Cloud Composer and Vertex AI DataProc

Overview

Introduction

Workflows are a common use case in data analytics - they involve ingesting, transforming, and analyzing data to find the meaningful information within. In Google Cloud Platform, the tool for orchestrating workflows is Cloud Composer, which is a hosted version of the popular open source workflow tool Apache Airflow. In this lab, you will use Cloud Composer to create a simple workflow that creates a Cloud Dataproc cluster, analyzes it using Cloud Dataproc and Apache Hadoop, then deletes the Cloud Dataproc cluster afterwards.

What is Cloud Composer?

Cloud Composer is a fully managed workflow orchestration service that empowers you to author, schedule, and monitor pipelines that span across clouds and on-premises data centers. Built on the popular Apache Airflow open source project and operated using the Python programming language, Cloud Composer is free from lock-in and easy to use.

By using Cloud Composer instead of a local instance of Apache Airflow, users can benefit from the best of Airflow with no installation or management overhead.

For more info about Cloud Composer, check out the docs!

What is Apache Airflow?

Apache Airflow is an open source tool used to programmatically author, schedule, and monitor workflows. There are a few key terms to remember relating to Airflow that you'll see throughout the experiment:

- DAG - a DAG (Directed Acyclic Graph) is a collection of organized tasks that you want to schedule and run. DAGs, also called workflows, are defined in standard Python files
- Operator - an operator describes a single task in a workflow

What is Cloud Dataproc?

Cloud Dataproc is Google Cloud Platform's fully-managed Apache Spark and Apache Hadoop service. Cloud Dataproc easily integrates with other GCP services, giving you a powerful and complete platform for data processing, analytics and machine learning.

For more info about Cloud Dataproc, check out the docs!

What you'll do

This experiment shows you how to create and run an Apache Airflow workflow in Cloud Composer that completes the following tasks:

- Creates a Cloud Dataproc cluster
- Runs an Apache Hadoop wordcount job on the cluster, and outputs its results to Cloud Storage
- Deletes the cluster

What you'll learn

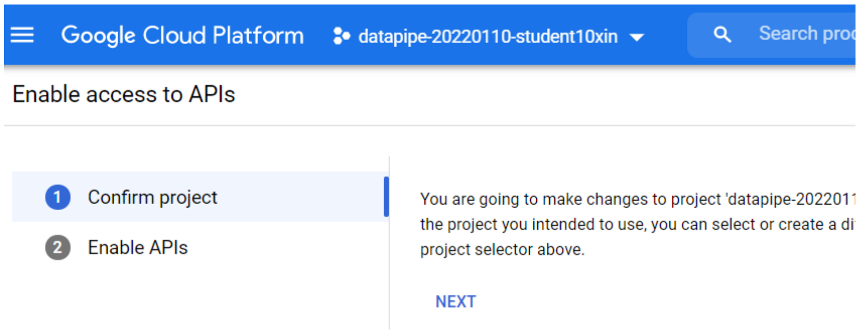
- How to create and run an Apache Airflow workflow in Cloud Composer
- How to use Cloud Composer and Cloud Dataproc to run an analytic on a dataset
- How to access your Cloud Composer environment through the Google Cloud Platform Console, Cloud SDK, and Airflow web interface

What you'll need

- GCP account provided by the instructor
- Basic CLI knowledge
- Basic understanding of Python

Enable the Cloud Composer, Cloud Dataproc, and Cloud Storage APIs by clicking the following URL

https://console.cloud.google.com/flows/enableapi?apiid=composer.googleapis.com,dataproc.googleapis.com,storage-component.googleapis.com&_ga=2.153805112.-1287172231.1548439725


Google Cloud Platform datapipe-20220110-student10xin Search proc

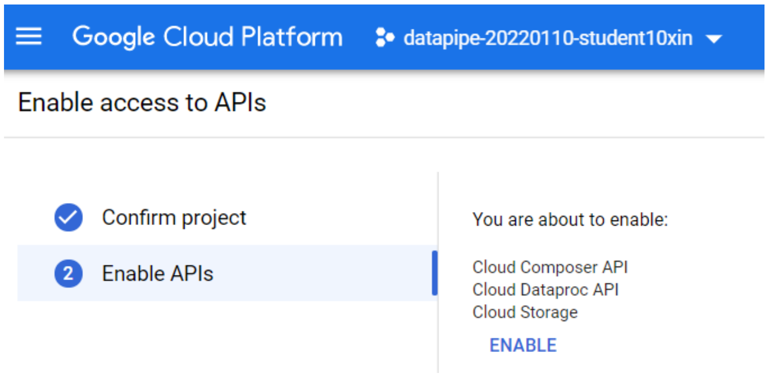
Enable access to APIs

1 Confirm project
2 Enable APIs

You are going to make changes to project 'datapipe-202201
the project you intended to use, you can select or create a di
project selector above.

NEXT

Choose **NEXT** after verifying that the correct datapipe project is noted.


Google Cloud Platform datapipe-20220110-student10xin

Enable access to APIs

✓ Confirm project
2 Enable APIs

You are about to enable:

Cloud Composer API
Cloud Dataproc API
Cloud Storage

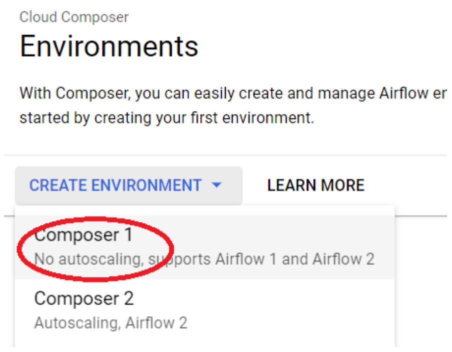
ENABLE

Choose **ENABLE**

Once they are enabled, you can ignore the button that says "Go to Credentials", if it is shown, and proceed to the next step of the experiment.

Cloud Composer Creation

Navigate to the Cloud Composer service console.


Cloud Composer

Environments

With Composer, you can easily create and manage Airflow er
started by creating your first environment.

