

Airflow, FaaS and Data Ingestion

Overview

Introduction

Trigger DAGs with Cloud Functions

Airflow is designed to run DAGs on a regular schedule, but you can also trigger DAGs in response to events. One way to do this is to use [Cloud Functions](#) to trigger Cloud Composer DAGs when a specified event occurs. For example, you can create a function that triggers a DAG when an object changes in a Cloud Storage bucket, or when a message is pushed to a Pub/Sub topic.

The example in this guide runs a DAG every time a change occurs in a Cloud Storage bucket. Changes to any object in a bucket trigger a function. This function makes a request to Airflow REST API of your Cloud Composer environment. Airflow processes this request and runs a DAG. The DAG outputs information about the change.

Before you begin

Enable APIs for your project

- Enable the Cloud Functions APIs.
[Enable the APIs](#)

Enable the Airflow REST API

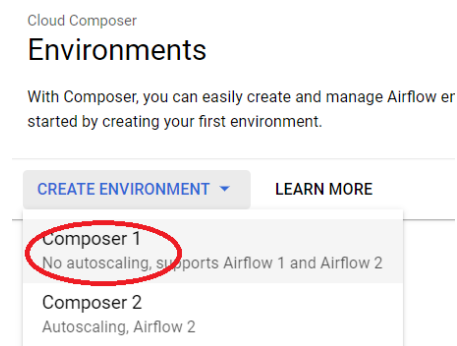
Depending on your version of Airflow you could have to enable this.

Note: for our chosen V2 Airflow environment in this experiment it is already enabled:

- For Airflow 2, the stable REST API is already enabled by default. If your environment has the stable API disabled, then [enable the stable REST API](#).
- For Airflow 1, [enable the experimental REST API](#). [View on GitHub](#) [Feedback](#)

Cloud Composer Creation

Navigate to the Cloud Composer service console.



Choose **CREATE ENVIRONMENT** and select the **Composer 1** type, this has the Airflow Webserver in a combined environment and this experiment is designed for that environment.

Create a Cloud Composer environment with the following configuration :

- Name: **serverless-composer-pubsub-environment**
- Location: us-central1
- Zone: us-central1-a

All other configurations can remain at their default.



Name *

serverless-composer-pubsub-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



For this implementation we'll use an Apache Airflow 2.x environment.

Google Cloud Platform

datapipe-20220110-student10xin



Composer

← Create environment

The list of instance tags applied to all node VMs. Tags are used to identify valid sources or targets for network firewalls. Each tag with the list must comply with RFC1035. Cannot be updated.

Image version

composer-1.17.7-airflow-2.0.2

The image version to use in the created environment. Image version value includes Composer version and Airflow version. Must specify location before selecting image version.

Number of schedulers *

1

Your environment can run more than one Airflow scheduler at the same time.

Python version

3

The Python version to use in the created environment. Must specify an image version before selecting Python version.

After select the region, zone and machine type, change the image to 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

This experiment triggers a DAG in response to changes in a Cloud Storage bucket. [create a new bucket](#) to use in this example.

Note: Do not use Cloud Composer environment buckets because this might cause the function to trigger several times per second. For the purposes of this example, we recommend to create a separate bucket.

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

- Name: *PROJECT_ID*-ingestion-bucket
- For example if you were student10, datapipe-20220110-student10xin-ingestion-bucket
- Add a label with key = project and value = keywest

✓ Name your bucket

Pick a globally unique, permanent name. [Naming guidelines](#)

datapipe-20220110-student10xin-ingestion-bucket

Tip: Don't include any sensitive information

Labels (optional)

Labels are key:value pairs that allow you to group related buckets together or with other Cloud Platform resources. [Learn more](#)

Key

project

Value

keywest



+ ADD LABEL

^ SHOW LESS

CONTINUE

Click **CONTINUE**

- Default storage class: Region
- Location: us-central1

✓ Name your bucket

Name: datapipe-20220110-student10xin-ingestion-bucket

• Choose where to store your data

This permanent choice defines the geographic placement of your data and affects cost, performance, and availability. [Learn more](#)

