**Manage Data Pipelines with Cloud Data Fusion and Cloud Composer.**

**Module Introduction.**

In this module, we will discuss how to manage data pipelines with Cloud Data Fusion and Cloud Composer.

Specifically, we will look at how you can use Cloud Data Fusion to visually build data pipelines and how you can use Cloud Composer to orchestrate work between Google Cloud services.



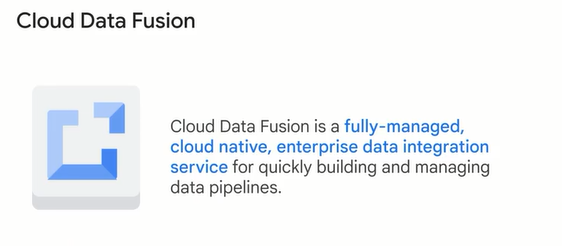
**Introduction to Cloud Data Fusion.**

Let's start with an introduction to Cloud Data Fusion.

Cloud Data Fusion provides a graphical user interface and APIs that increase time efficiency and reduce complexity.

It equips business users, developers and data scientists to quickly and easily build, deploy and manage data integration pipelines.

Cloud Data Fusion is essentially a graphical, no-code tool to build data pipelines.

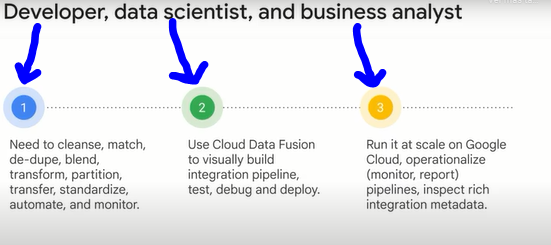


Cloud Data Fusion is used by developers, data scientists and business analysts alike.

For developers, Cloud Data Fusion allows you to cleanse, match, remove duplicates, blend, transform, partition, transfer, standardize, automate and monitor data.

Data scientists can use Cloud Data Fusion to visually build integration pipelines, test, debug and deploy applications.

Business analysts can run Cloud Data Fusion at scale on Google Cloud, operationalize pipelines and inspect rich integration metadata.



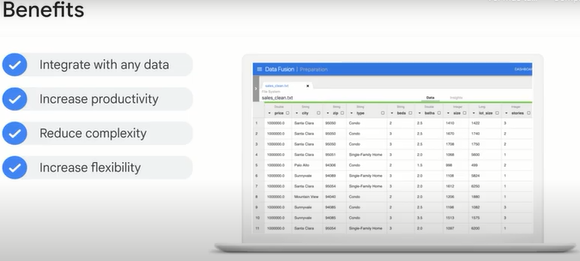
Cloud Data Fusion offers a number of benefits: Integrate with any data through a rich ecosystem of connectors for a variety of legacy and modern systems, relational databases, file systems, cloud services, object stores, NoSQL, EBCDIC and more.

Increase productivity: If you have to constantly move between numerous systems to gather insight, your productivity is significantly reduced.

With Cloud Data Fusion, your data from all the different sources can be pooled into a view like in BigQuery, Cloud Spanner or any other Google Cloud technologies, allowing you to be more productive faster.

Reduce complexity: Through a visual interface for building data pipelines, code-free transformations and reusable pipeline templates.

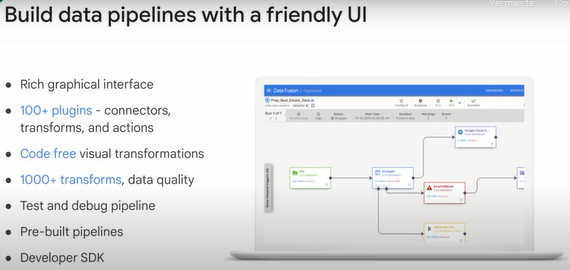
Increase flexibility: Through support for on-prem and cloud environments, interoperability with the open source software CDAP.



At a high level, Cloud Data Fusion provides you with a graphical user interface to build data pipelines with no code.