CREATE ENVIRONMENT LEARN MORE

Composer 1
No autoscaling, supports Airflow 1 and Airflow 2

Composer 2
Autoscaling, Airflow 2

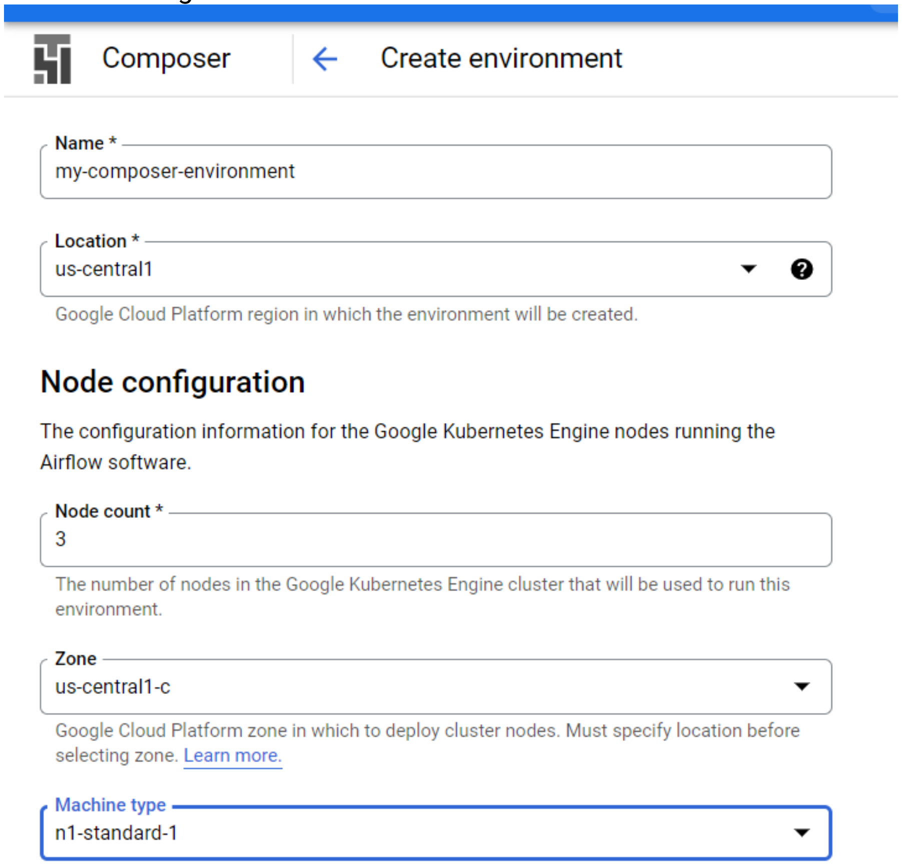
Choose **CREATE ENVIRONMENT** and select the **Composer 1** type.

Create a Cloud Composer environment with the following configuration :

- Name: **my-composer-environment**
- Location: us-central1

- Zone: us-central1-a

All other configurations can remain at their default.



Composer | Create environment

Name *
my-composer-environment

Location *
us-central1

Google Cloud Platform region in which the environment will be created.

Node configuration

The configuration information for the Google Kubernetes Engine nodes running the Airflow software.

Node count *
3

The number of nodes in the Google Kubernetes Engine cluster that will be used to run this environment.

Zone
us-central1-c

Google Cloud Platform zone in which to deploy cluster nodes. Must specify location before selecting zone. Learn more.

Machine type
n1-standard-1

Click **CREATE** the bottom.

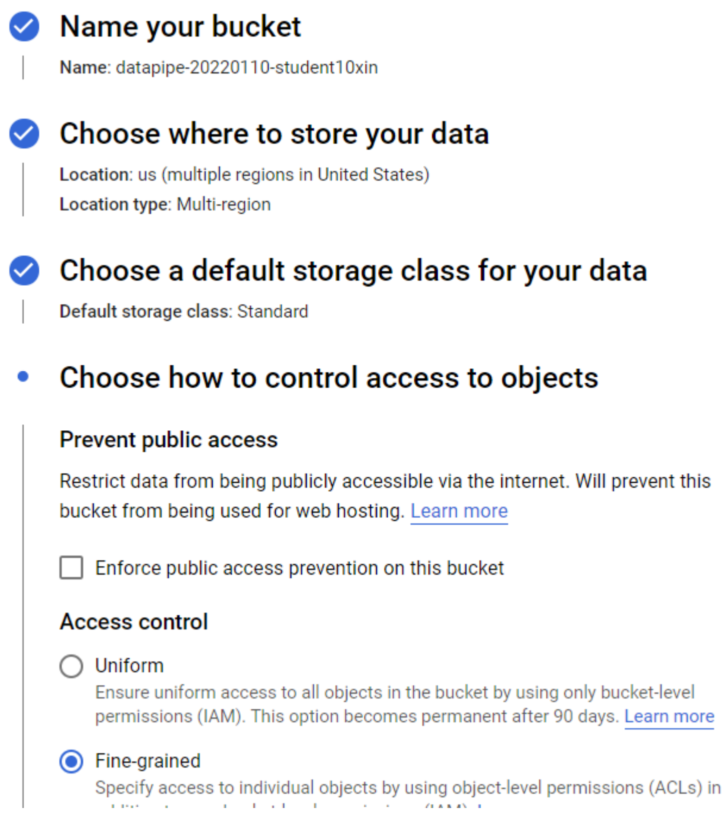
Note: The environment creation process is completed when the green checkmark displays to the left of the environment name on the **Environments** page in the GCP Console. It will take a few minutes to create, and it is okay to navigate away from this page while Environment creation is in progress.

Create Cloud Storage

In your project, create a Cloud Storage bucket with the following configuration:

- Name: <your-project-id>
- Default storage class: Multi-regional
- Location: United States
- Access Control Model: fine-grained

Click **CREATE**


Name your bucket

Name: datapipe-20220110-student10xin

Choose where to store your data

Location: us (multiple regions in United States)

Location type: Multi-region

Choose a default storage class for your data

Default storage class: Standard

Choose how to control access to objects

Prevent public access

Restrict data from being publicly accessible via the internet. Will prevent this bucket from being used for web hosting. Learn more

