04 2024
NAMESPACE: spark-operator
STATUS: deployed
REVISION: 1
TEST SUITE: None
burnergcp@cloudshell:~ (spark-419510)$
```

Check the objects in the spark-operator namespace

```
burnergcp@cloudshell:~ (spark-419510)$ kubectl get all -n spark-operator
NAME                                      READY   STATUS    RESTARTS   AGE
pod/spark-operator-77446ddc6c-2dctn       1/1     Running   0          99s

NAME                                READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/spark-operator      1/1     1            1           99s

NAME                                           DESIRED   CURRENT   READY   AGE
replicaset.apps/spark-operator-77446ddc6c      1         1         1       99s
burnergcp@cloudshell:~ (spark-419510)$
```

Check the spark custom resource definitions installed as part of the spark operator

```
burnergcp@cloudshell:~ (spark-419510)$ kubectl get crd|grep spark
scheduledsparkapplications.sparkoperator.k8s.io   2024-04-09T07:17:00Z
sparkapplications.sparkoperator.k8s.io            2024-04-09T07:17:00Z
burnergcp@cloudshell:~ (spark-419510)$
```

Now let's run a sample spark application to check if the spark operator works as expected.

The spark-operator repo provides some example applicaitons, however the image uses points to 'gcr.io/*' which is now deprecated

```
burnergcp@cloudshell:~/spark-operator/examples (spark-419510)$ grep 'image:' *
spark-operator-with-metrics.yaml:        image: gcr.io/spark-operator/spark-operator:v1beta2-1.3.0-3.1.1
spark-operator-with-webhook.yaml:        image: gcr.io/spark-operator/spark-operator:v1beta2-1.3.0-3.1.1
spark-operator-with-webhook.yaml:        image: gcr.io/spark-operator/spark-operator:v1beta2-1.3.0-3.1.1
spark-pi-configmap.yaml:  image: "gcr.io/spark-operator/spark:v3.1.1"
spark-pi-custom-resource.yaml:  image: "gcr.io/spark-operator/spark:v3.1.1"
spark-pi-prometheus.yaml:  image: "gcr.io/spark-operator/spark:v3.1.1-gcs-prometheus"
spark-pi-schedule.yaml:    image: "gcr.io/spark-operator/spark:v3.1.1"
spark-pi.yaml:  image: "gcr.io/spark-operator/spark-py:v3.1.1"
spark-py-pi.yaml:  image: "gcr.io/spark-operator/spark-py:v3.1.1"
burnergcp@cloudshell:~/spark-operator/examples (spark-419510)$
burnergcp@cloudshell:~/spark-operator/examples (spark-419510)$
```

The docker pull fails suggested the images are no more available on the gcr registry (These are being ported to the Artifact Registry.)

```
burnergcp@cloudshell:~/spark-operator/examples (spark-419510)$ docker pull gcr.io/spark-operator/spark-py:v3.1.1
Error response from daemon: manifest for gcr.io/spark-operator/spark-py:v3.1.1 not found: manifest unknown: Failed to fetch "v3.1.1" from request "/v2/spark-operator/spark-py/manifests/v3.1.1".
burnergcp@cloudshell:~/spark-operator/examples (spark-419510)$
```

At this point a custom Spark image can be built or use 'fstkfupg/spark-poc-shyam:3.1.1'.

The example we are going to use is '`spark-operator/examples/spark-pi.yaml`'. Let's replace the image and the namespace (to point to spark-operator)

```
#Update the image and namespace
sed -i 's/gcr.io\/spark-operator\/spark:v3.1.1/fstkfupg\/spark-poc-shyam:3.1.1/g' spark-operator/examples/spark-p
sed -i 's/namespace\: default/namespace\: spark-operator/g' spark-operator/examples/spark-pi.yaml
```

Now that we have the image changed let's create the spark application.

Deploy a test Spark Applicaton

kubectl apply -f spark-operator/examples/spark-pi.yaml

```
burnergcp@cloudshell:~ (spark-419510)$ k get sparkapplications.sparkoperator.k8s.io -n spark-operator
NAME       STATUS      ATTEMPTS   START                  FINISH                 AGE
spark-pi   COMPLETED   1          2024-04-09T07:20:02Z   2024-04-09T07:20:58Z   5m44s
burnergcp@cloudshell:~ (spark-419510)$
```

The Spark Application will launch a Driver and an Executor tasks and finally runs to completion,

Note the sequence of events (SparkApplicationAdded → SparkApplicationSubmitted →SparkDriverRunning → SparkExecutorPending → SparkExecutorRunning → SparkDriverCompleted ->SparkApplicationCompleted ->SparkExecutorCompleted)

