Setting up a CI/CD pipeline for your data-processing workflow

This experiment describes how to set up a continuous integration/continuous deployment (CI/CD) pipeline for processing data by implementing CI/CD methods with managed products on Google Cloud. Data scientists and analysts can adapt the methodologies from CI/CD practices to help to ensure high quality, maintainability, and adaptability of the data processes and workflows. The methods that you can apply are as follows:

- Version control of source code.
- Automatic building, testing, and deployment of apps.
- Environment isolation and separation from production.
- Replicable procedures for environment setup.

This experiment is intended for data scientists and analysts who build recurrent running data-processing jobs to help structure their research and development (R&D) to systematically and automatically maintain data-processing workloads.

Deployment architecture

In this guide, you use the following Google Cloud products:

- Cloud Build to create a CI/CD pipeline for building, deploying, and testing a data-processing workflow, and the data processing itself. Cloud Build is a managed service that runs your build on Google Cloud. A build is a series of build steps where each step is run in a Docker container.
- Cloud Composer to define and run the steps of the workflow, such as starting the data processing, testing and verifying results. Cloud Composer is a managed Apache Airflow service, which offers an environment where you can create, schedule, monitor, and manage complex workflows, such as the data-processing workflow in this experiment.

- Dataflow to run the Apache Beam WordCount example as a sample data process.

The CI/CD pipeline

At a high level, the CI/CD pipeline consists of the following steps:

1. Cloud Build packages the WordCount sample into a self-running Java Archive (JAR) file using the Maven builder. The Maven builder is a container with Maven installed in it. When a build step is configured to use the Maven builder, Maven runs the tasks.
2. Cloud Build uploads the JAR file to Cloud Storage.
3. Cloud Build runs unit tests on the data-processing workflow code and deploys the workflow code to Cloud Composer.
4. Cloud Composer picks up the JAR file and runs the data-processing job on Dataflow.

The following diagram shows a detailed view of the CI/CD pipeline steps.

In this experiment, the deployments to the test and production environments are separated into two different Cloud Build pipelines—a test and a production pipeline.

In the preceding diagram, the test pipeline consists of the following steps:

1. A developer commits code changes to the Cloud Source Repositories.
2. Code changes trigger a test build in Cloud Build.
3. Cloud Build builds the self-executing JAR file and deploys it to the test JAR bucket on Cloud Storage.
4. Cloud Build deploys the test files to the test-file buckets on Cloud Storage.
5. Cloud Build sets the variable in Cloud Composer to reference the newly deployed JAR file.
6. Cloud Build tests the data-processing workflow Directed Acyclic Graph (DAG) and deploys it to the Cloud Composer bucket on Cloud Storage.
7. The workflow DAG file is deployed to Cloud Composer.
8. Cloud Build triggers the newly deployed data-processing workflow to run.

In the preceding diagram, the production pipeline consists of the following steps:

1. A developer manually runs the production deployment pipeline in Cloud Build.

2. Cloud Build copies the latest self-executing JAR file from the test JAR bucket to the production JAR bucket on Cloud Storage.
3. Cloud Build tests the production data-processing workflow DAG and deploys it to the Cloud Composer bucket on Cloud Storage.
4. The production workflow DAG file is deployed to Cloud Composer.

In this experiment, the production data-processing workflow is deployed to the same Cloud Composer environment as the test workflow, to give a consolidated view of all data-processing workflows. For the purposes of this experiment, the environments are separated by using different Cloud Storage buckets to hold the input and output data.

To completely separate the environments, you need multiple Cloud Composer environments created in different projects, which are by default separated from each other. This separation helps to secure your production environment. This approach is outside the scope of this experiment. For more information about how to access resources across multiple Google Cloud projects, see Setting service account permissions.

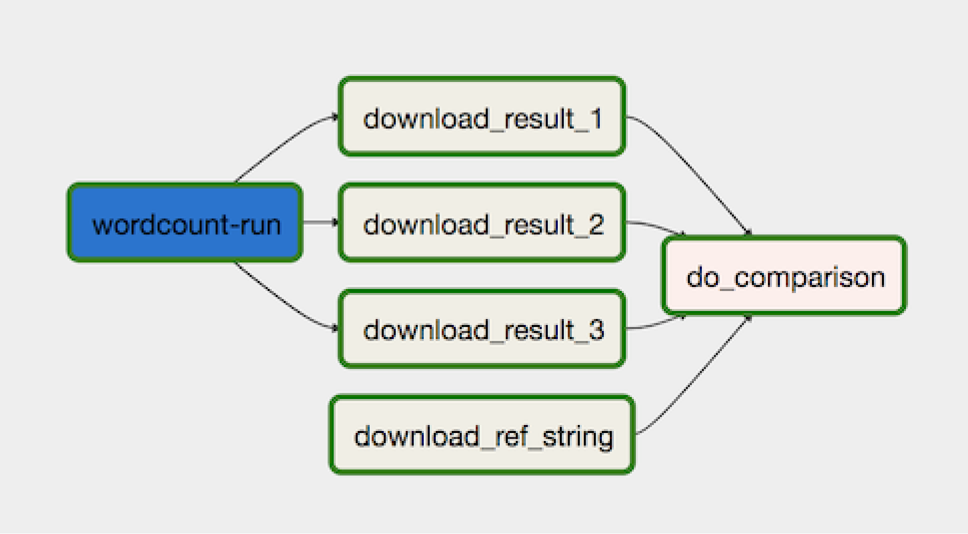
The data-processing workflow

The instructions for how Cloud Composer runs the data-processing workflow are defined in a Directed Acyclic Graph (DAG) written in Python. In the DAG, all the steps of the data-processing workflow are defined together with the dependencies between them.