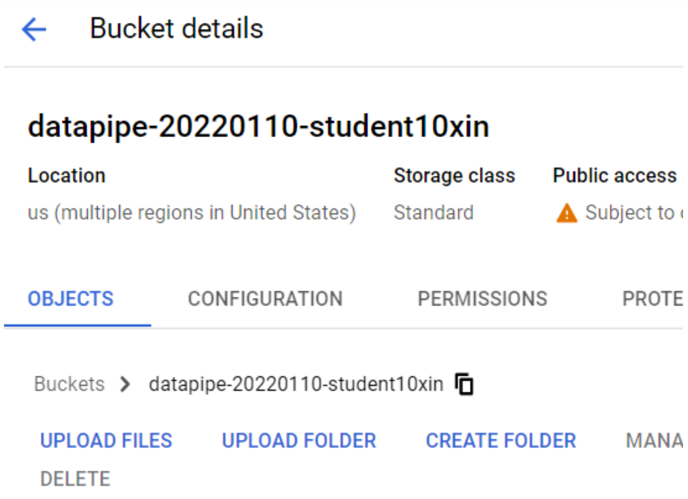
☐ Enforce public access prevention on this bucket

Access control

○ Uniform
Ensure uniform access to all objects in the bucket by using only bucket-level permissions (IAM). This option becomes permanent after 90 days. Learn more

● Fine-grained
Specify access to individual objects by using object-level permissions (ACLs) in

Bucket is created for this experiment


Bucket details

datapipe-20220110-student10xin

Location	Storage class	Public access
us (multiple regions in United States)	Standard	⚠ Subject to

Setting Up Apache Airflow

If your Composer Environment has not finished being created, you can skip ahead to the Sample Workflow step and come back to this part after.

Viewing Composer Environment Information

In the GCP Console, open the Environments page

Click the name of the environment to see its details.

The **Environment details** page provides information, such as the Airflow web interface URL, Google Kubernetes Engine cluster ID, name of the Cloud Storage bucket, and path for the /dags folder.

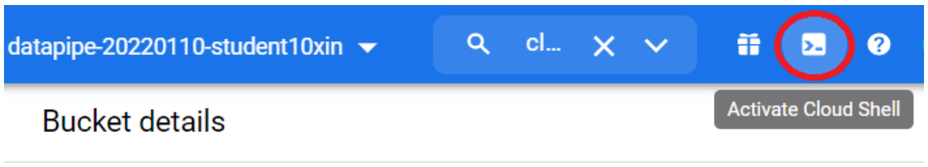
In Airflow, a DAG (Directed Acyclic Graph) is a collection of organized tasks that you want to schedule and run. DAGs, also called workflows, are defined in standard Python files. Cloud Composer only schedules the DAGs in the /dags folder. The /dags folder is in the Cloud Storage bucket that Cloud Composer creates automatically when you create your environment.

Setting Apache Airflow Environment Variables

Apache Airflow variables are an Airflow-specific concept that is distinct from environment variables. In this step, you'll set the following four Airflow variables: `gcp_project`, `gcs_bucket`, `gce_region` and `gce_zone`.

Using `gcloud` to Set Variables

First, open up your Cloud Shell, which has the Cloud SDK conveniently installed for you.



Set the account being used for Cloud Shell with the account email that you were provided by the instructor

```
gcloud config set account student10@innovationinsoftware.com
```

Set the environment variable `COMPOSER_INSTANCE` to the name of your Composer environment

```
COMPOSER_INSTANCE=my-composer-environment
```

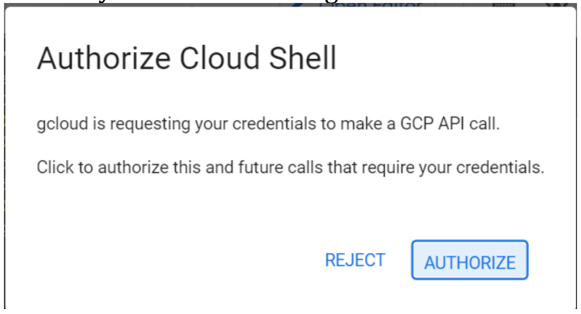
To set Airflow variables using the gcloud command-line tool, use the `gcloud composer environments run` command with the `variables` sub-command. This `gcloud composer` command executes the Airflow CLI sub-command `variables`. The sub-command passes the arguments to the `gcloud` command line tool.

You'll run this command three times, replacing the variables with the ones relevant to your project.

Set the `gcp_project` using the following command, replacing <your-project-id> with your project ID.

```
gcloud composer environments run ${COMPOSER_INSTANCE} \
  --location us-central1 variables -- --set gcp_project <your-project-id>
```

Note: If you are shown the gcloud authorization dialog choose AUTHORIZE


Authorize Cloud Shell

gcloud is requesting your credentials to make a GCP API call.

Click to authorize this and future calls that require your credentials.

Your output will look something like this

```
kubeconfig entry generated for us-central1-my-composer-env-123abc-gke.
Executing within the following Kubernetes cluster namespace: composer-1-10-0-airflow-1-10-2-123abc
[2020-04-17 20:42:49,713] {settings.py:176} INFO - settings.configure_orm(): Using pool settings.
pool_size=5, pool_recycle=1800, pid=449
[2020-04-17 20:42:50,123] {default_celery.py:90} WARNING - You have configured a result_backend of
redis://airflow-redis-service.default.svc.cluste
r.local:6379/0, it is highly recommended to use an alternative result_backend (i.e. a database).
[2020-04-17 20:42:50,127] {__init__.py:51} INFO - Using executor CeleryExecutor
[2020-04-17 20:42:50,433] {app.py:52} WARNING - Using default Composer Environment Variables.
Overrides have not been applied.
[2020-04-17 20:42:50,440] {configuration.py:522} INFO - Reading the config from
/etc/airflow/airflow.cfg
[2020-04-17 20:42:50,452] {configuration.py:522} INFO - Reading the config from
/etc/airflow/airflow.cfg
```

Command will show something that looks like an error but is not.

```
[2022-01-11 08:12:54,907] {configuration.py:732} INFO - Reading the config from
/etc/airflow/airflow.cfg
[2022-01-11 08:12:55,298] {configuration.py:732} INFO - Reading the config from
/etc/airflow/airflow.cfg
```

The 'variables' command is deprecated and removed in Airflow 2.0, please use 'variables list' instead

This is a known issue in GKE and can be ignored.