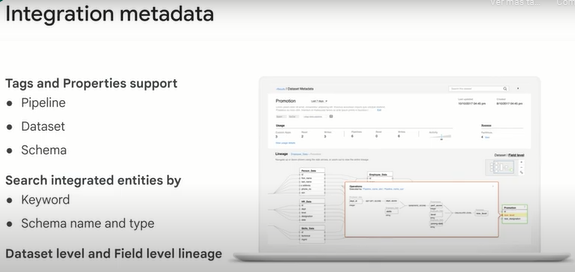
You can use existing templates, connectors to Google Cloud and other cloud services providers and an entire library of transformations to help you get your data in the format and quality you want.

Also, you can test and debug the pipeline and follow along with each node as it receives and processes data.



As you will see in the next lesson, you can tag pipelines to help organize them more efficiently for your team, and you can use the unified search functionality to quickly find field values or other keywords across your pipelines and schemas.

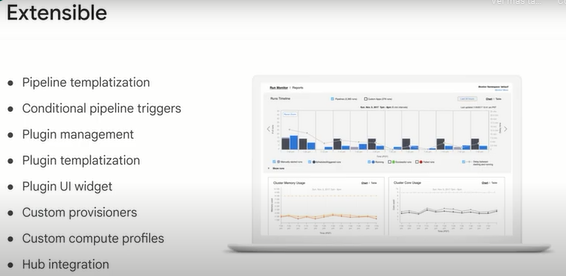
Lastly, we'll talk about how Cloud Data Fusion tracks the lineage of transformations that happen before and after any given field on your data set.



One of the advantages of Cloud Data Fusion is that it's extensible.

This includes the ability to templatize pipelines, create conditional triggers and manage and templatize plugins.

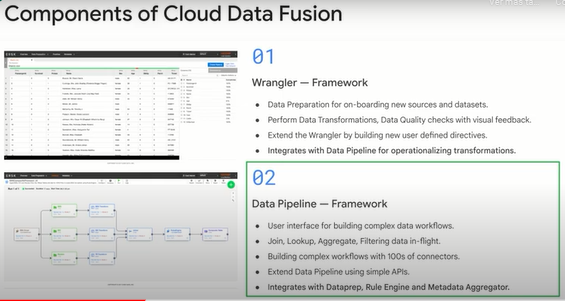
There is a UI widget plugin as well as custom provisioners, custom compute profiles and the ability to integrate to hubs.



**Components of Cloud Data Fusion.**

The two major user interface components we will focus our attention on in this course are the Wrangler UI for exploring data sets visually and building pipelines with no code, and the data pipeline UI for drawing pipelines right onto a canvas.

You can choose from existing templates for common data processing paths like cloud storage to BigQuery.



There are other features of Cloud Data Fusion that you should be aware of too.

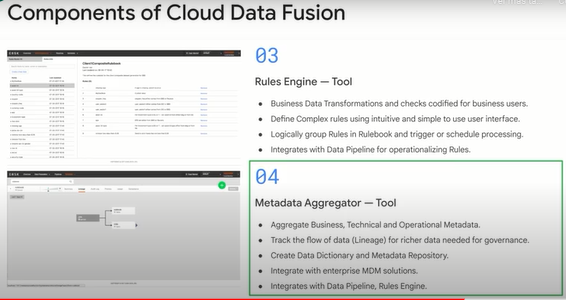
There's an integrated rules engine where business users can program in their predefined checks and transformations and store them in a single place.

Then data engineers can call these rules as part of a rulebook or pipeline later.

We mentioned data lineage as part of field metadata earlier.

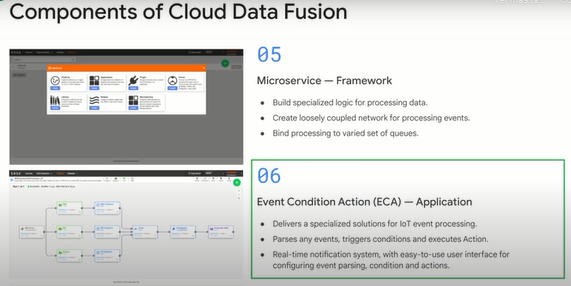
You can use the metadata aggregator to access the lineage of each field in a single UI and analyze other rich metadata about your pipelines and schemas as well.

For example, you can create and share a data dictionary for your schemas directly within the tool.



Other features, such as the microservice framework allow you to build specialized logic for processing data.

You can also use the Event Condition Action, ECA, application to parse any event, trigger conditions and execute an action based on those conditions.



**Cloud Data Fusion UI.**

Managing your pipelines is easiest when you have the right tools.