Location type

- ☐ Multi-region
Highest availability across largest area
- ☐ Dual-region
High availability and low latency across 2 regions
- ☒ Region
Lowest latency within a single region

Location

us-central1 (Iowa)




CONTINUE

Click **CONTINUE**

- Standard storage class for operational buckets

- **Choose a default storage class for your data**

A storage class sets costs for storage, retrieval, and operations. Pick a default storage class based on how long you plan to store your data and how often it will be accessed. [Learn more](#)

- ☒ **Standard** 
Best for short-term storage and frequently accessed data
- ☐ **Nearline**
Best for backups and data accessed less than once a month
- ☐ **Coldline**
Best for disaster recovery and data accessed less than once a quarter
- ☐ **Archive**
Best for long-term digital preservation of data accessed less than once a year

CONTINUE

Click **CONTINUE**

- Access Control Model: fine-grained to allow both objects and buckets to have permissions

- **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 addition to your bucket-level permissions (IAM). [Learn more](#)

Click **CONTINUE**

- Choose object versioning to allow for objects to never be overwritten in place and update the version expiration to 1 day

- **Choose how to protect object data**

Your data is always protected with Cloud Storage but you can also choose from these additional data protection options to prevent data loss. Note that object versioning and retention policies cannot be used together.

Protection tools

☐ None

☒ **Object versioning (best for data recovery)**

For restoring deleted or overwritten objects. To minimize the cost of storing versions, we recommend limiting the number of noncurrent versions per object and scheduling them to expire after a number of days. [Learn more](#)

Max. number of versions per object

1

If you want overwrite protection, increase the count to at least 2 versions per object. Version count includes live and noncurrent versions.

Expire noncurrent versions after

1

days

7 days recommended for Standard storage class

☐ **Retention policy (best for compliance)**

For preventing the deletion or modification of the bucket's objects for a specified minimum duration of time after being uploaded. [Learn more](#)

Click **CREATE**

Bucket is created for this experiment


datapipe-20220110-student10xin

cloud storage

←

Bucket details

datapipe-20220110-student10xin-ingestion-bucket

Location	Storage class	Public access	Protection
us-central1 (Iowa)	Standard	 Subject to object ACLs	Object versioning

OBJECTS

CONFIGURATION

PERMISSIONS

PROTECTION

LIFECYCLE

Buckets

datapipe-20220110-student10xin-ingestion-bucket

UPLOAD FILES

UPLOAD FOLDER

CREATE FOLDER

MANAGE HOLDS

DOWNLOAD

Setting Up Apache Airflow

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.

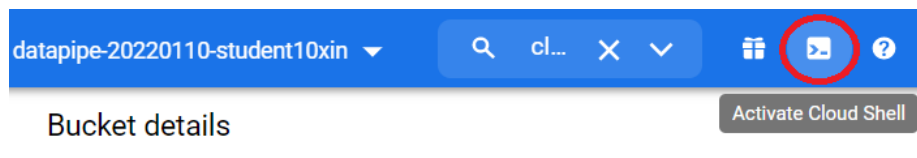
In Airflow, a **DAG** (Directed Acyclic Graph) is a collection of organized tasks that you want to schedule and run. DAGs, also called workflows, are defined in standard Python files. Cloud Composer only schedules the DAGs in the /dags folder. The /dags folder is in the Cloud Storage bucket that Cloud 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=serverless-composer-pubsub-environment
```

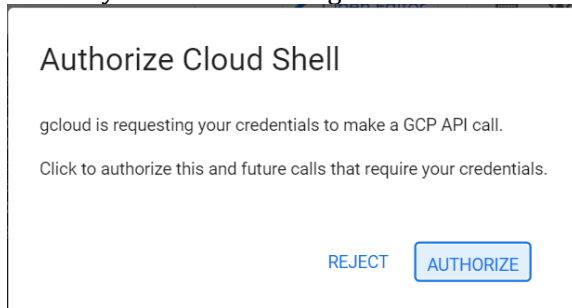
To set Airflow variables using the gcloud command-line tool, use the `gcloud composer environments run` command with the `variables` sub-command. This `gcloud composer` command executes the Airflow CLI sub-command [variables](#). The sub-command passes the arguments to the `gcloud` command line tool.

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



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-student10x/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 four variables you just set: `gcp_project`, `gcs_bucket`, `gce_region` and `gce_zone`.

Get the Airflow web server URL

This example makes REST API requests to the Airflow web server endpoint. You use the part of the Airflow web interface URL before `.appsspot.com` in your Cloud Function code.

1. In the Google Cloud Console, go to the **Environments** page.
[Go to Environments](#)
2. Click the name of your environment.
3. On the **Environment details** page, go to the **Environment details** tab.
4. The URL of the Airflow web server is listed in the **Airflow web UI** item.

Get the client_id of the IAM proxy

To make a request to the Airflow REST API endpoint, the function requires the client ID of the IAM proxy that protects the Airflow web server.

Cloud Composer does not provide this information directly. Instead, make an unauthenticated request to the Airflow web server and capture the client ID from the redirect URL:

To gather the client_id with curl