Set the `gcs_bucket` using the following command, replacing `<your-bucket-name>` with the multi-region bucket you created in setup earlier. If you followed our recommendation, your bucket name is the same as your project ID. Your output will be similar to the previous command.

```
gcloud composer environments run ${COMPOSER_INSTANCE} \
   --location us-central1 variables -- --set gcs_bucket gs://<your-bucket-name>
```

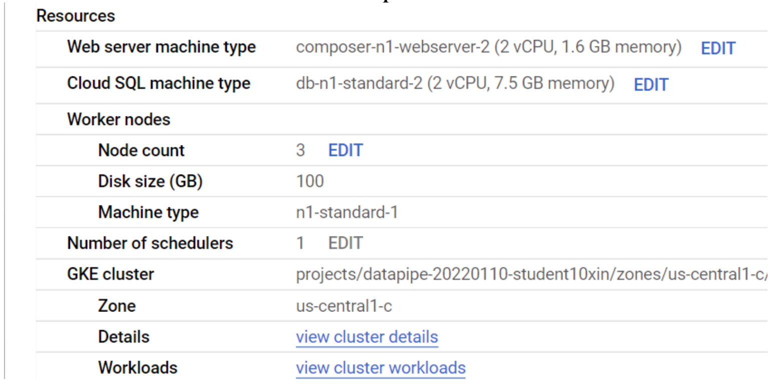
Set the `gce_zone` using the following command. Your output will be similar to the previous commands.

```
gcloud composer environments run ${COMPOSER_INSTANCE} \
   --location us-central1 variables -- --set gce_zone us-central1-a
```

Set the `gce_region` using the following command. Your output will be similar to the previous commands.

```
gcloud composer environments run ${COMPOSER_INSTANCE} \
   --location us-central1 variables -- --set gce_region us-central1
```

If we forget what zone we created the Composer environment in we can view that from the Environment tab Resources in the Cloud Composer service console.



Resources

Web server machine type	composer-n1-webserver-2 (2 vCPU, 1.6 GB memory)	EDIT
Cloud SQL machine type	db-n1-standard-2 (2 vCPU, 7.5 GB memory)	EDIT
Worker nodes		
Node count	3	EDIT
Disk size (GB)	100	
Machine type	n1-standard-1	
Number of schedulers	1	EDIT
GKE cluster	projects/datapipe-20220110-student10xin/zones/us-central1-c/	
Zone	us-central1-c	
Details	view cluster details	
Workloads	view cluster workloads	

(Optional) Using gcloud **to view a variable**

To see the value of a variable, run the Airflow CLI sub-command `variables` with the `get` argument or use the Airflow UI.

For example:

```
gcloud composer environments run ${COMPOSER_INSTANCE} \
   --location us-central1 variables --  --get gcs_bucket
```

You can do this with any of the three variables you just set: `gcp_project`, `gcs_bucket`, and `gce_zone`.

Sample Workflow

Let's take a look at the code for the DAG we'll be using in this step. Don't worry about downloading files yet, just follow along here.

There's a lot to unpack here, so let's break it down a bit.

```
from airflow import models
from airflow.contrib.operators import dataproc_operator
from airflow.utils import trigger_rule
```

We start off with some Airflow imports:

- `airflow.models` - Allows us to access and create data in the Airflow database.
- `airflow.contrib.operators` - Where operators from the community live. In this case, we need the `dataproc_operator` to access the Cloud Dataproc API.
- `airflow.utils.trigger_rule` - For adding trigger rules to our operators. Trigger rules allow fine-grained control over whether an operator should execute in relation to the status of its parents.

```
output_file = os.path.join(
    models.Variable.get('gcs_bucket'), 'wordcount',
    datetime.datetime.now().strftime('%Y%m%d-%H%M%S')) + os.sep
```

This specifies the location of our output file. The notable line here is `models.Variable.get('gcs_bucket')` which will grab the `gcs_bucket` variable value from the Airflow database.

```
WORDCOUNT_JAR = (
    'file:///usr/lib/hadoop-mapreduce/hadoop-mapreduce-examples.jar'
)

input_file = 'gs://pub/shakespeare/rose.txt'

wordcount_args = ['wordcount', input_file, output_file]
```

- `WORDCOUNT_JAR` - Location of the .jar file we'll eventually run on the Cloud Dataproc cluster. It is already hosted on GCP for you.
- `input_file` - Location of the file containing the data our Hadoop job will eventually compute on. We'll upload the data to that location together in Step 5.
- `wordcount_args` - Arguments that we'll pass into the jar file.

```
yesterday = datetime.datetime.combine(
    datetime.datetime.today() - datetime.timedelta(1),
    datetime.datetime.min.time())
```

This will give us a datetime object equivalent representing midnight on the previous day. For instance, if this is executed at 11:00 on March 4th, the datetime object would represent 00:00 on March 3rd. This has to do with how Airflow handles scheduling. More info on that can be found here.

```
default_dag_args = {
   'start_date': yesterday,
   'email_on_failure': False,
   'email_on_retry': False,
   'retries': 1,
   'retry_delay': datetime.timedelta(minutes=5),
   'project_id': models.Variable.get('gcp_project')
}
```

The `default_dag_args` variable in the form of a dictionary should be supplied whenever a new DAG is created:

- `'email_on_failure'` - Indicates whether email alerts should be sent when a task failed
- `'email_on_retry'` - Indicates whether email alerts should be sent when a task is retried
- `'retries'` - Denotes how many retry attempts Airflow should make in the case of a DAG failure
- `'retry_delay'` - Denotes how long Airflow should wait before attempting a retry
- `'project_id'` - Tells the DAG what GCP Project ID to associate it with, which will be needed later with the Dataproc Operator

```
with models.DAG(
      'composer_hadoop_tutorial',
      schedule_interval=datetime.timedelta(days=1),
      default_args=default_dag_args) as dag:
```

Using `with models.DAG` tells the script to include everything below it inside of the same DAG. We also see three arguments passed in:

- The first, a string, is the name to give the DAG that we're creating. In this case, we're using `composer_hadoop_tutorial`.
- `schedule_interval` - A `datetime.timedelta` object, which here we have set to one day. This means that this DAG will attempt to execute once a day after the `'start_date'` that was set earlier in `'default_dag_args'`
- `default_args` - The dictionary we created earlier containing the default arguments for the DAG