We'll now take a high level look at the Cloud Data Fusion UI as you saw in the component overview.

Here are some of the key user interface elements that you will encounter when using Data Fusion.

Let's look at each of them in turn.

Under Control Center is the section for applications, artifacts and a data set.

Here you could have multiple pipelines associated with a particular application.

The Control Center gives you the ability to see everything at a glance and search for what you need, whether it's a particular data set, pipeline or other artifact, like a data dictionary, for example.

Under the Pipeline section, you have a developer studio.

You can preview, export, schedule a job or project.

You also have a connector and a function palette and a navigation section.

Under the Wrangler section, you have connections, transforms, data quality, insights and functions.

Under the Integration metadata section, you can search, add tags and properties and see the data lineage for field and data.

The Hub allows you to see all the available plugins, sample use cases and prebuilt pipelines.

Entities include the ability to create pipelines, upload an application, plugin, driver, library and directives.

There are two components in Administration, management and configuration.

Under management, you have services and metrics.

Under configuration, you have namespace, compute profiles, preferences, system artifacts and the REST client.

**Build a pipeline.**

Now that we've looked at the components in the UI, we'll discuss the process of building a data pipeline.

A pipeline is represented visually as a series of stages arranged in a graph.

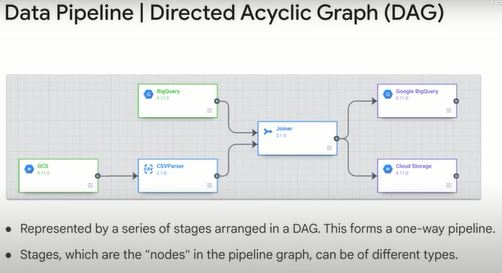
These graphs are called DAGs, or Directed Acyclic Graphs, because they flow from one direction to another, and they cannot feed into themselves.

Acyclic simply means not a circle.

Each stage is a node.

And as you can see here, nodes can be of a different type, you may start with a node that pulls data from Cloud Storage, then passes it on to a node that parses a CSV.

The next node takes multiple nodes, has an input and joins them together before passing the joined data to two separate data sink nodes.



As you saw in our previous example, you can have multiple nodes fork out from a single parent node.

This is useful because you may want to kick off another data processing work stream that should not be blocked by any processing on a separate series of nodes.

You can combine data from two or more nodes into a single output in a sink.

In Cloud data fusion, the studio is the user interface where you author and create new pipelines.

The area where you create nodes and chain them together in your pipeline is your canvas.

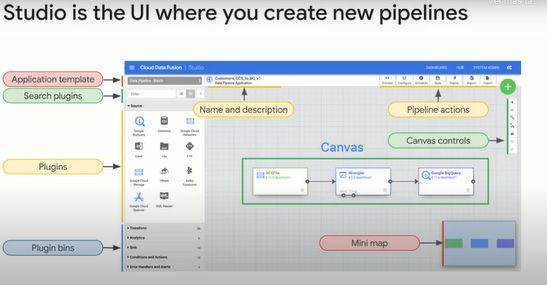
If you have many nodes in a pipeline, the Canvas can get visually cluttered, so use the mini map to help navigate around a huge pipeline quickly.

You can interact with the canvas and add objects by using the Canvas control panel.

When you're ready to save and run the entire pipeline, you can do so with the pipeline actions toolbar at the top.

Don't forget to give your pipeline a name and description, as well as make use of the many pre-existing templates and plugins so you don't have to write your pipeline from scratch.

Here, we've used a template or data pipeline batch, which gives us the three nodes you see here to move data from a Cloud storage file, process it in a wrangler and output it to BigQuery.



You should make use of preview mode before you deploy and run your pipeline in production to ensure everything you run will run properly.

While a pipeline is in preview, you can click on each node and see any sample data or errors that you will need to correct before deploying.

After deployment.

You can monitor the health of your pipeline and collect key summary stats of each execution.

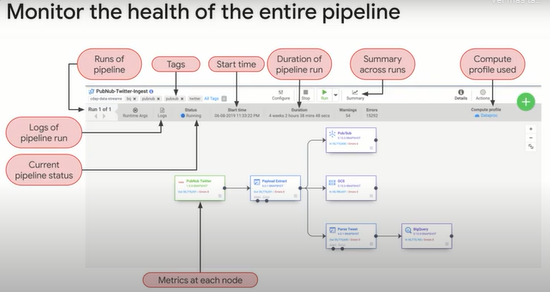
Here, we're ingesting data from Twitter and Google Cloud and parsing each tweet before loading them into a variety of data sinks.