```
kubectl describe sparkapplications -n spark-operator |grep 'Events:' -A10
sparkapplication.sparkoperator.k8s.io/spark-pi created
NAME       STATUS   ATTEMPTS   START   FINISH   AGE
spark-pi                                        0s
Events:
  Type    Reason                     Age   From            Message
  ----    ------                     ----  ----            -------
  Normal  SparkApplicationAdded      2m1s  spark-operator  SparkApplication spark-pi was added, enqueuing it for submission
  Normal  SparkApplicationSubmitted  116s  spark-operator  SparkApplication spark-pi was submitted successfully
  Normal  SparkDriverRunning         95s   spark-operator  Driver spark-pi-driver is running
  Normal  SparkExecutorPending       87s   spark-operator  Executor [spark-pi-14bcf88ec1bacd1d-exec-1] is pending
  Normal  SparkExecutorRunning       66s   spark-operator  Executor [spark-pi-14bcf88ec1bacd1d-exec-1] is running
  Normal  SparkDriverCompleted       60s   spark-operator  Driver spark-pi-driver completed
  Normal  SparkApplicationCompleted  60s   spark-operator  SparkApplication spark-pi completed
  Normal  SparkExecutorCompleted     58s   spark-operator  Executor [spark-pi-14bcf88ec1bacd1d-exec-1] completed
burnergcp@cloudshell:~ (spark-419510)$
```

Generate the connectionUri for authn/Authzn handshake

So far we have been running these using kubectl which in turn uses the .kube/config that is provided as part of the cluster creation. While the a regular kubeconfig file has enough details, i.e (cluster details, CA certs, Users details and User Tokens) to authenticate with a kubeAPI server the version that's provided by the GKE cluster federates the user auth with gloud-auth plugin. So this Bring Your Own Auth sort of kubeconfig will not suffice to authenticate from a client that can't integrate with gloud-auth-plugin. For this reason we need to find a way to get the composer operator authenticate with the GKE control plane over the client certificates.

This can be achieved by creating a dedicated secret 'airflow-secret' and bootstrapping with the 'spark-operator' Kubernetes Service Account.

Create a kubernetes Secret

```
apiVersion: v1
kind: Secret
metadata:
  name: airflow-secret
  namespace: spark-operator
  annotations:
    kubernetes.io/service-account.name: spark-operator
type: kubernetes.io/service-account-token


```

```
burnergcp@cloudshell:~ (spark-419510)$ cat secret.yaml
apiVersion: v1
kind: Secret
metadata:
  name: airflow-secret
  namespace: spark-operator
  annotations:
    kubernetes.io/service-account.name: spark
type: kubernetes.io/service-account-token
burnergcp@cloudshell:~ (spark-419510)$
```

```
burnergcp@cloudshell:~ (spark-419510)$ kubectl apply -f secret.yaml
secret/airflow-secret created
burnergcp@cloudshell:~ (spark-419510)$
```

Derive the Client Token

We should now be able to retrieve the secret token from the secret, assigning it to the TOKEN environment variable to be used downstream to ease creating the client kubernetes certificate.

TOKEN=`kubectl get secret airflow-secret -n spark-operator -o=jsonpath="{.data.token}"|base64 -d`

Also retrieve the client name for our curated kubeconfig

GKE_USER=`kubectl config view -o jsonpath='{.users.*.name}'`

echo $GKE_USER

```
gke_spark-419510_europe-west2-b_sparkburnergcp@cloudshell:~ (spark-419510)$ GKE_USER=`kubectl config view -o jsonpath='{.users.*.name}'`
burnergcp@cloudshell:~ (spark-419510)$
burnergcp@cloudshell:~ (spark-419510)$ echo $GKE_USER
gke_spark-419510_europe-west2-b_spark
burnergcp@cloudshell:~ (spark-419510)$
```

Let's add this to our current kubeconfig under the users section (we'll clean it up later)

k config set-credentials $GKE_USER --token=$TOKEN