Create a Dataproc Cluster

Next, we'll create a `dataproc_operator.DataprocClusterCreateOperator` which creates a Cloud Dataproc Cluster.

```
   create_dataproc_cluster = dataproc_operator.DataprocClusterCreateOperator(
      task_id='create_dataproc_cluster',
      cluster_name='composer-hadoop-tutorial-cluster-{{ ds_nodash }}',
      num_workers=2,
      zone=models.Variable.get('gce_zone'),
      master_machine_type='n1-standard-1',
      worker_machine_type='n1-standard-1')
```

Within this operator, we see a few arguments, all but the first of which are specific to this operator:

- `task_id` - Just like in the BashOperator, this is the name we assign to the operator, which is viewable from the Airflow UI
- `cluster_name` - The name we assign the Cloud Dataproc cluster. Here, we've named it `composer-hadoop-tutorial-cluster-{{ ds_nodash }}` (see info box for optional additional information)
- `num_workers` - The number of workers we allocate to the Cloud Dataproc cluster
- `zone` - The geographical region where we want the cluster to live, as saved within the Airflow database. This will read the `'gce_zone'` variable we set in Step 3
- `master_machine_type` - The type of machine we want to allocate to the Cloud Dataproc master
- `worker_machine_type` - The type of machine we want to allocate to the Cloud Dataproc worker

Additional Information about the `cluster_name` variable

The `{{ ds_nodash }}` part of the parameter is there because Airflow supports jinja2 templating. It is a parameter that gets rendered by Airflow at runtime every time the operator kicks off. In this case, `{{ ds_nodash }}` gets replaced with the execution_date of the DAG in YYYYMMDD format. If you're unfamiliar with jinja2 templating, that's fine! No knowledge of it is needed to complete this experiment.

Submit an Apache Hadoop Job

The `dataproc_operator.DataProcHadoopOperator` allows us to submit a job to a Cloud Dataproc cluster.

```
  run_dataproc_hadoop = dataproc_operator.DataProcHadoopOperator(
    task_id='run_dataproc_hadoop',
    main_jar=WORDCOUNT_JAR,
    cluster_name='composer-hadoop-tutorial-cluster-{{ ds_nodash }}',
    arguments=wordcount_args)
```

We provide several parameters:

- `task_id` - Name we assign to this piece of the DAG
- `main_jar` - Location of the .jar file we wish to run against the cluster
- `cluster_name` - Name of the cluster to run the job against, which you'll notice is identical to what we find in the previous operator

- `arguments` - Arguments that get passed into the jar file, as you would if executing the .jar file from the command line

Delete the Cluster

The last operator we'll create is the `dataproc_operator.DataprocClusterDeleteOperator`.

```
  delete_dataproc_cluster = dataproc_operator.DataprocClusterDeleteOperator(
    task_id='delete_dataproc_cluster',
    cluster_name='composer-hadoop-tutorial-cluster-{{ ds_nodash }}',
    trigger_rule=trigger_rule.TriggerRule.ALL_DONE)
```

As the name suggests, this operator will delete a given Cloud Dataproc cluster. We see three arguments here:

- `task_id` - Just like in the BashOperator, this is the name we assign to the operator, which is viewable from the Airflow UI
- `cluster_name` - The name we assign the Cloud Dataproc cluster. Here, we've named it `composer-hadoop-tutorial-cluster-{{ ds_nodash }}` (see info box after "Create a Dataproc Cluster" for optional additional information)
- `trigger_rule` - We mentioned Trigger Rules briefly during the imports at the beginning of this step, but here we have one in action. By default, an Airflow operator does not execute unless all of its upstream operators have successfully completed. The `ALL_DONE` trigger rule only requires that all upstream operators have completed, regardless of whether or not they were successful. Here this means that even if the Hadoop job failed, we still want to tear the cluster down.

```
 create_dataproc_cluster >> run_dataproc_hadoop >> delete_dataproc_cluster
```

Lastly, we want these operators to execute in a particular order, and we can denote this by using Python bitshift operators. In this case, `create_dataproc_cluster` will always execute first, followed by `run_dataproc_hadoop` and finally `delete_dataproc_cluster`.

Putting it all together, the code looks like this:

```
# Copyright 2018 Google LLC
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

# [START composer_hadoop_tutorial]
"""Example Airflow DAG that creates a Cloud Dataproc cluster, runs the Hadoop
wordcount example, and deletes the cluster.

This DAG relies on three Airflow variables
https://airflow.apache.org/concepts.html#variables
* gcp_project - Google Cloud Project to use for the Cloud Dataproc cluster.
* gce_zone - Google Compute Engine zone where Cloud Dataproc cluster should be
  created.
* gcs_bucket - Google Cloud Storage bucket to use for result of Hadoop job.
  See https://cloud.google.com/storage/docs/creating-buckets for creating a
  bucket.
"""

import datetime
import os

from airflow import models
from airflow.contrib.operators import dataproc_operator
from airflow.utils import trigger_rule

# Output file for Cloud Dataproc job.
output_file = os.path.join(
    models.Variable.get('gcs_bucket'), 'wordcount',
    datetime.datetime.now().strftime('%Y%m%d-%H%M%S')) + os.sep
# Path to Hadoop wordcount example available on every Dataproc cluster.
WORDCOUNT_JAR = (
    'file:///usr/lib/hadoop-mapreduce/hadoop-mapreduce-examples.jar'
)
# Arguments to pass to Cloud Dataproc job.
input_file = 'gs://pub/shakespeare/rose.txt'

wordcount_args = ['wordcount', input_file, output_file]

yesterday = datetime.datetime.combine(
    datetime.datetime.today() - datetime.timedelta(1),
    datetime.datetime.min.time())

default_dag_args = {
    # Setting start date as yesterday starts the DAG immediately when it is
    # detected in the Cloud Storage bucket.
    'start_date': yesterday,
    # To email on failure or retry set 'email' arg to your email and enable
    # emailing here.
    'email_on_failure': False,
    'email_on_retry': False,
    # If a task fails, retry it once after waiting at least 5 minutes
    'retries': 1,
    'retry_delay': datetime.timedelta(minutes=5),
    'project_id': models.Variable.get('gcp_project')
}

# [START composer_hadoop_schedule]
with models.DAG(
        'composer_hadoop_tutorial',
        # Continue to run DAG once per day
        schedule_interval=datetime.timedelta(days=1),
        default_args=default_dag_args) as dag:
    # [END composer_hadoop_schedule]

    # Create a Cloud Dataproc cluster.
    create_dataproc_cluster = dataproc_operator.DataprocClusterCreateOperator(
        task_id='create_dataproc_cluster',
        # Give the cluster a unique name by appending the date scheduled.
        # See https://airflow.apache.org/code.html#default-variables
        cluster_name='composer-hadoop-tutorial-cluster-{{ ds_nodash }}',
        num_workers=2,
        zone=models.Variable.get('gce_zone'),
        master_machine_type='n1-standard-1',
        worker_machine_type='n1-standard-1')

    # Run the Hadoop wordcount example installed on the Cloud Dataproc cluster
    # master node.
    run_dataproc_hadoop = dataproc_operator.DataProcHadoopOperator(
        task_id='run_dataproc_hadoop',
        main_jar=WORDCOUNT_JAR,
        cluster_name='composer-hadoop-tutorial-cluster-{{ ds_nodash }}',
        arguments=wordcount_args)

    # Delete Cloud Dataproc cluster.
    delete_dataproc_cluster = dataproc_operator.DataprocClusterDeleteOperator(
        task_id='delete_dataproc_cluster',
        cluster_name='composer-hadoop-tutorial-cluster-{{ ds_nodash }}',
        # Setting trigger_rule to ALL_DONE causes the cluster to be deleted
        # even if the Dataproc job fails.
        trigger_rule=trigger_rule.TriggerRule.ALL_DONE)

    # [START composer_hadoop_steps]
    # Define DAG dependencies.
    create_dataproc_cluster >> run_dataproc_hadoop >> delete_dataproc_cluster
    # [END composer_hadoop_steps]

# [END composer_hadoop]
```