If you have multiple pipelines, it's recommended that you make liberal use of the tags feature to help you quickly find and organize each pipeline for your organization.

You can view the start time, the duration of the pipeline run and the overall summary across runs for each pipeline.

You can quickly see the data throughput at each node in the pipeline simply by interacting with the node.

Note the compute profile used in the Cloud

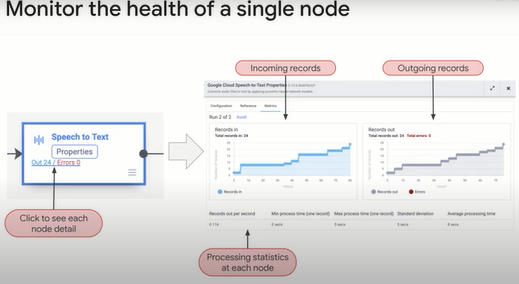


.

Clicking on a node gives you detail on the inputs, outputs and errors for that given node.

Here, we are integrating with the speech to text API to process audio files into searchable text.

You can track the individual health of each node and get useful metrics like records out per second, average processing time, and Max processing time, which can alert you to any anomalies in your pipeline.



You can set your pipelines to run automatically at certain intervals.

If your pipeline normally takes a long time to process the entire data set, you can also specify a maximum number of concurrent runs to help avoid processing data unnecessarily.

Keep in mind that Cloud data fusion is designed for batch data pipelines.

We'll dive into streaming data pipelines in future modules.

One of the big features of Cloud data fusion is the ability to attract the lineage of a given field value.

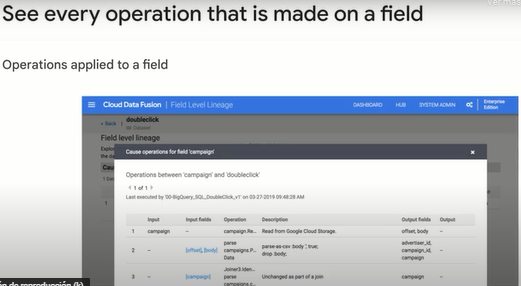
Let's take this example of a campaign field for double click data set and track every transform operation that happened before and after this field.

Here, you can see the lineage of operations that are applied to the campaign field between the campaign dataset and the double click dataset.

Note the time this field was last changed by a pipeline run and each of the input fields and descriptions that interacted with the field as part of processing it between datasets.

Imagine the use cases if you've inherited a set of analytical reports and you want to walk back upstream all of the logic that went into a certain field.

Well, now you can.



**Explore data using wrangler.**

The We've discussed the core components, tools and processes of building data pipelines.

Now we'll look at using Wrangler to explore the data set.

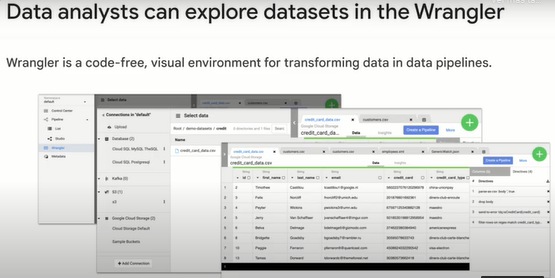
So far in the course, we have focused on building new pipelines for our data sets.

That presumes we know what the data is and what transformations need to be made already.

Oftentimes, a new data set still needs to be explored and analyzed for insights.

The Wrangler UI is the cloud data fusion environment for exploring new data sets visually for insights.

Here, you can inspect the data set and build a series of transformation steps called directives to stitch together (“*coser juntos*”) a pipeline.



Here's what the Wrangler UI looks like.

Starting from the left, you have your connections to existing data sets.

You can add new connections to a variety of data sources like Google cloud storage, BigQuery or even other cloud providers.

Once you specify your connection, you can browse all of the files and tables in that source.