```
burnergcp@cloudshell:~ (spark-419510)$ k config set-credentials $GKE_USER --token=$TOKEN
User "gke_spark-419510_europe-west2-b_spark" set.
burnergcp@cloudshell:~ (spark-419510)$
```

Now we have the default kubeconfig with the Server Address and the valid cluster CA, the only thing that needs updating is the user certificates retrieved from the secret. We can now remove the 'exec:' part as we don't need to rely on the auth plugin anymore.

(This certificate generation process on a CICD platform needs to be automated using similar steps, an ideal way is to use the Kubectl config set commands for cluster, User and Context details independently)

The curated kubeconfig as follows that uses the spark-operator identity

We now need to convert this certifcate into something the Composer (Airflow) understands, i.e the ‘connection_url’. For this purpose we need to wrap the kubeconfig into json format and ensure the tokens are not redacted'

kubectl config view -ojson

```
burnergcp@cloudshell:~ (spark-419510)$ kubectl config view -ojson
{
    "kind": "Config",
    "apiVersion": "v1",
    "preferences": {},
    "clusters": [
        {
            "name": "gke_spark-419510_europe-west2-b_spark",
            "cluster": {
                "server": "https://34.39.4.214",
                "certificate-authority-data": "DATA+OMITTED"
            }
        }
    ],
    "users": [
        {
            "name": "gke_spark-419510_europe-west2-b_spark",
            "user": {
                "token": "REDACTED"
            }
        }
    ],
    "contexts": [
        {
            "name": "gke_spark-419510_europe-west2-b_spark",
            "context": {
                "cluster": "gke_spark-419510_europe-west2-b_spark",
                "user": "gke_spark-419510_europe-west2-b_spark"
            }
        }
    ],
    "current-context": "gke_spark-419510_europe-west2-b_spark"
}
burnergcp@cloudshell:~ (spark-419510)$
```

Use the ‘--raw’ option to print the tokens to the kubeconfig file

Build a client token based kubeconfig

kubectl config view -ojson --raw > kubeconfig

```
burnergcp@cloudshell:~ (spark-419510)$ kubectl config view -ojson --raw > kubeconfig
burnergcp@cloudshell:~ (spark-419510)$
burnergcp@cloudshell:~ (spark-419510)$ ls -lrt kubeconfig
-rw-r--r-- 1 burnergcp burnergcp 3741 Apr  8 23:33 kubeconfig
burnergcp@cloudshell:~ (spark-419510)$
```

We need the use the airflow connection library to derive the connection_uri. thankfully there is a straight forward function *Connection.get_uri()* we can leverage.

Following is a quick script to print the connection_uri as expected by the composer. This connection_uri needs to made availabe to the composer environment , hence placed in the secret manager so that the airflow DAG can retrieve it when needed.

Wrap the Kubeconfig into airflow ready ConnectionUri

```
import json
from airflow.models.connection import Connection

with open('/home/burnergcp/kubeconfig') as f:
    kube_config_json = json.load(f)

c = Connection(
    conn_id='sparkgke',
    conn_type='kubernetes',
    extra=json.dumps(dict(
        in_cluster="false",
        kube_config=kube_config_json,
        namespace='spark-operator'))
)
print(c.get_uri())
```

That is all we need to do on the GKE side. We now move on to the Composer project to setup the Composer environment

Composer Environment Setup

Pre-requisites

Enable the Secret Manager API

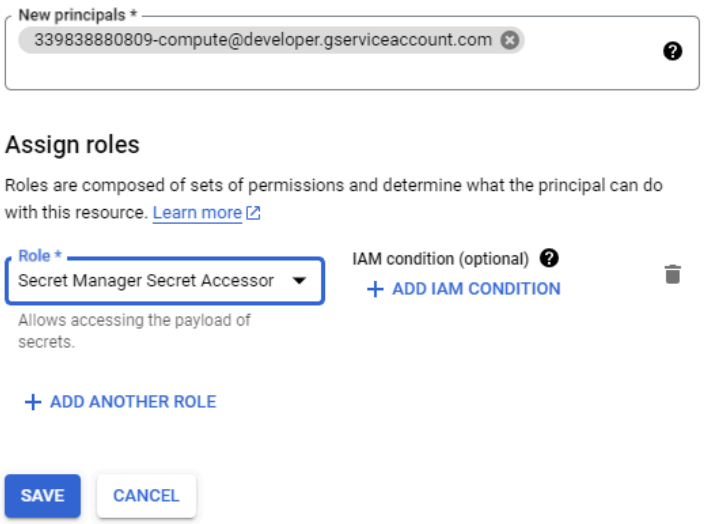
so that we can use the product in the first place