Critical Success Note: If you skipped Step: Setting up Apache Airflow because your environment wasn't created, make sure to go back to it now before proceeding to the next steps.

Upload Airflow Files to Cloud Storage

Copy the DAG to Your /dags Folder

1. First, open up your Cloud Shell, which has the Cloud SDK conveniently installed for you.
2. Clone the python samples repo and change to the composer/workflows directory

```
git clone https://github.com/GoogleCloudPlatform/python-docs-samples.git && cd python-docs-samples/composer/workflows
```

3. Run the following command to set the name of your DAGs folder to an environment variable

```
DAGS_FOLDER=$(gcloud composer environments describe ${COMPOSER_INSTANCE} \
--location us-central1 --format="value(config.dagGcsPrefix)")
```

4. Run the following `gsutil` command to copy the tutorial code to where your /dags folder is created

```
gsutil cp hadoop_tutorial.py $DAGS_FOLDER
```

Your output will look something like this:

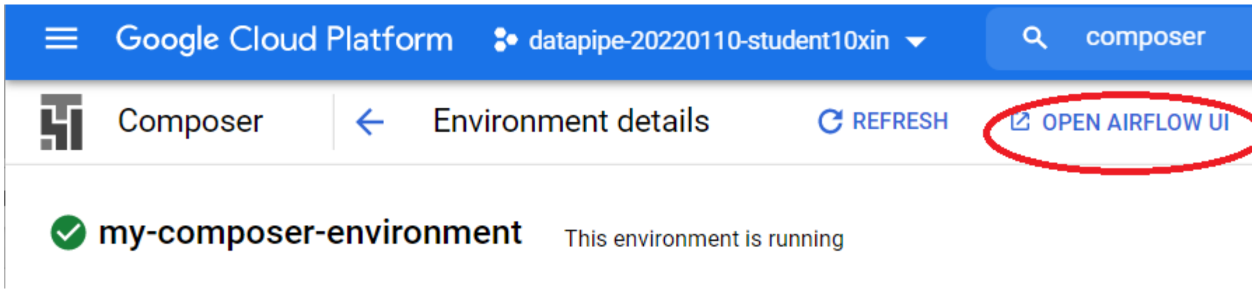
```
Copying file://hadoop_tutorial.py [Content-Type=text/x-python]...
/ [1 files][  4.1 KiB/  4.1 KiB]
Operation completed over 1 objects/4.1 KiB.
```

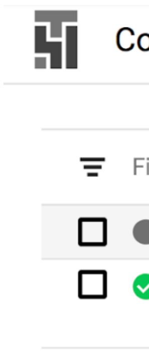
Note: Experiment `composer_hadoop_tutorial`

Using the Airflow UI

To access the Airflow web interface using the GCP console:

1. Open the **Environments** page.
2. In the **Airflow webserver** column for the environment, click the new window icon. The Airflow web UI opens in a new browser window.

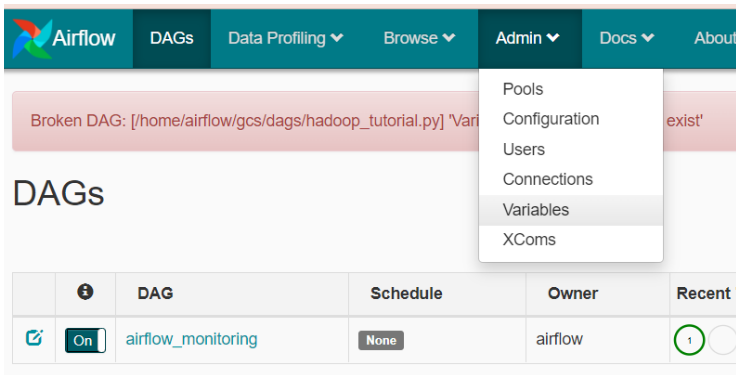




For information about the Airflow UI, see Accessing the web interface.

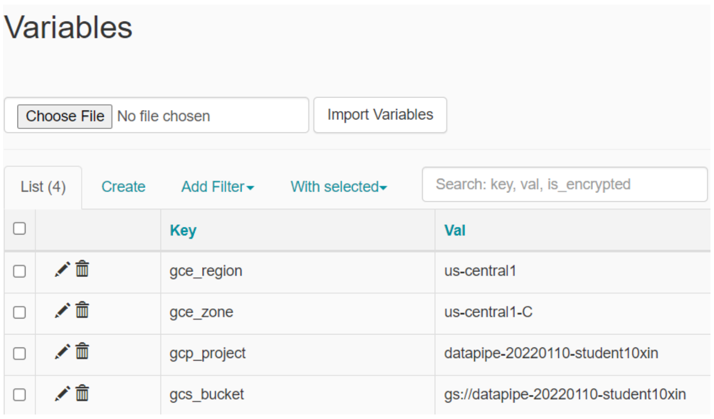
View Variables

The variables you set earlier are persisted in your environment. You can view the variables by selecting **Admin > Variables** from the Airflow UI menu bar.