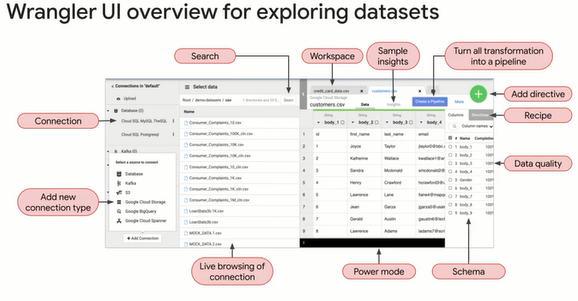
Here, you can see a cloud storage bucket of demo data sets and all the CSV files of customer complaints.

Once you've found an example data set like customers.csv here, you can explore the rows and columns visually and view sample insights.

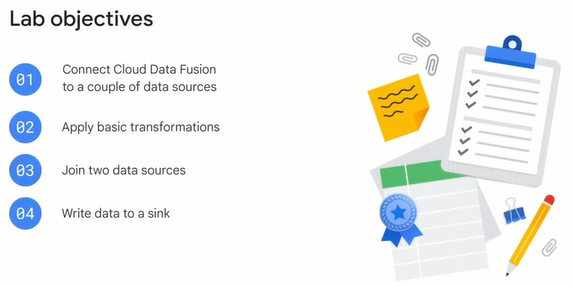
As you explore the data, you might want to create new calculated fields, drop columns, filter rows or otherwise wrangle the data.

You can do so using the Wrangler UI by adding new directives to form a data transformation recipe.

When you're happy with your transformations, you can create a pipeline that you can then run at regular intervals.



**Lab: Building and Executing a Pipeline Graph with Data Fusion 2.5.**



**Overview**

This tutorial shows you how to use the Wrangler and Data Pipeline features in Cloud Data Fusion to clean, transform, and process taxi trip data for further analysis.

**Task 1. Creating a Cloud Data Fusion instance**

Thorough directions for creating a Cloud Data Fusion instance can be found in the [Creating a Cloud Data Fusion instance Guide](https://cloud.google.com/data-fusion/docs/how-to/create-instance" \t "_blank). The essential steps are as follows:

1. To ensure the training environment is properly configured you must first stop and restart the Cloud Data Fusion API. Run the command below in the Cloud Shell. It will take a few minutes to complete.

gcloud services disable datafusion.googleapis.com

Your output says that the operation finished successfully.

Next, restart the connection to the Cloud Data Fusion API.

1. In the Google Cloud Console, enter **Cloud Data Fusion API** in the top search bar. Click on the result for Cloud Data Fusion API.
2. On the page that loads click **Enable**.
3. When the API has been enabled again, the page will refresh and show the option to disable the API along with other details on the API usage and performance.
4. On the **Navigation menu**, select **Data Fusion**.
5. To create a Cloud Data Fusion instance, click **Create an Instance**.
6. Enter a name for your instance.
7. Select **Basic** for the Edition type.
8. Under **Authorization** section, click **Grant Permission**.
9. Leave all other fields as their defaults and click **Create**.

**Note:**Creation of the instance can take around 15 minutes.

1. Once the instance is created, you need one additional step to grant the service account associated with the instance permissions on your project. Navigate to the instance details page by clicking the instance name.
2. Copy the service account to your clipboard.
3. In the GCP Console navigate to the **IAM & Admin > IAM**.
4. On the IAM Permissions page, add the service account you copied earlier as a new principals and grant the **Cloud Data Fusion API Service Agent** role, by clicking the **Add** button.



1. Click **Save**.

**Task 2. Loading the data**

Once the Cloud Data Fusion instance is up and running, you can start using Cloud Data Fusion. However, before Cloud Data Fusion can start ingesting data you have to take some preliminary steps.

1. In this example, Cloud Data Fusion will read data out of a storage bucket. In the [cloud shell console](https://cloud.google.com/shell/docs/starting-cloud-shell" \t "_blank) execute the following commands to create a new bucket and copy the relevant data into it:

export BUCKET=$GOOGLE\_CLOUD\_PROJECT

gsutil mb gs://$BUCKET

gsutil cp gs://cloud-training/OCBL017/ny-taxi-2018-sample.csv gs://$BUCKET

**Note:**The created bucket name is your project id.

1. In the command line, execute the following command to create a bucket for temporary storage items that Cloud data Fusion will create:

gsutil mb gs://$BUCKET-temp

**Note:**The created bucket name is your project id followed by "-temp".

1. Click the **View Instance** link on the Data Fusion instances page, or the details page of an instance. Click **username**. If prompted to take a tour of the service click on **No, Thanks**. You should now be in the Cloud Data Fusion UI.