The CI/CD pipeline automatically deploys the DAG definition from Cloud Source Repositories to Cloud Composer in each build. This process ensures that Cloud Composer is always up to date with the latest workflow definition without needing any human intervention.

In the DAG definition for the test environment, an end-to-end test step is defined in addition to the data-processing workflow. The test step helps make sure that the data-processing workflow runs correctly.

The data-processing workflow is illustrated in the following diagram.



The data-processing workflow consists of the following steps:

1. Run the WordCount data process in Dataflow.
2. Download the output files from the WordCount process. The WordCount process outputs three files:
   - download_result_1
   - download_result_2
   - download_result_3
3. Download the reference file, called download_ref_string.
4. Verify the result against the reference file. This integration test aggregates all three results and compares the aggregated results with the reference file.

Using a task-orchestration framework such as Cloud Composer to manage the data-processing workflow helps alleviate the code complexity of the workflow.

The tests

In addition to the integration test that verifies the data-processing workflow from end to end, there are two unit tests in this experiment. The unit tests are automatic tests on the data-processing code and the data-processing workflow code. The test on the data-processing code is written in Java and runs automatically during the Maven build process. The test on the data-processing workflow code is written in Python and runs as an independent build step.

Objectives

- Configure the Cloud Composer environment.

- Create Cloud Storage buckets for your data.
- Create the build, test, and production pipelines.
- Configure the build trigger.

Costs

This experiment uses the following billable components of Google Cloud:

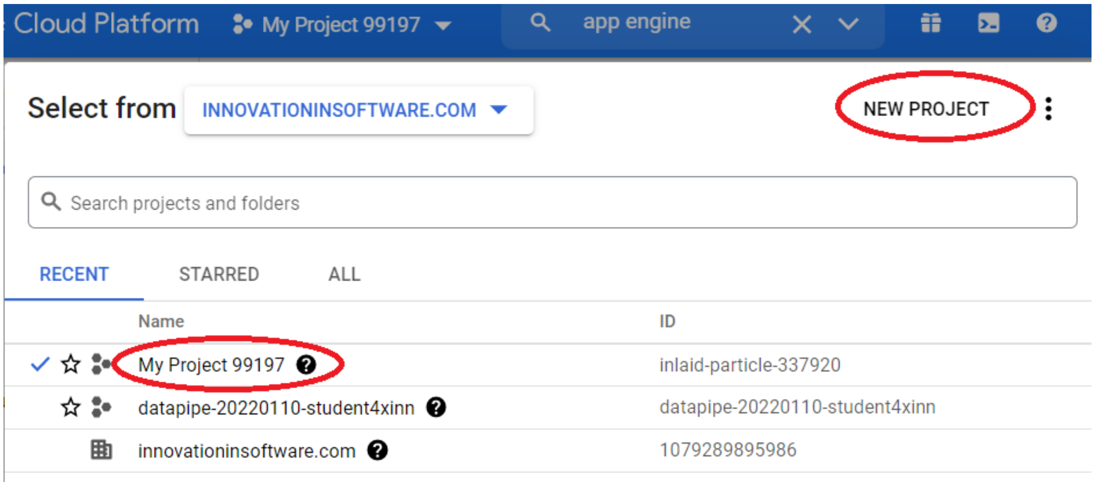
- Cloud Source Repositories
- Cloud Build
- Cloud Composer
- Dataflow
- Cloud Storage

To generate a cost estimate based on your projected usage, use the pricing calculator.

When you finish this experiment, you can avoid continued billing by deleting the resources you created. For more information, see Clean up.

API Enablement

1. For only this experiment we will use a different Project ID, My Project 2342334 or similar in your accounts. If it does not already exist then please create it as a New Project and use that new account. You can verify this will resolve the issue with our datapipe accounts by loading the App Engine service console



Name	ID
✓ ☆ My Project 99197	inlaid-particle-337920
☆ datapipe-20220110-student4xinn	datapipe-20220110-student4xinn
innovationinsoftware.com	1079289895986

2. Enable the Cloud Build, Cloud Source Repositories, Cloud Composer, and Dataflow APIs.

   Enable the APIs

Sample code

The sample code is in two folders:

- The env-setup folder contains shell scripts for the initial setup of the Google Cloud environment.
- The source-code folder contains code that is developed over time, needs to be source controlled, and triggers automatic build and test processes. This folder contains the following subfolders:
  - The data-processing-code folder contains the Apache Beam process source code.
  - The workflow-dag folder contains the composer DAG definitions for the data-processing workflows with the steps to design, implement, and test the Dataflow process.
  - The build-pipeline folder contains two Cloud Build configurations—one for the test pipeline and the other for the production pipeline. This folder also contains a support script for the pipelines.

For the purpose of this experiment, the source code files for data processing and for DAG workflow are in different folders in the same source code repository. In a production environment, the source code files are usually in their own source code repositories and are managed by different teams.