```
curl -v AIRFLOW_URL 2>&1 >/dev/null | grep -o "client_id\[A-Za-z0-9-]*\.apps\.googleusercontent\.com"
```

Replace `AIRFLOW_URL` with the URL of the Airflow web interface.

In the output, search for the string following `client_id`. For example:

```
client_id=836436932391-16q2c5f5dcsfne177va9bvf4j280t35c.apps.googleusercontent.com
```

To gather the client_id with Python

Save the following code in a file called `get_client_id.py`. Fill in your values for `project_id`, `location`, and `composer_environment`, then run the code in Cloud Shell or your local environment.

```
# This script is intended to be used with Composer 1 environments
# In Composer 2, the Airflow Webserver is not in the tenant project
# so there is no tenant client ID
# See https://cloud.google.com/composer/docs/composer-2/environment-architecture
# for more details
import google.auth
import google.auth.transport.requests
```

```

import requests
import six.moves.urllib.parse

# Authenticate with Google Cloud.
# See: https://cloud.google.com/docs/authentication/getting-started
credentials, _ = google.auth.default(
    scopes=["https://www.googleapis.com/auth/cloud-platform"]
)
authed_session = google.auth.transport.requests.AuthorizedSession(credentials)

# project_id = 'YOUR_PROJECT_ID'
# location = 'us-central1'
# composer_environment = 'YOUR_COMPOSER_ENVIRONMENT_NAME'

environment_url = (
    "https://composer.googleapis.com/v1beta1/projects/{}/locations/{}"
    "/environments/{}"
).format(project_id, location, composer_environment)
composer_response = authed_session.request("GET", environment_url)
environment_data = composer_response.json()
composer_version = environment_data["config"]["softwareConfig"]["imageVersion"]
if "composer-1" not in composer_version:
    version_error = ("This script is intended to be used with Composer 1 environments. "
        "In Composer 2, the Airflow Webserver is not in the tenant project, "
        "so there is no tenant client ID. "
        "See https://cloud.google.com/composer/docs/composer-2/environment-architecture for more "
        "details.")
    raise (RuntimeError(version_error))
airflow_uri = environment_data["config"]["airflowUri"]

# The Composer environment response does not include the IAP client ID.
# Make a second, unauthenticated HTTP request to the web server to get the
# redirect URI.
redirect_response = requests.get(airflow_uri, allow_redirects=False)
redirect_location = redirect_response.headers["location"]

# Extract the client_id query parameter from the redirect.
parsed = six.moves.urllib.parse.urlparse(redirect_location)
query_string = six.moves.urllib.parse.parse_qs(parsed.query)
print(query_string["client_id"][0])

```

Trigger a DAG from Cloud Functions

Upload a DAG to your environment

[Upload a DAG to your environment](#). The following example DAG outputs the received DAG run configuration. You trigger this DAG from a function, which you create later in this guide.

Save the following code to `trigger_faas.py`, and upload the file directly to the DAGs folder through the Cloud Composer service console. This code is also provided in the experiments folder of the GitHub repo.