Broken DAG: [/home/airflow/gcs/dags/hadoop_tutorial.py] 'Vari ... exist'

		DAG	Schedule	Owner	Recent
	On	airflow_monitoring	None	airflow	1



Variables

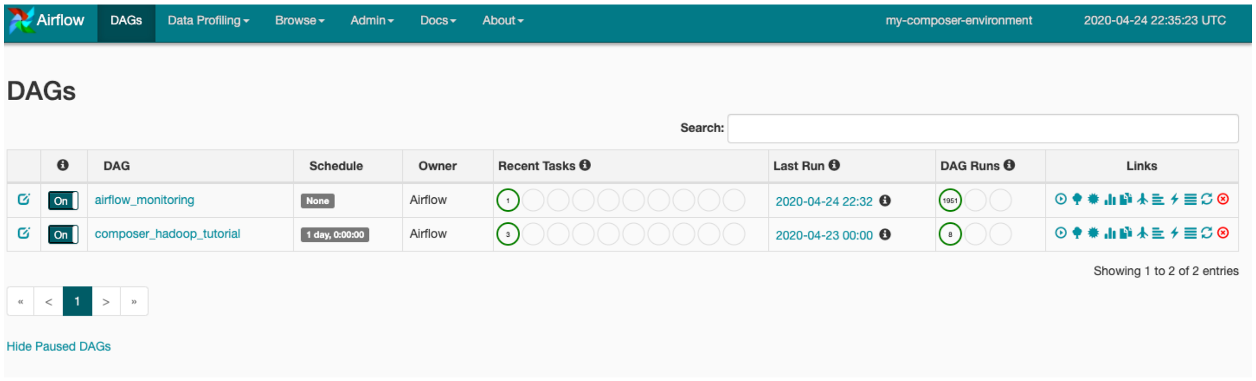
Choose File No file chosen | Import Variables

List (4) | Create | Add Filter | With selected | Search: key, val, is_encrypted

	Key	Val
	gce_region	us-central1
	gce_zone	us-central1-C
	gcp_project	datapipe-20220110-student10xin
	gcs_bucket	gs://datapipe-20220110-student10xin

Exploring DAG Runs

When you upload your DAG file to the `dags` folder in Cloud Storage, Cloud Composer parses the file. If no errors are found, the name of the workflow appears in the DAG listing, and the workflow is queued to run immediately. To look at your DAGs, click on **DAGs** at the top of the page.



Airflow | DAGs | Data Profiling | Browse | Admin | Docs | About | my-composer-environment | 2020-04-24 22:35:23 UTC

DAGs

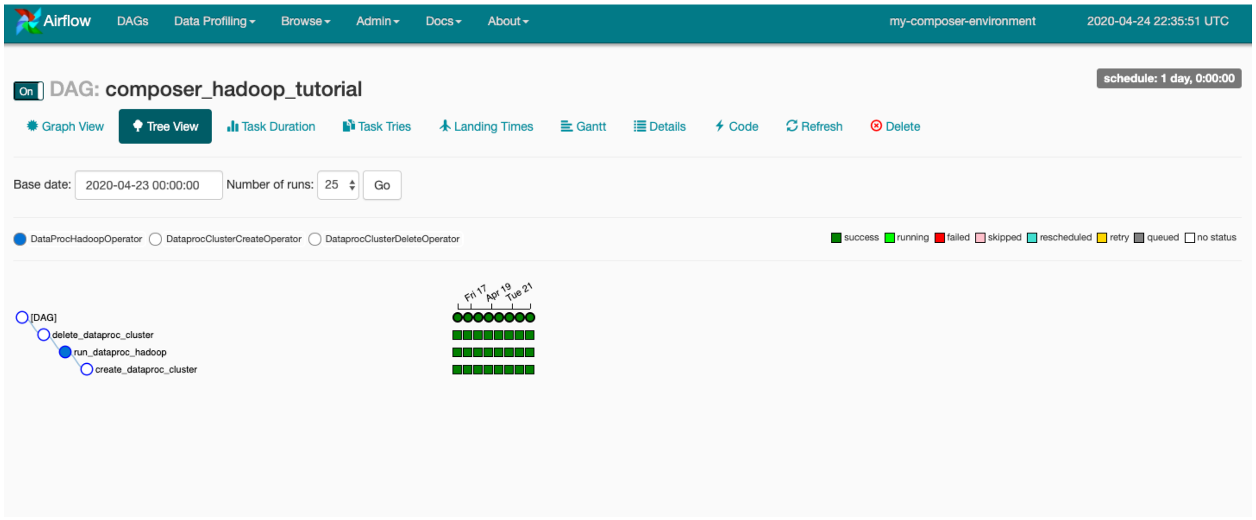
Search:

		DAG	Schedule	Owner	Recent Tasks	Last Run	DAG Runs	Links
	On	airflow_monitoring	None	Airflow	1	2020-04-24 22:32	1951	
	On	composer_hadoop_tutorial	1 day, 0:00:00	Airflow	3	2020-04-23 00:00	8	

Showing 1 to 2 of 2 entries

Hide Paused DAGs

Click `composer_hadoop_tutorial` to open the DAG details page. This page includes a graphical representation of workflow tasks and dependencies.