Setting up your environment

In this experiment, you run all commands in Cloud Shell. Cloud Shell appears as a window at the bottom of the Google Cloud Console.

1. In the Cloud Console, open Cloud Shell:

   Open Cloud Shell

2. Clone the sample code repository:

   ```
   git clone https://github.com/GoogleCloudPlatform/ci-cd-for-data-processing-workflow.git
   ```

3. Run a script to set environment variables:

   ```
   cd ~/ci-cd-for-data-processing-workflow/env-setup
   source set_env.sh
   ```

CRITICAL EXPERIMENT NOTE: If you walk away from the shell for an extended period of time, and it expires you will have to rerun the scripts that set the environment variables. That could include

```
$ cd ~/ci-cd-for-data-processing-workflow/env-setup
$ source set_env.sh
$ set_composer_variables.sh
$ export COMPOSER_DAG_BUCKET=$(gcloud composer environments describe $COMPOSER_ENV_NAME \
   --location $COMPOSER_REGION \
   --format="get(config.dagGcsPrefix)")
$ export COMPOSER_SERVICE_ACCOUNT=$(gcloud composer environments describe $COMPOSER_ENV_NAME \
   --location $COMPOSER_REGION \
   --format="get(config.nodeConfig.serviceAccount)")
$ gcloud config set account student10@innovationinsoftware.com
$ gcloud auth configure-docker
$ git config --global user.email "student10@innovationinsoftware.com"
$ git config --global user.name "George Niece"
```

The script sets the following environment variables:

- Your Google Cloud project ID
- Your region and zone
- The name of your Cloud Storage buckets that are used by the build pipeline and the data-processing workflow.

Because environment variables aren't retained between sessions, if your Cloud Shell session shuts down or disconnects while you are working through the experiment, you need to reset the environment variables.

Creating the Cloud Composer environment

In this experiment, you set up a Cloud Composer environment that consists of `n1-standard-1` nodes.

1. In Cloud Shell, create the Cloud Composer environment:

   ```
   gcloud composer environments create $COMPOSER_ENV_NAME \
       --location $COMPOSER_REGION \
       --zone $COMPOSER_ZONE_ID \
       --machine-type n1-standard-1 \
       --node-count 3 \
       --disk-size 20 \
       --python-version 2
   ```

   Note: It usually takes about 15 minutes to provision the Cloud Composer environment, but it can take up to one hour. Wait until this process is completed before continuing onto the next steps.
2. Run a script to set the variables in the Cloud Composer environment. The variables are needed for the data-processing DAGs.

   ```
   cd ~/ci-cd-for-data-processing-workflow/env-setup
   chmod +x set_composer_variables.sh
   ./set_composer_variables.sh
   ```

   The script sets the following environment variables:

   - Your Google Cloud project ID
   - Your region and zone
   - The name of your Cloud Storage buckets that are used by the build pipeline and the data-processing workflow.

Extract the Cloud Composer environment properties

Cloud Composer uses a Cloud Storage bucket to store DAGs. Moving a DAG definition file to the bucket triggers Cloud Composer to automatically read the files. You created the Cloud Storage bucket for Cloud Composer when you created the Cloud Composer environment. In the following procedure, you extract the URL for the buckets, and then configure your CI/CD pipeline to automatically deploy DAG definitions to the Cloud Storage bucket.

1. In Cloud Shell, export the URL for the bucket as an environment variable:

   ```
   export COMPOSER_DAG_BUCKET=$(gcloud composer environments describe
   $COMPOSER_ENV_NAME \
       --location $COMPOSER_REGION \
       --format="get(config.dagGcsPrefix)")
   ```

2. Export the name of the service account that Cloud Composer uses in order to have access to the Cloud Storage buckets:

   ```
   export COMPOSER_SERVICE_ACCOUNT=$(gcloud composer environments
   describe $COMPOSER_ENV_NAME \
       --location $COMPOSER_REGION \
       --format="get(config.nodeConfig.serviceAccount)")
   ```

Creating the Cloud Storage buckets

In this section you create a set of Cloud Storage buckets to store the following:

- Artifacts of the intermediate steps of the build process.
- The input and output files for the data-processing workflow.
- The staging location for the Dataflow jobs to store their binary files.

To create the Cloud Storage buckets, complete the following step:

- In Cloud Shell, create Cloud Storage buckets and give the Cloud Composer service account permission to run the data-processing workflows:

  ```
  cd ~/ci-cd-for-data-processing-workflow/env-setup
  chmod +x create_buckets.sh
  ./create_buckets.sh
  ```

Pushing the source code to Cloud Source Repositories

In this experiment, you have one source code base that you need to put into version control. The following step shows how a code base is developed and changes over time. Whenever changes are pushed to the repository, the pipeline to build, deploy, and test is triggered.