```
import datetime

import airflow
from airflow.operators.bash_operator import BashOperator

with airflow.DAG(
    'composer_sample_trigger_response_dag',
    start_date=datetime.datetime(2021, 1, 1),
    # Not scheduled, trigger only
    schedule_interval=None) as dag:

    # Print the dag_run's configuration, which includes information about
    the
    # Cloud Storage object change.
    print_gcs_info = BashOperator(
        task_id='print_gcs_info', bash_command='echo {{ dag_run.conf }}')
```

Composer	Environments	+ CREATE	DELETE			
Filter environments						
Name ↑	Location	Composer version	Airflow version	Airflow webserver	Logs	DAGs folder
serverless-composer-pubsub-environment	us-central1	1.17.7	2.0.2	Airflow	Logs	DAGs

[←](#)
Bucket details

us-central1-serverless-comp-d1a6690f-bucket

Location

Storage class

Public access

Protection

us-central1 (Iowa)
Standard
⚠ Subject to object ACLs
None

OBJECTS

CONFIGURATION

PERMISSIONS



PROTECTION

LIFECYC

Buckets > us-central1-serverless-comp-d1a6690f-bucket > dags

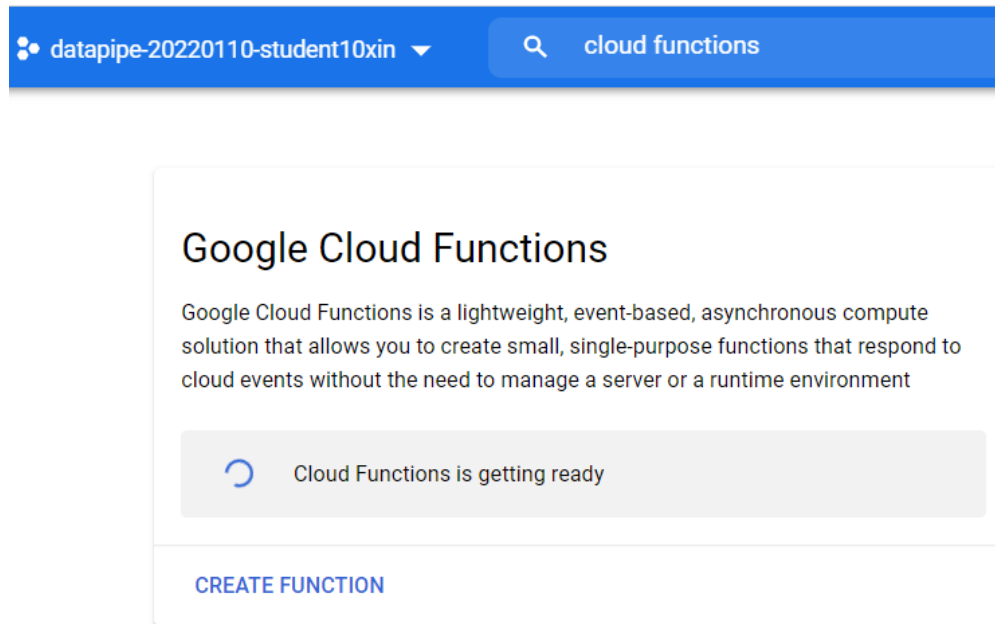
[UPLOAD FILES](#)
[UPLOAD FOLDER](#)
[CREATE FOLDER](#)
[MANAGE HOLDS](#)
[DOWNL](#)

Filter by name prefix only
Filter
Filter objects and folders

<input type="checkbox"/>	Name	Size	Type	Created ?
<input type="checkbox"/>	 airflow_monitoring.py	729 B	text/x-python	Jan 12, 20...
<input type="checkbox"/>	 trigger_faas.py	515 B	text/x-python	Jan 12, 20...

Deploy a Cloud Function that triggers the DAG

[Deploy a Python Cloud Function](#) using the following configuration parameters and content.




Specify Cloud Function configuration parameters

- **Trigger.** For this example, select a trigger that works when a new object is created in a bucket, or an existing object gets overwritten.
 - **Trigger Type.** Cloud Storage.
 - **Event Type.** [Finalize / Create](#).
 - **Bucket.** Select the bucket that must trigger this function. We created that at the beginning of this experiment.
 - **Retry on failure.** We recommend to disable this option for the purposes of this example. If you use your own function in a production environment, enable this option to [handle transient errors](#).

Function name *
trigger-dag ?

Region
us-central1 ▼ ?

Trigger

 **Cloud Storage**

Trigger type
Cloud Storage ▼

Event type *
Finalize/Create ▼ ?

Bucket *
datapipe-20220110-student10xin-ingestion-bucket BROWSE

☐ Retry on failure ?

SAVE CANCEL

Choose **SAVE**

Choose **NEXT**

Change the runtime to **Python 3.7** and replace the code in main.py and requirements.txt with the values and code from this experiment (remember main.py is in the GitHub repo)

Add requirements

Specify the dependencies in the requirements.txt file:

```
requests_toolbelt==0.9.1
google-auth==2.3.3
```

Add the Cloud Function code

Put the following code to the main.py file and make the following replacements:

- Replace the value of the client_id variable with the client_id value that you obtained earlier. Look for YOUR-CLIENT-ID
- Replace the value of the webserver_id variable with your tenant project ID, which is a part of the Airflow web interface URL before .appspot.com. You obtained the Airflow web interface URL earlier. Look for YOUR-TENANT-PROJECT
- Specify the Airflow REST API version that you use:
 - We use the stable Airflow REST API, so we have set the USE_EXPERIMENTAL_API variable to False.

The following is in the experiments folder of our GitHub repo as main.py

The screenshot shows the Google Cloud Functions console. At the top, there's a breadcrumb navigation: 'Cloud Functions' with a left arrow and 'Create function'. Below this, there are two tabs: 'Configuration' (checked) and 'Code' (selected with a blue circle and number 2). The 'Configuration' tab shows 'Runtime' set to 'Python 3.7' and 'Entry point' set to 'trigger_dag'. The 'Code' tab shows the source code editor with a file list on the left containing 'main.py' and 'requirements.txt'. The code in the editor is as follows:

```
Press Alt+F1 for Accessibility Options.
1 from google.auth.transport.requests import request
2 from google.oauth2 import id_token
3 import requests
4
5
6 IAM_SCOPE = 'https://www.googleapis.com/auth/cloud-platform'
7 OAUTH_TOKEN_URI = 'https://www.googleapis.com/oauth2/v3/token'
8 # If you are using the stable API, set this value to False
9 # For more info about Airflow APIs see https://airflow.apache.org/docs/apache-airflow/1.10.0/api.html
10 USE_EXPERIMENTAL_API = True
11
12
13 def trigger_dag(data, context=None):
14     """Makes a POST request to the Composer DAG
15
16     When called via Google Cloud Functions (GCF)
17     data and context are Background function parameters
18 """
```

Ensure you have changed the code to add your Airflow project and client ID that we gathered earlier.


```

from google.auth.transport.requests import Request
from google.oauth2 import id_token
import requests

IAM_SCOPE = 'https://www.googleapis.com/auth/iam'
OAUTH_TOKEN_URI = 'https://www.googleapis.com/oauth2/v4/token'
# If you are using the stable API, set this value to False
# For more info about Airflow APIs see
https://cloud.google.com/composer/docs/access-airflow-api
USE_EXPERIMENTAL_API = False

def trigger_dag(data, context=None):
    """Makes a POST request to the Composer DAG Trigger API

    When called via Google Cloud Functions (GCF),
    data and context are Background function parameters.

    For more info, refer to
    https://cloud.google.com/functions/docs/writing/background#functions\_b
ackground\_parameters-python

    To call this function from a Python script, omit the ``context``
    argument
    and pass in a non-null value for the ``data`` argument.