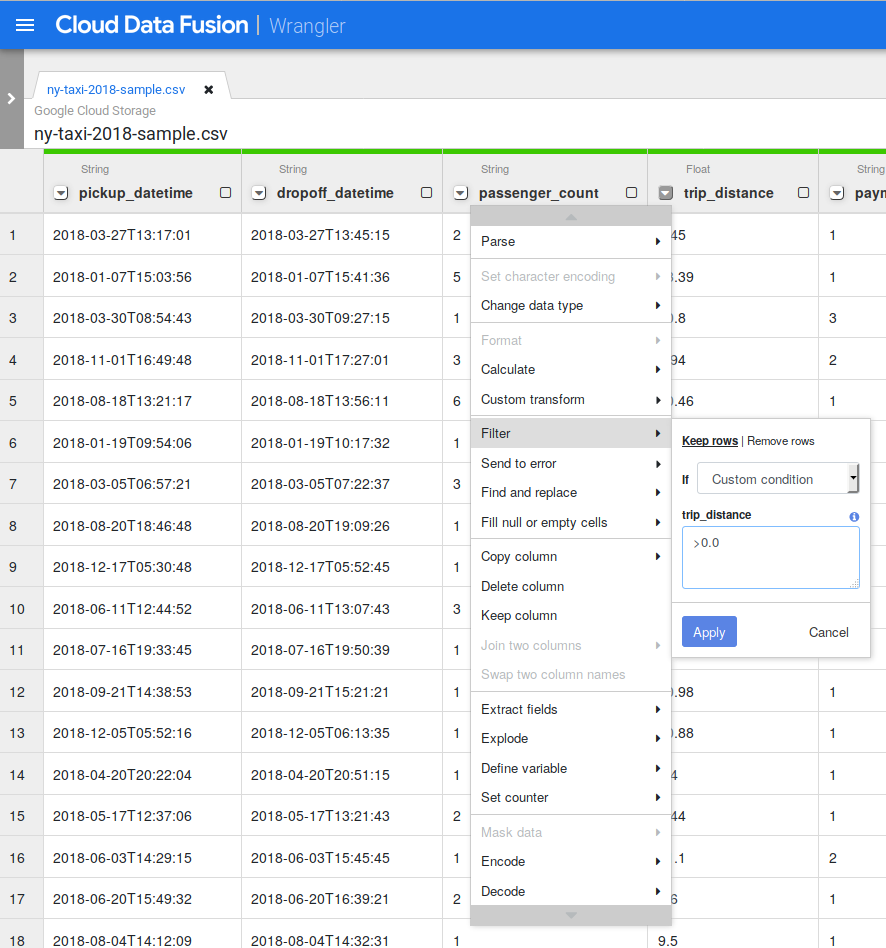
**Note:**You may need to reload or refresh the Cloud Fusion UI pages to allow prompt loading of the page.

1. **Wrangler** is an interactive, visual tool that lets you see the effects of transformations on a small subset of your data before dispatching large, parallel-processing jobs on the entire dataset. On the Cloud Data Fusion UI, choose **Wrangler**. On the left side, there is a panel with the pre-configured connections to your data, including the Cloud Storage connection.
2. Under **GCS**, select **Cloud Storage Default**.
3. Click on the bucket corresponding to your project name.
4. Select **ny-taxi-2018-sample.csv**. The data is loaded into the Wrangler screen in row/column form.

**Task 3. Cleaning the data**

Now, you will perform some transformations to parse and clean the taxi data.