```
burnergcp@cloudshell:~ (composer-13322)$ gcloud services enable secretmanager.googleapis.com
Operation "operations/acat.p2-339838880809-e2c26e41-5fd4-45de-89d5-0a530c398bb9" finished successfully.
burnergcp@cloudshell:~ (composer-13322)$
```

The composer environment needs access to fetch secrets from Secret Manager, so it needs the ‘Secret Accessor Role’. As we are going to create the composer environment with the default compute SA in the PoC, granting the role the same.

Never do this production, always use a dedicated SA with curated permissions

Set up IAM for the PoC



Setup Secret Manager to host the connection_uri to access the Spark Operator on the GKE cluster

The connection_uri we managed to create using the client token based kubeconfig needs to be persisted in the Secret Manager. Composer is a bit touchy about the naming convention when it comes to fetching the actual connection. It expects the secret name to be prefixed with '*airflow-connections-*'

Create a new secret 'airflow-connections-sparkgke'. where 'airflow-connections-' is the default prefix sparkgke is our userdefined friendly connection_id name

The prefix configuration can be alteredusing some overrides but who cares for this PoC, we will stick to the defaults and use 'airflow-connections-'

So the composer during the execution time will expect the 'kubernetes_conn_id' as 'sparkgke' prepends 'airflow-connections-' and tries to resolve the secret and fetches it to kick off the spark operator.

et

with the secret value in the first versi

parkgke

ntifiable and unique within this project.

or import it directly from a file.

-b_spark%22%7D%7D%5D%2C+%22c
gke_spark-419510_europe-west2-
iespace=spark-operator

FE10D0D6

y

matically manages where this secret i

you can customize the locations by ch

essible regardless of how they are rep

t be changed after a secret is created

ocations for this secret

---

with a Google-managed key by defau

use a customer-managed key instea

ncryption key

uired

CANCEL

Validate Secret

Ensure the secret is created with the right prefix and can be retrieved.

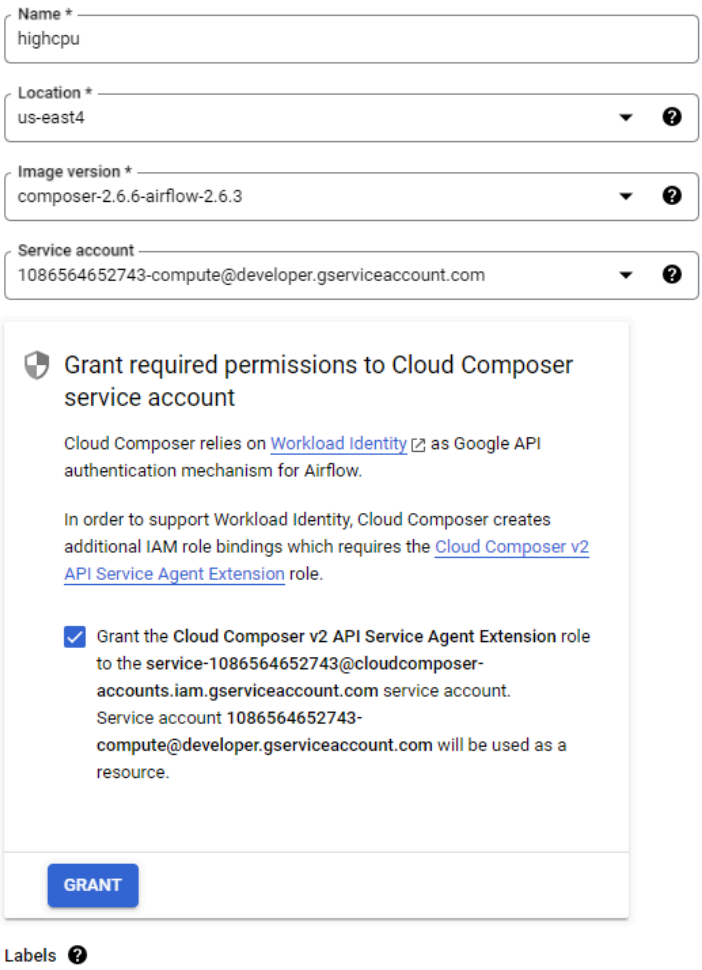
Use the following command to confirm the secret is created as expected with no typos, spaces or tabs