    This function is currently only compatible with Composer v1
    environments.
    """

    # Fill in with your Composer info here
    # Navigate to your webserver's login page and get this from the URL
    # Or use the script found at
    # https://github.com/GoogleCloudPlatform/python-docs-
samples/blob/master/composer/rest/get\_client\_id.py
    client_id = 'YOUR-CLIENT-ID'
    # This should be part of your webserver's URL:
    # {tenant-project-id}.appspot.com
    webserver_id = 'YOUR-TENANT-PROJECT'
    # The name of the DAG you wish to trigger
    dag_name = 'composer_sample_trigger_response_dag'

```

```

if USE_EXPERIMENTAL_API:
    endpoint = f'api/experimental/dags/{dag_name}/dag_runs'
    json_data = {'conf': data, 'replace_microseconds': 'false'}
else:
    endpoint = f'api/v1/dags/{dag_name}/dagRuns'
    json_data = {'conf': data}
webserver_url = (
    'https://'
    + webserver_id
    + '.appspot.com/'
    + endpoint
)
# Make a POST request to IAP which then Triggers the DAG
make_iap_request(
    webserver_url, client_id, method='POST', json=json_data)

# This code is copied from
# https://github.com/GoogleCloudPlatform/python-docs-
# samples/blob/master/iap/make_iap_request.py
# START COPIED IAP CODE
def make_iap_request(url, client_id, method='GET', **kwargs):
    """Makes a request to an application protected by Identity-Aware
    Proxy.

    Args:
        url: The Identity-Aware Proxy-protected URL to fetch.
        client_id: The client ID used by Identity-Aware Proxy.
        method: The request method to use
            ('GET', 'OPTIONS', 'HEAD', 'POST', 'PUT', 'PATCH', 'DELETE')
        **kwargs: Any of the parameters defined for the request function:
            https://github.com/requests/requests/blob/master/requests/
api.py

            If no timeout is provided, it is set to 90 by default.

    Returns:
        The page body, or raises an exception if the page couldn't be
        retrieved.
    """
    # Set the default timeout, if missing
    if 'timeout' not in kwargs:
        kwargs['timeout'] = 90

    # Obtain an OpenID Connect (OIDC) token from metadata server or using
    service

```

```

# account.
google_open_id_connect_token = id_token.fetch_id_token(Request(),
client_id)

# Fetch the Identity-Aware Proxy-protected URL, including an
# Authorization header containing "Bearer " followed by a
# Google-issued OpenID Connect token for the service account.
resp = requests.request(
    method, url,
    headers={'Authorization': 'Bearer {}'.format(
        google_open_id_connect_token)}, **kwargs)
if resp.status_code == 403:
    raise Exception('Service account does not have permission to '
                    'access the IAP-protected application.')
elif resp.status_code != 200:
    raise Exception(
        'Bad response from application: {!r} / {!r} / {!r}'.format(
            resp.status_code, resp.headers, resp.text))
else:
    return resp.text
# END COPIED IAP CODE

```

Test your function

To check that your function and DAG work as intended:

1. Wait until your function deploys.
2. Upload a file to your Cloud Storage bucket. As an alternative, you can trigger the function manually by selecting the **Test function** action for it in Google Cloud Console.
3. Check the DAG page [in the Airflow web interface](#) from the Environments console view for your Cloud Composer instance. The DAG should have one active or already completed DAG run.
4. In the Airflow UI, check task logs for this run. You should see that the `print_gcs_info` task outputs the data received from the function to the logs:

```

[2021-04-04 18:25:44,778] {bash_operator.py:154} INFO - Output:
[2021-04-04 18:25:44,781] {bash_operator.py:158} INFO - Triggered from
GCF:

```

```
{bucket: example-storage-for-gcf-triggers, contentType: text/plain,  
  crc32c: dldNmg==, etag: COW+26Sb5e8CEAE=, generation:  
1617560727904101,  
  ... }  
[2021-04-04 18:25:44,781] {bash_operator.py:162} INFO - Command exited  
with  
  return code 0h
```

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.
5. Delete the Cloud Function.

You've now learned to do some cool stuff with Cloud Composer, Cloud Functions and Pub/Sub.

Congratulations!

In this experiment, you created and ran a serverless data ingestion pipeline triggering Pub/Sub with FaaS and Apache Airflow.