- In Cloud Shell, push the `source-code` folder to Cloud Source Repositories:

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

gcloud auth configure-docker
git config --global user.email "student10@innovationinsoftware.com"
git config --global user.name "George Niece"
gcloud source repos create $SOURCE_CODE_REPO
cp -r ~/ci-cd-for-data-processing-workflow/source-code
~/$SOURCE_CODE_REPO
cd ~/$SOURCE_CODE_REPO
git init
git remote add google \
    https://source.developers.google.com/p/$GCP_PROJECT_ID/r/$SOURCE
_CODE_REPO

git add .
git commit -m 'initial commit'
git push google master
```

These are standard commands to initialize Git in a new directory and push the content to a remote repository.

Creating Cloud Build pipelines

In this section, you create the build pipelines that build, deploy, and test the data-processing workflow.

Grant access to Cloud Build service account

Cloud Build deploys Cloud Composer DAGs and triggers workflows, which are enabled when you add additional access to the Cloud Build service account. For more information about the different roles available when working with Cloud Composer, see the access control documentation.

1. In Cloud Shell, add the composer.admin role to the Cloud Build service account so the Cloud Build job can set Airflow variables in Cloud Composer:

   ```
   gcloud projects add-iam-policy-binding $GCP_PROJECT_ID \
       --
   member=serviceAccount:$PROJECT_NUMBER@cloudbuild.gserviceaccount.com
   \
       --role=roles/composer.admin
   ```

2. Add the composer.worker role to the Cloud Build service account so the Cloud Build job can trigger the data workflow in Cloud Composer:

   ```
   gcloud projects add-iam-policy-binding $GCP_PROJECT_ID \
       --
   member=serviceAccount:$PROJECT_NUMBER@cloudbuild.gserviceaccount.com
   \
       --role=roles/composer.worker
   ```

Create the build and test pipeline

The build and test pipeline steps are configured in the YAML configuration file. In this experiment, you use prebuilt builder images for git, maven, gsutil, and gcloud to run the tasks in each build step. You use configuration variable substitutions to define the environment settings at build time. The source code repository location is defined by variable substitutions, as well as the locations of Cloud Storage buckets. The build needs this information to deploy the JAR file, test files, and the DAG definition.

- In Cloud Shell, submit the build pipeline configuration file to create the pipeline in Cloud Build:

  ```
  cd ~/ci-cd-for-data-processing-workflow/source-code/build-pipeline
  ```

  vi or nano build_deploy_test.yaml

  remove the following section, and this step must be revised for the later build options when you manual trigger production.

  ```
  - name: ' gcr.io/cloud-solutions-images/apache-airflow:1.10'
    entrypoint: 'python'
  ```

```
 args: ['test_compare_xcom_maps.py']
 dir: '$REPO_NAME/workflow-dag'
 id: 'unit-test-on-operator-code'
```

Execute the build for test automation