```
gcloud secrets versions access 1 --secret=airflow-connections-sparkgke; echo ""
```

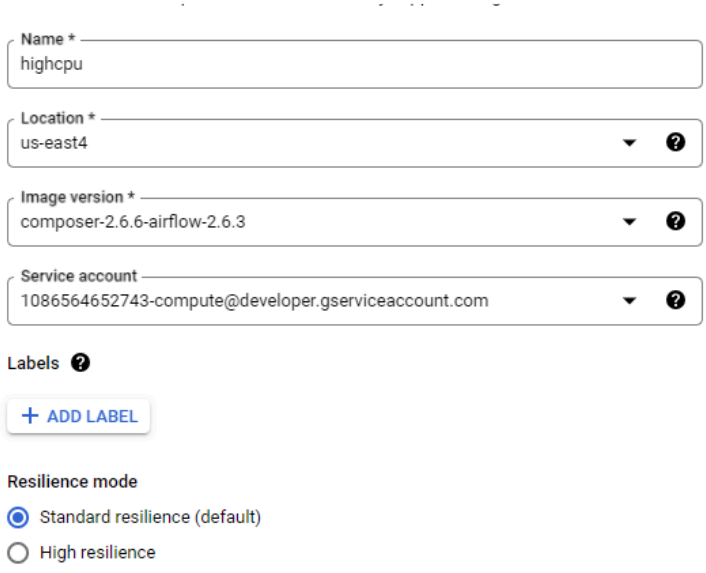
Create Composer Environment

Create a cloud composer environment

Leave the service account default to the compute developer (as discussed earlier), however it needs Cloud Composer V2 API Service Agent Extenstion role as it complains below. This can be set from the same screen without having to navigate to IAM screen or using gcloud command.
Check the box and click on Grant while the warning disappears

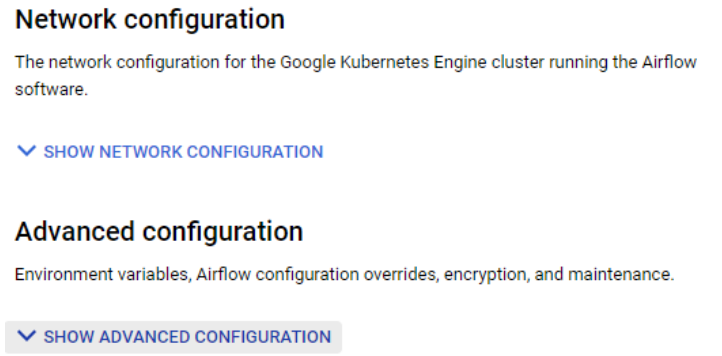


Now that the Cloud Composer V2 API Service Agent Extenstion role is assigned to the service account, composer create screen IAM validation is now OK.



Setup Secret Manager as the backend

The Composer needs an additional setting in the ‘Advanced Configuration’ section. This is to configure the backend where composer stores and retrieves its sensitive information. As the aim of the PoC is to prove that composer can use Google Secret Manager as the secret backend, this needs to be set up accordingly.

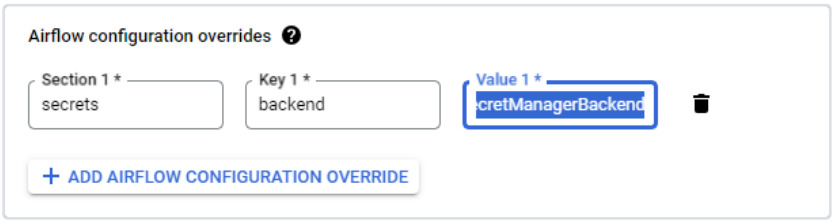


In the ‘Airflow configuration overides’ add a new configuration override

Section 1: secrets

Key 1: backend

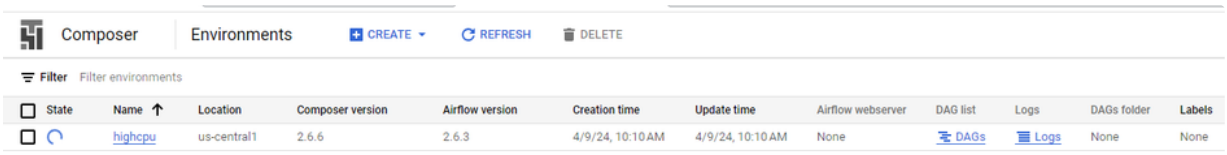
Value 1: `airflow.providers.google.cloud.secrets.secret_manager.CloudSecretManagerBackend`

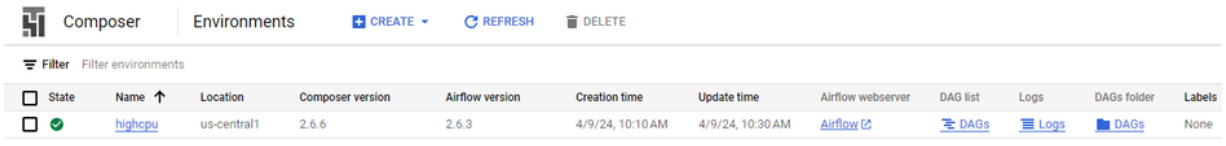


Click the 'Create' button to create the composer environment