Airflow | DAGs | Data Profiling | Browse | Admin | Docs | About | my-composer-environment | 2020-04-24 22:35:51 UTC

On DAG: composer_hadoop_tutorial

schedule: 1 day, 0:00:00

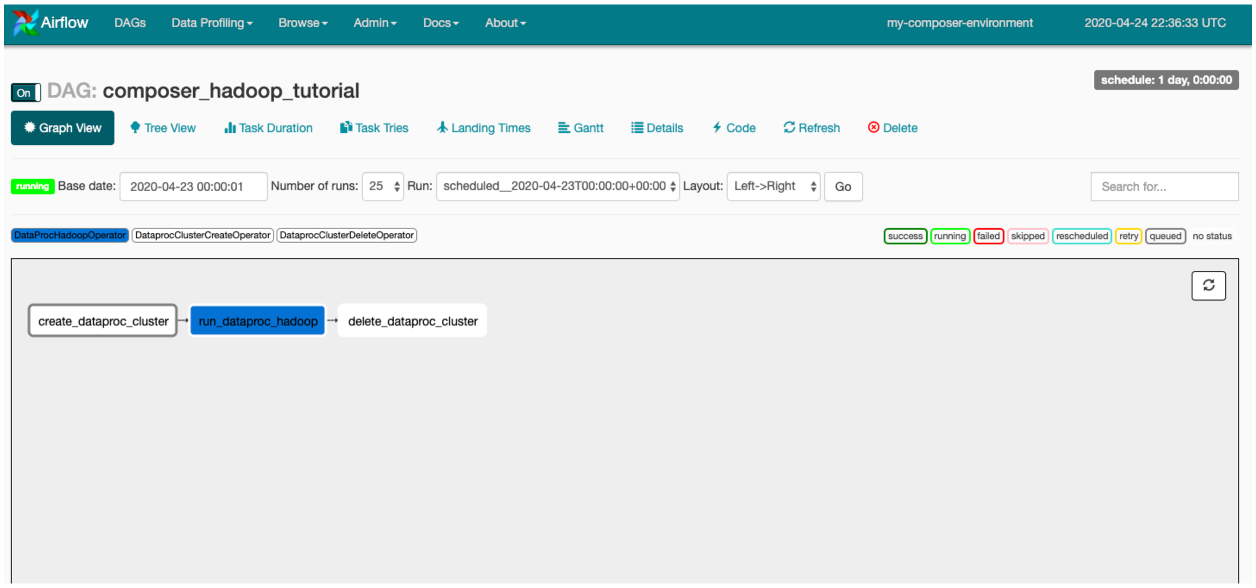
Graph View | Tree View | Task Duration | Task Tries | Landing Times | Gantt | Details | Code | Refresh | Delete

Base date: 2020-04-23 00:00:00 Number of runs: 25 Go

DataProcHadoopOperator DataprocClusterCreateOperator DataprocClusterDeleteOperator

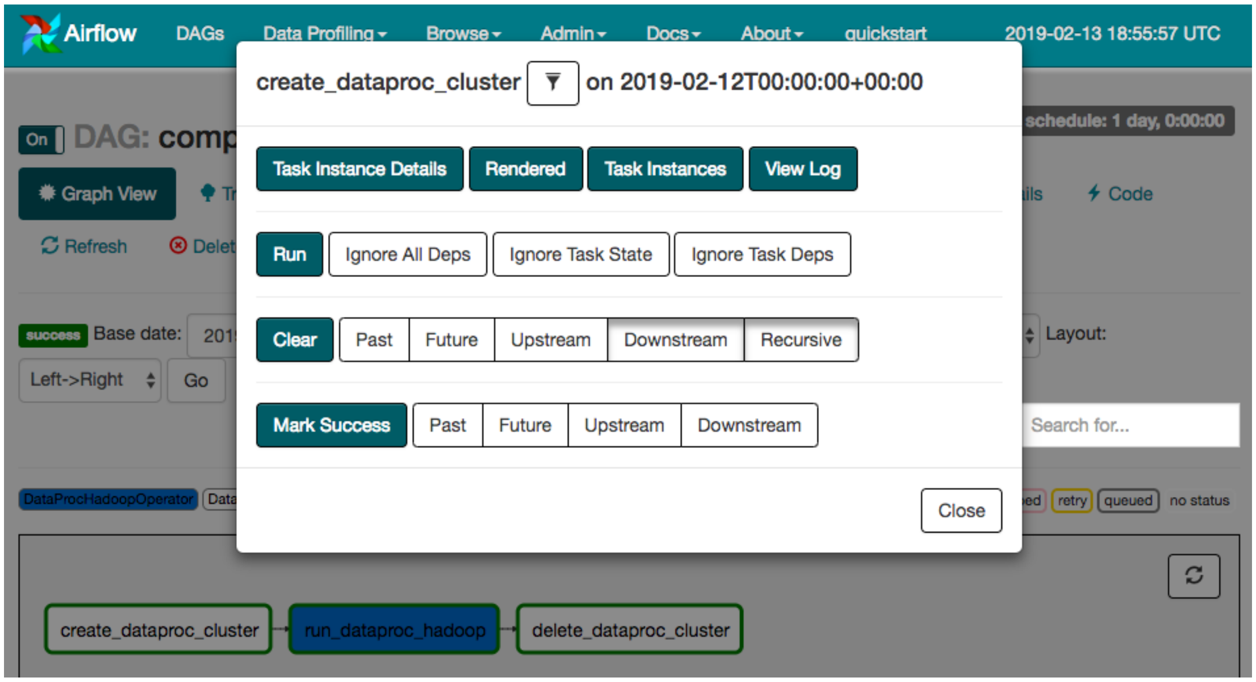
success running failed skipped rescheduled retry queued no status

Now, in the toolbar, click **Graph View** and then mouseover the graphic for each task to see its status. Note that the border around each task also indicates the status (green border = running; red = failed, etc.).



To run the workflow again from the **Graph View**:

1. In the Airflow UI Graph View, click the `create_dataproc_cluster` graphic.
2. Click **Clear** to reset the three tasks and then click **OK** to confirm.



You can also check the status and results of the `composer-hadoop-tutorial` workflow by going to the following GCP Console pages:

- Cloud Dataproc Clusters to monitor cluster creation and deletion. Note that the cluster created by the workflow is ephemeral: it only exists for the duration of the workflow and is deleted as part of the last workflow task.
- Cloud Dataproc Jobs to view or monitor the Apache Hadoop wordcount job. Click the Job ID to see job log output.
- Cloud Storage Browser to see the results of the wordcount in the `wordcount` folder in the Cloud Storage bucket you created for this experiment.

Cleanup

To avoid incurring charges to your GCP account for the resources used in this experiment:

1. (Optional) To save your data, download the data from the Cloud Storage bucket for the Cloud Composer environment and the storage bucket you created for experiment.
2. Delete the Cloud Storage bucket you created for this experiment.
3. Delete the Cloud Storage bucket for the environment.
4. Delete the Cloud Composer environment. Note that deleting your environment does not delete the storage bucket for the environment.

You've now learned to do some cool stuff with Cloud Composer and DataProc.

Congratulations!

In this experiment, you created and ran an emoji pipeline.