```
gcloud builds submit --config=build_deploy_test.yaml --
substitutions=\
REPO_NAME=$SOURCE_CODE_REPO,\
_DATAFLOW_JAR_BUCKET=$DATAFLOW_JAR_BUCKET_TEST,\
_COMPOSER_INPUT_BUCKET=$INPUT_BUCKET_TEST,\
_COMPOSER_REF_BUCKET=$REF_BUCKET_TEST,\
_COMPOSER_DAG_BUCKET=$COMPOSER_DAG_BUCKET,\
_COMPOSER_ENV_NAME=$COMPOSER_ENV_NAME,\
_COMPOSER_REGION=$COMPOSER_REGION,\
_COMPOSER_DAG_NAME_TEST=$COMPOSER_DAG_NAME_TEST
```

This command instructs Cloud Build to run a build with the following steps:

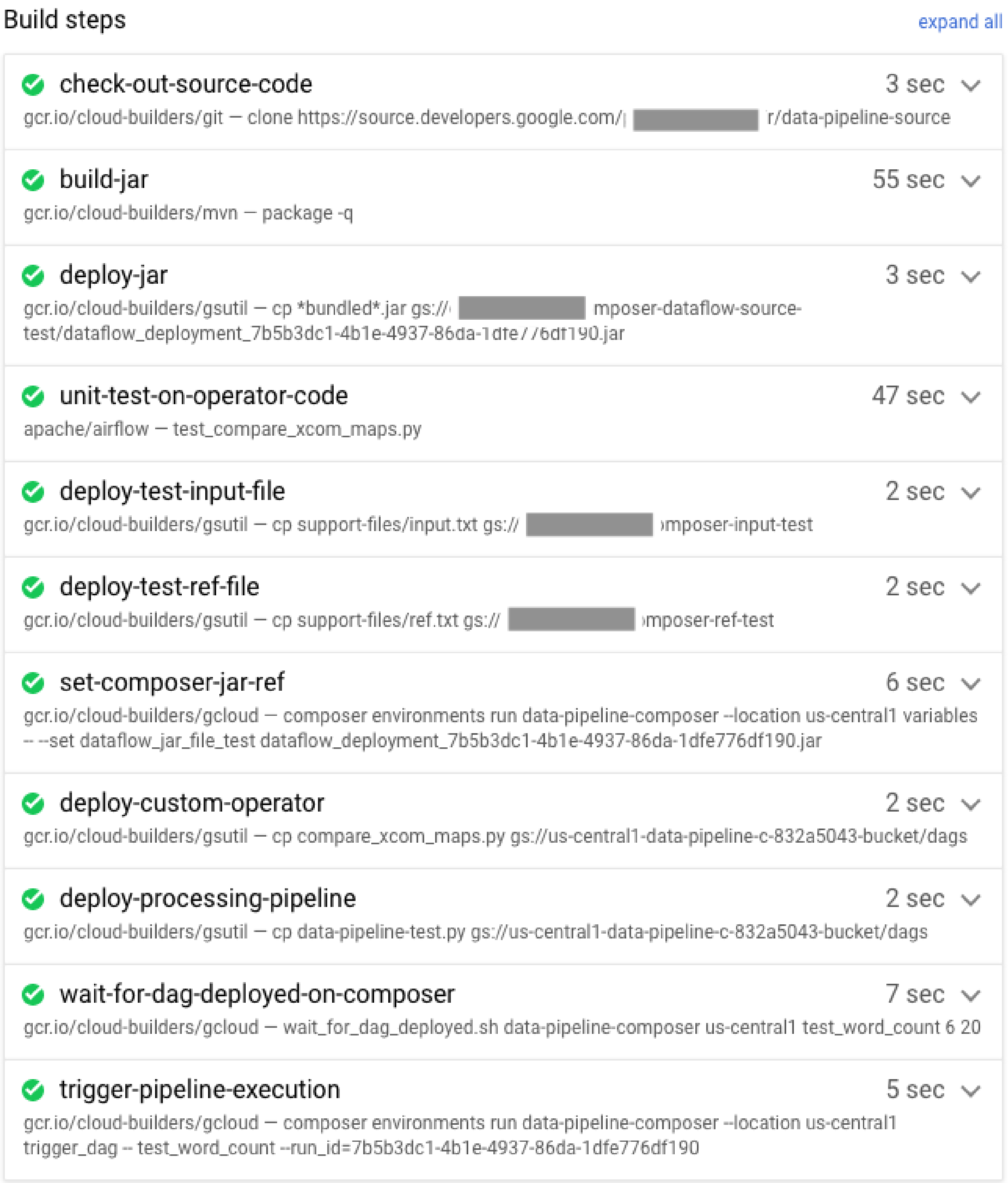
1. Build and deploy the WordCount self-executing JAR file.
    a. Check out the source code.
    b. Compile the WordCount Beam source code into a self-executing JAR file.
    c. Store the JAR file on Cloud Storage where it can be picked up by Cloud Composer to run the WordCount processing job.
2. Deploy and set up the data-processing workflow on Cloud Composer.
    a. Run the unit test on the custom-operator code used by the workflow DAG.
    b. Deploy the test input file and the test reference file on Cloud Storage. The test input file is the input for the WordCount processing job. The test reference file is used as a reference to verify the output of the WordCount processing job.
    c. Set the Cloud Composer variables to point to the newly built JAR file.
    d. Deploy the workflow DAG definition to the Cloud Composer environment.
3. Run the data-processing workflow in the test environment to trigger the test-processing workflow.

Verify the build and test pipeline

After you submit the build file, verify the build steps.

1. In the Cloud Console, go to the **Build History** page to view a list of all past and currently running builds.

   Go to Build History page
2. Click the build that is currently running.
3. On the **Build details** page, verify that the build steps match the previously described steps.



✓ check-out-source-code 3 sec
gcr.io/cloud-builders/git — clone https://source.developers.google.com/[redacted]r/data-pipeline-source

✓ build-jar 55 sec
gcr.io/cloud-builders/mvn — package -q

✓ deploy-jar 3 sec
gcr.io/cloud-builders/gsutil — cp *bundled*.jar gs://[redacted]mposer-dataflow-source-test/dataflow_deployment_7b5b3dc1-4b1e-4937-86da-1dfe776df190.jar

✓ unit-test-on-operator-code 47 sec
apache/airflow — test_compare_xcom_maps.py

✓ deploy-test-input-file 2 sec
gcr.io/cloud-builders/gsutil — cp support-files/input.txt gs://[redacted]mposer-input-test

✓ deploy-test-ref-file 2 sec
gcr.io/cloud-builders/gsutil — cp support-files/ref.txt gs://[redacted]mposer-ref-test

✓ set-composer-jar-ref 6 sec
gcr.io/cloud-builders/gcloud — composer environments run data-pipeline-composer --location us-central1 variables -- --set dataflow_jar_file_test dataflow_deployment_7b5b3dc1-4b1e-4937-86da-1dfe776df190.jar

✓ deploy-custom-operator 2 sec
gcr.io/cloud-builders/gsutil — cp compare_xcom_maps.py gs://us-central1-data-pipeline-c-832a5043-bucket/dags

✓ deploy-processing-pipeline 2 sec
gcr.io/cloud-builders/gsutil — cp data-pipeline-test.py gs://us-central1-data-pipeline-c-832a5043-bucket/dags

✓ wait-for-dag-deployed-on-composer 7 sec
gcr.io/cloud-builders/gcloud — wait_for_dag_deployed.sh data-pipeline-composer us-central1 test_word_count 6 20

✓ trigger-pipeline-execution 5 sec
gcr.io/cloud-builders/gcloud — composer environments run data-pipeline-composer --location us-central1 trigger_dag -- test_word_count --run_id=7b5b3dc1-4b1e-4937-86da-1dfe776df190

On the **Build details** page, the **Status** field of the build says Build successful when the build finishes.

4. In Cloud Shell, verify that the WordCount sample JAR file was copied into the correct bucket:

```
gsutil ls gs://$DATAFLOW_JAR_BUCKET_TEST/dataflow_deployment*.jar
```

The output is similar to the following:

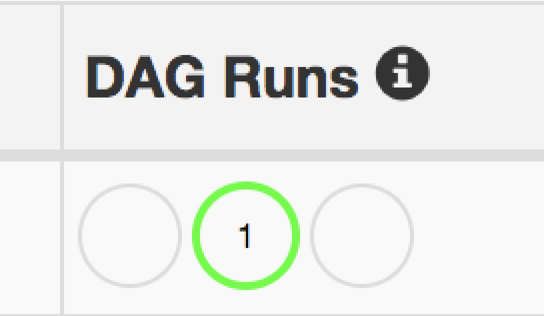
```
gs://...-composer-dataflow-source-test/dataflow_deployment_e88be61e-50a6-4aa0-beac-38d75871757e.jar
```

5. Get the URL to your Cloud Composer web interface. Make a note of the URL because it's used in the next step.