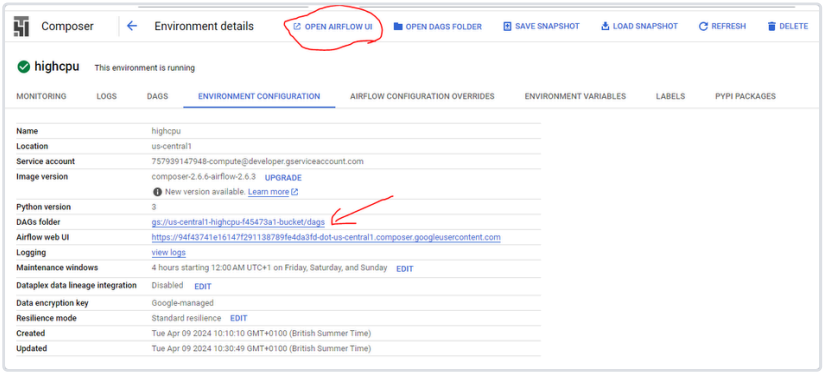
Create Composer Environment

The composer starts to spin up and might take 15-20 mins.





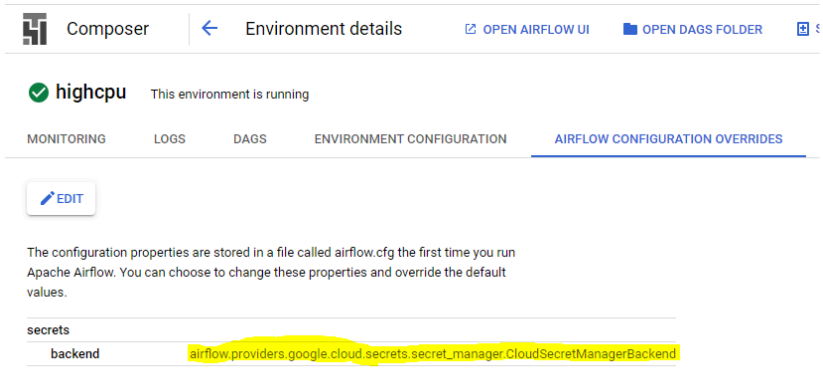
Once the Composer environment is ready, it provides the configuration as to where it fed the DAGs and application code from, Its the GCS bucket as listed below. Also the AIRFLOW UI button get enabled.



Validate the Secret Backend

Before you go ahead, it's important to ensure the secrets backed for the composer is indeed the Google Secrets Manager, this can be checked from the UI or using CLI

On the UI look for the 'AIRFLOW CONFIGURTAION OVERRIDES'



Or use the following gcloud command to list the backend

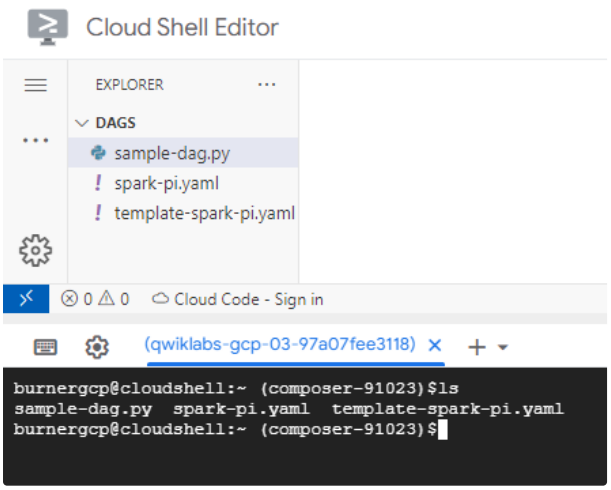
```
gcloud composer environments describe highcpu --location=us-east1|grep secrets
```

Once we are happy with the backend settings, let's move on to deploying the DAGs along with the associated application code.

Familiarise with the DAG and SparkApplications

For this PoC, let's use the same spark-py.yaml based Spark Application we deployed using the Kubectl locally on the tentant GKE cluster. We need a DAG to orchestrate the SparkOperator to be able to submit the job to the Cluster

As the PoC's aim is to highlight the Integration between the Composer and Spark Operator we chose the simplest scala application for test.



File	Purpose	Content
sample-dag.py	DAG to orchestrate the workflow	Click here to expand...

```
"""
This is an example
In this example, w
The first task is
and the second tas

Spark-on-k8s opera
https://github.com
"""

from datetime impo

# [START import_mo
# The DAG object;
from airflow impor
# Operators; we ne
from airflow.opera
from airflow.provi
from airflow.provi
from airflow.provi
from airflow.utils
#k8s_hook = Kuberr
# [END import_modu

# [START default_a
# These args will
# You can override
default_args = {
    'owner': 'gkes
    'depends_on_pa
    'start_date':
}
# [END default_arg

# [START instantia

dag = DAG(
    'spark_demo',
    start_date=day
```

		<details><summary>Click here to expand...</summary><pre>
default_args=d
schedule_inter
tags=['secmana
)
Define the BashO
hello_world_task =
task_id='hello
bash_command='
dag=dag
)

Define the task

submit = SparkKube
task_id='spark
namespace='spa
application_fi
kubernetes_cor
do_xcom_push=T
)

hello_world_task >
</pre></details> |
| template-spark-pi.yaml | A template file, basically a pointer to the actual Spark Application manifest | <pre>/home/airflow/gcs/dags/sp</pre> |
| spark-pi.yaml | The yaml that contains the Spark Application manifest as provided in the Spark Operator Examples | <details><summary>Click here to expand...</summary><pre>
apiVersion: "spark
kind: SparkApplica
metadata:
  name: spark-pi
  namespace: spark
spec:
  type: Scala
  mode: cluster
  # Replce with yo
  image: "fstkfupg
  imagePullPolicy:
  mainClass: org.a
  mainApplicationF
  sparkVersion: "3
  restartPolicy:
    type: Never
  volumes:
    - name: "test-
      hostPath:
        path: "/tm
        type: Dire
  driver:
    cores: 1
    coreLimit: "12
    memory: "512m"
    labels:
      version: 3.1
    serviceAccount
</pre></details> |

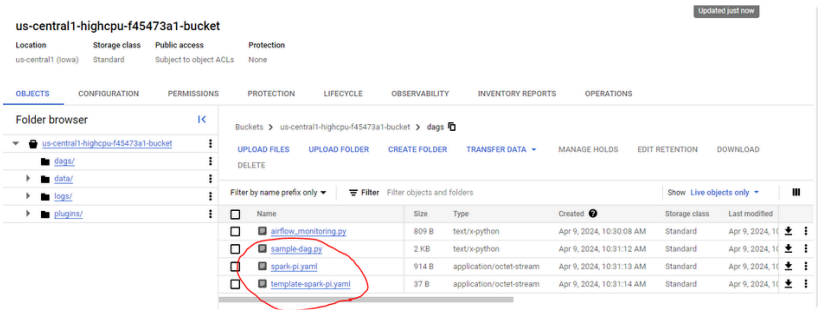
```
    volumeMounts:
      - name: "tes
        mountPath:
  executor:
    cores: 1
    instances: 1
    memory: "512m"
    labels:
      version: 3.1
    volumeMounts:
      - name: "tes
        mountPath:

```

Deploy the DAG

Copy the dag and associated spark code to the gcs bucket that the composer is configured to

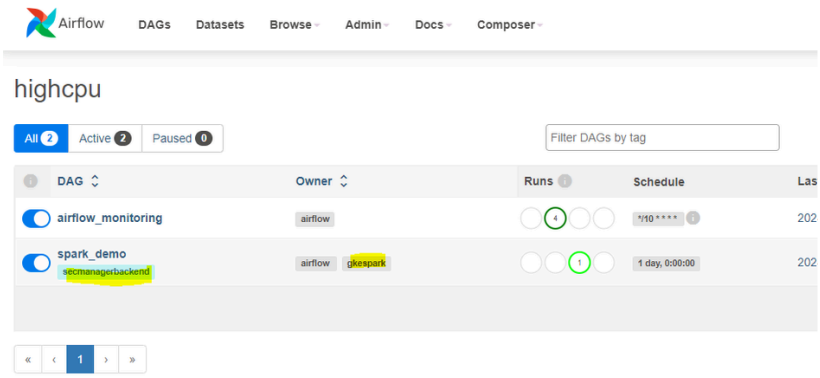
```
burnergcp@cloudshell:~ (composer-91023)$ls -lrt
total 4
drwxr-xr-x 2 student_01_01625eba5c88 student_01_01625eba5c88 4096 Apr  9 08:37 dags
burnergcp@cloudshell:~ (composer-91023)$cd dags