```
gcloud composer environments describe $COMPOSER_ENV_NAME \
    --location $COMPOSER_REGION \
    --format="get(config.airflowUri)"
```

6. Use the URL from the previous step to go to the Cloud Composer UI to verify a successful DAG run. If the **Dag Runs** column doesn't display any information, wait a few minutes and reload the page.

   a. To verify that the data-processing workflow DAG test_word_count is deployed and is in running mode, hold the pointer over the light-green circle below **DAG Runs** and verify that it says **Running**.



   b. To see the running data-processing workflow as a graph, click the light-green circle, and then on the **Dag Runs** page, click **Dag Id:** **`test_word_count`**.
   c. Reload the **Graph View** page to update the state of the current DAG run. It usually takes between three to five minutes for the workflow to finish. To verify that the DAG run finishes successfully, hold the pointer over each task to verify that the tooltip says **State: success**. The last task, named do_comparison, is the integration test that verifies the process output against the reference file.

Create the production pipeline

When the test processing workflow runs successfully, you can promote the current version of the workflow to production. There are several ways to deploy the workflow to production:

- Manually.
- Automatically triggered when all the tests pass in the test or staging environments.
- Automatically triggered by a scheduled job.

The automatic approaches are beyond the scope of this experiment. For more information, see Release Engineering.

In this experiment, you do a manual deployment to production by running the Cloud Build production deployment build. The production deployment build follows these steps:

1. Copy the WordCount JAR file from the test bucket to the production bucket.
2. Set the Cloud Composer variables for the production workflow to point to the newly promoted JAR file.
3. Deploy the production workflow DAG definition on the Cloud Composer environment and running the workflow.

Variable substitutions define the name of the latest JAR file that is deployed to production with the Cloud Storage buckets used by the production processing workflow. To create the Cloud Build pipeline that deploys the production airflow workflow, complete the following steps:

1. In Cloud Shell, read the filename of the latest JAR file by printing the Cloud Composer variable for the JAR filename:

```
export DATAFLOW_JAR_FILE_LATEST=$(gcloud composer environments run
$COMPOSER_ENV_NAME \
    --location $COMPOSER_REGION variables -- \
    --get dataflow_jar_file_test 2>&1 | grep -i '.jar')
```

2. Use the build pipeline configuration file, deploy_prod.yaml, to create the pipeline in Cloud Build:

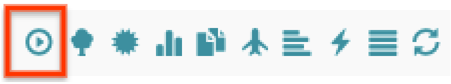
```
cd ~/ci-cd-for-data-processing-workflow/source-code/build-pipeline
gcloud builds submit --config=deploy_prod.yaml --substitutions=\
REPO_NAME=$SOURCE_CODE_REPO,\
_DATAFLOW_JAR_BUCKET_TEST=$DATAFLOW_JAR_BUCKET_TEST,\
_DATAFLOW_JAR_FILE_LATEST=$DATAFLOW_JAR_FILE_LATEST,\
_DATAFLOW_JAR_BUCKET_PROD=$DATAFLOW_JAR_BUCKET_PROD,\
_COMPOSER_INPUT_BUCKET=$INPUT_BUCKET_PROD,\
_COMPOSER_ENV_NAME=$COMPOSER_ENV_NAME,\
_COMPOSER_REGION=$COMPOSER_REGION,\
_COMPOSER_DAG_BUCKET=$COMPOSER_DAG_BUCKET,\
_COMPOSER_DAG_NAME_PROD=$COMPOSER_DAG_NAME_PROD
```

Verify the data-processing workflow created by the production pipeline

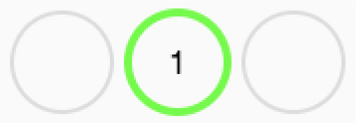
1. Get the URL for your Cloud Composer UI:

```
gcloud composer environments describe $COMPOSER_ENV_NAME \
    --location $COMPOSER_REGION \
    --format="get(config.airflowUri)"
```

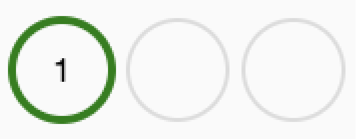
2. To verify that the production data-processing workflow DAG is deployed, go to the URL that you retrieved in the previous step and verify that prod_word_count DAG is in the list of DAGs.
    a. On the **DAGs** page, in the `prod_word_count` row, click **Trigger Dag**.



    b. In the **Confirmation** dialog, click **Confirm**.
3. Reload the page to update the DAG run status. To verify that the production data-processing workflow DAG is deployed and is in running mode, hold the pointer over the light-green circle below **DAG Runs** and verify that it says **Running**.



4. After the run succeeds, hold the pointer over the dark-green circle below the **DAG runs** column and verify that it says **Success**.



5. In Cloud Shell, list the result files in the Cloud Storage bucket:

```
gsutil ls gs://$RESULT_BUCKET_PROD
```

The output is similar to the following:

```
gs://...-composer-result-prod/output-00000-of-00003
gs://...-composer-result-prod/output-00001-of-00003
gs://...-composer-result-prod/output-00002-of-00003
```

Note: Typically, the production data workflow job execution is either triggered by events, such as files being stored in buckets, or is scheduled to run on a regular basis. It's important that the deployment job ensures that production data workflow isn't currently running before you deploy. In a production environment, you can use `dag_state` of the Airflow CLI commands to retrieve the status of a DAG run.

Configuring a build trigger

You set up a Cloud Build trigger that triggers a new build when changes are pushed to the master branch of the source repository.

1. In Cloud Shell, run the following command to get all the substitution variables needed for the build. Make a note of these values because they are needed in a later step.

   ```
   echo "_DATAFLOW_JAR_BUCKET : ${DATAFLOW_JAR_BUCKET_TEST}
   _COMPOSER_INPUT_BUCKET : ${INPUT_BUCKET_TEST}
   _COMPOSER_REF_BUCKET : ${REF_BUCKET_TEST}
   _COMPOSER_DAG_BUCKET : ${COMPOSER_DAG_BUCKET}
   _COMPOSER_ENV_NAME : ${COMPOSER_ENV_NAME}
   _COMPOSER_REGION : ${COMPOSER_REGION}
   _COMPOSER_DAG_NAME_TEST : ${COMPOSER_DAG_NAME_TEST}"
   ```

2. In the Cloud Console, go to the **Build triggers** page.

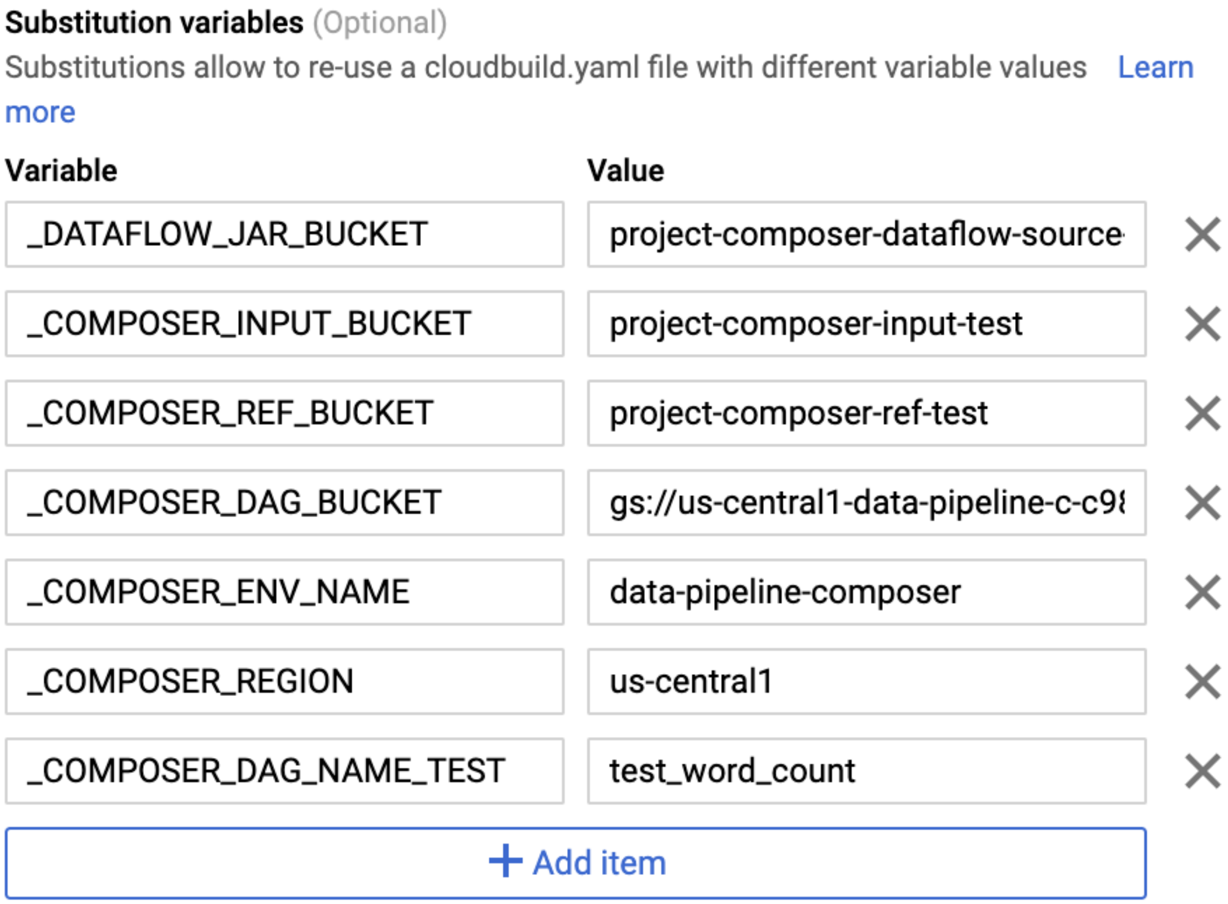
   Go to Build Triggers page
3. Click **Create trigger**.
4. Click **Cloud Source Repository**, and then click **Continue**.
5. Click **`data-pipeline-source`**, and then click **Continue**.
6. To configure trigger settings, complete the following steps:

   - In the **Name** field, enter trigger-build-in-test-environment.
   - For **Trigger type**, click **Branch**.
   - In the **Branch (regex)** field, enter master.
   - For **Configuration**, click **Cloud Build configuration file (yaml or json)**.
   - In the **Cloud Build configuration file location** field, enter build-pipeline/build_deploy_test.yaml.

7. On the **Trigger settings** page, replace the variables with values from your environment that you got from the earlier step. Add the following one at a time and click **+ Add item** for each of the name-value pairs.

   - _DATAFLOW_JAR_BUCKET
   - _COMPOSER_INPUT_BUCKET

- _COMPOSER_REF_BUCKET
- _COMPOSER_DAG_BUCKET
- _COMPOSER_ENV_NAME
- _COMPOSER_REGION
- _COMPOSER_DAG_NAME_TEST



Substitution variables (Optional)

Substitutions allow to re-use a cloudbuild.yaml file with different variable values Learn more

Variable	Value
_DATAFLOW_JAR_BUCKET	project-composer-dataflow-source
_COMPOSER_INPUT_BUCKET	project-composer-input-test
_COMPOSER_REF_BUCKET	project-composer-ref-test
_COMPOSER_DAG_BUCKET	gs://us-central1-data-pipeline-c-c9
_COMPOSER_ENV_NAME	data-pipeline-composer
_COMPOSER_REGION	us-central1
_COMPOSER_DAG_NAME_TEST	test_word_count

8. Click **Create trigger**.

Test the trigger

To test the trigger, you add a new word to the test input file and make the corresponding adjustment to the test reference file. You verify that the build pipeline is triggered by a commit push to Cloud Source Repositories and that the data-processing workflow runs correctly with the updated test files.

1. In Cloud Shell, add a test word at the end of the test file:

```
echo "testword" >>  ~/$SOURCE_CODE_REPO/workflow-dag/support-files/input.txt
```

2. Update the test result reference file, ref.txt, to match the changes done in the test input file:

   ```
   echo "testword: 1" >>  ~/$SOURCE_CODE_REPO/workflow-dag/support-files/ref.txt
   ```

3. Commit and push changes to Cloud Source Repositories:

   ```
   cd ~/$SOURCE_CODE_REPO
   git add .
   git commit -m 'change in test files'
   git push google master
   ```

4. In the Cloud Console, go to the **History** page.

   GO TO HISTORY PAGE
5. To verify that a new build is triggered by the previous push to master branch, on the current running build, the **Trigger** column says **Push to master branch**.
6. In Cloud Shell, get the URL for your Cloud Composer web interface:

   ```
   gcloud composer environments describe $COMPOSER_ENV_NAME \
       --location $COMPOSER_REGION --format="get(config.airflowUri)"
   ```

7. After the build finishes, go to the URL from the previous command to verify that the test_word_count DAG is running.

   Wait until the DAG run finishes, which is indicated when the light green circle in the **DAG runs** column goes away. It usually takes between three to five minutes for the process to finish.
8. In Cloud Shell, download the test result files:

   ```
   mkdir ~/result-download
   cd ~/result-download
   gsutil cp gs://$RESULT_BUCKET_TEST/output* .
   ```

9. Verify that the newly added word is in one of the result files:

   ```
   grep testword output*
   ```

   The output is similar to the following:

   ```
   output-00000-of-00003:testword: 1
   ```

Clean up

To avoid incurring charges to your Google Cloud account for the resources used in this experiment delete the individual resources.

Delete the individual resources

If you want to keep the project used for this experiment, run the following steps to delete the resources you created in this experiment.

1. To delete the Cloud Build trigger, complete the following steps:
    a. In the Cloud Console, go to the **Triggers** page.

       Go to Triggers page
    b. Next to the trigger that you created, click **More ...**, and then click **Delete**.

2. In Cloud Shell, delete the Cloud Composer environment:

```
gcloud -q composer environments delete $COMPOSER_ENV_NAME \
    --location $COMPOSER_REGION
```

3. Delete the Cloud Storage buckets and their files:

```
gsutil -m rm -r gs://$DATAFLOW_JAR_BUCKET_TEST \
    gs://$INPUT_BUCKET_TEST \
    gs://$REF_BUCKET_TEST \
    gs://$RESULT_BUCKET_TEST \
    gs://$DATAFLOW_STAGING_BUCKET_TEST \
    gs://$DATAFLOW_JAR_BUCKET_PROD \
    gs://$INPUT_BUCKET_PROD \
    gs://$RESULT_BUCKET_PROD \
    gs://$DATAFLOW_STAGING_BUCKET_PROD
```

4. Delete the repository:

```
gcloud -q source repos delete $SOURCE_CODE_REPO
```

5. Delete the files and folder you created:

```
rm -rf ~/ci-cd-for-data-processing-workflow
rm -rf ~/$SOURCE_CODE_REPO
rm -rf ~/result-download
```