burnergcp@cloudshell:~ (composer-91023)$ls -lrt
total 12
-rw-r--r-- 1 student_01_01625eba5c88 student_01_01625eba5c88   37 Apr  9 08:36 template-spark-pi.yaml
-rw-r--r-- 1 student_01_01625eba5c88 student_01_01625eba5c88  914 Apr  9 08:36 spark-pi.yaml
-rw-r--r-- 1 student_01_01625eba5c88 student_01_01625eba5c88 2027 Apr  9 08:37 sample-dag.py
burnergcp@cloudshell:~ (composer-91023)$
burnergcp@cloudshell:~ (composer-91023)$gsutil cp * gs://us-central1-highcpu-f45473a1-bucket/dags
Copying file://sample-dag.py [Content-Type=text/x-python]...
Copying file://spark-pi.yaml [Content-Type=application/octet-stream]...
Copying file://template-spark-pi.yaml [Content-Type=application/octet-stream]...
\ [3 files][  2.9 KiB/  2.9 KiB]
Operation completed over 3 objects/2.9 KiB.
burnergcp@cloudshell:~ (composer-91023)$
```

They seem to be availabe in the dags folder on the GCS bucket Composer reads from

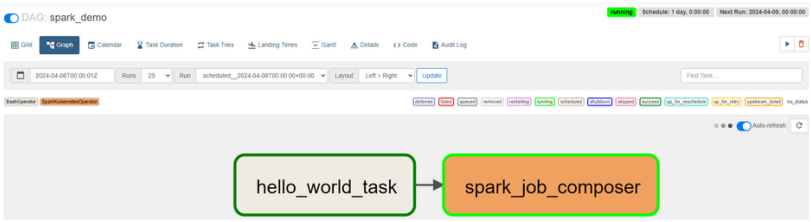


The worker nodes of composer will eventually read the DAGs and expose them on the UI (note that the tags and owner metadata is reflected on the UI for ease of management)

Monitor the Orchestration



Submit a manual run



Clean up the Spark applications in the spark-operator namespace

```
burnergcp@cloudshell:~ (spark-419510)$ k get sparkapplications.sparkoperator.k8s.io -n spark-operator
No resources found in spark-operator namespace.
burnergcp@cloudshell:~ (spark-419510)$
```

We now see the Composer DAG's SparkKubernetesOperator sumbmitting the spark-py.yaml using the connection_id 'sparkgke'

```
burnergcp@cloudshell:~ (spark-419510)$ k get sparkapplications.sparkoperator.k8s.io -n spark-operator
NAME                        STATUS    ATTEMPTS   START                  FINISH       AGE
spark-job-composer-xt9riotq   RUNNING   1          2024-04-09T13:32:43Z   <no value>   21s
burnergcp@cloudshell:~ (spark-419510)$
```

```
burnergcp@cloudshell:~ (spark-419510)$ k get pods -n spark-operator
NAME                                  READY   STATUS      RESTARTS      AGE
spark-job-composer-gw5gnszc-driver    0/1     Completed   0             19s
spark-job-composer-xt9riotq-driver    0/1     Completed   0             107s
spark-operator-77446ddc6c-ctwdn       1/1     Running     2 (92m ago)   93m
burnergcp@cloudshell:~ (spark-419510)$
```