1. To the left of the body column, click the **Down** arrow.
2. Click **Parse > CSV**, select **Set first row as header** and then click **Apply.** The data splits into multiple columns.
3. Because the body column isn't needed anymore, click the **Down** arrow next to the body column and choose **Delete column**.
4. You'll notice that all of the column types have been loaded in as String. Click the **Down** arrow next to the trip\_distance column, select **Change data type** and then click on **Float**. Repeat for the total\_amount column.
5. If you look at the data closely, you may find some anomalies, such as negative trip distances. You can avoid those negative values by filtering out in **Wrangler**. Click the **Down** arrow next to the trip\_distance column and select **Filter**. Click if **Custom condition** and input >0.0



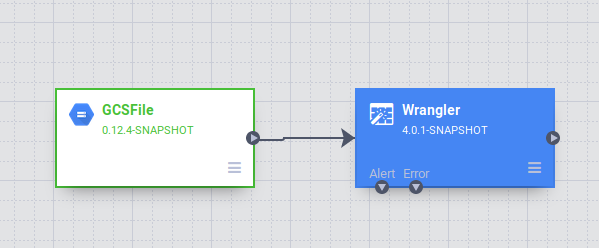
1. Click on **Apply**.

**Task 4. Creating the pipeline**

Basic data cleansing is now complete and you've run transformations on a subset of your data. You can now create a batch pipeline to run transformations on all your data.

Cloud Data Fusion translates your visually built pipeline into an Apache Spark or MapReduce program that executes transformations on an ephemeral Cloud Dataproc cluster in parallel. This enables you to easily execute complex transformations over vast quantities of data in a scalable, reliable manner, without having to wrestle with infrastructure and technology.

1. On the upper-right side of the Google Cloud Fusion UI, click **Create a Pipeline**.
2. In the dialog that appears, select **Batch pipeline**.
3. In the Data Pipelines UI, you will see a GCSFile source node connected to a Wrangler node. The Wrangler node contains all the transformations you applied in the Wrangler view captured as directive grammar. Hover over the Wrangler node and select **Properties**.

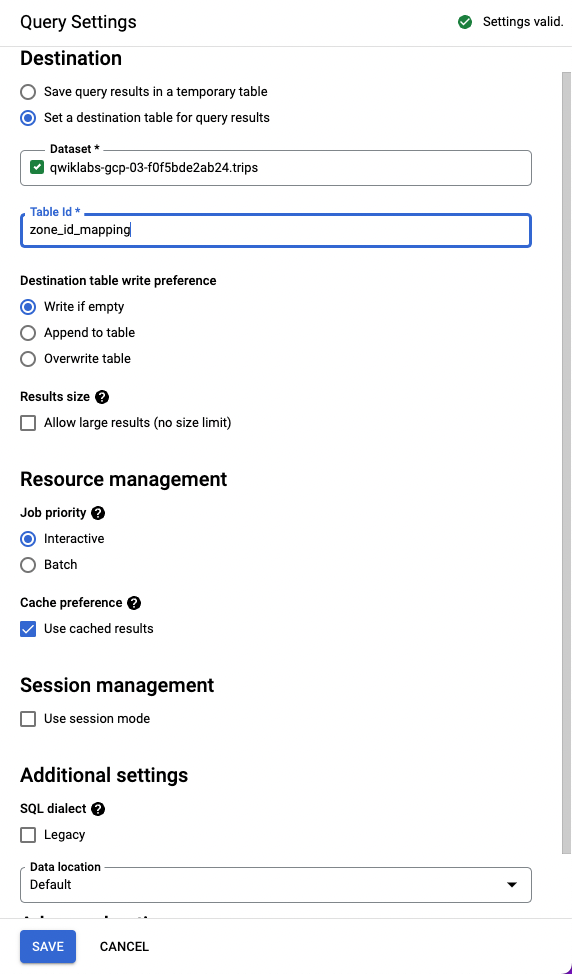
.

1. At this stage, you can apply more transformations by clicking the **Wrangle** button. Delete the extra column by pressing the red trashcan icon beside its name. Click **Validate** on top right corner to check for any errors. To close the Wrangler tool click the **X** button in the top right corner.

**Task 5. Adding a data source**

The taxi data contains several cryptic columns such as pickup\_location\_id, that aren't immediately transparent to an analyst. You are going to add a data source to the pipeline that maps the pickup\_location\_id column to a relevant location name. The mapping information will be stored in a BigQuery table.

1. In a separate tab, [open the BigQuery UI in the Cloud Console](https://console.cloud.google.com/bigquery). Click **Done** on the 'Welcome to BigQuery in the Cloud Console' launch page.
2. In the Explorer section of the BigQuery UI, click the three dots beside your GCP Project ID (it will start with qwiklabs).
3. On the menu that appears click the **Create dataset** link.
4. In the **Dataset ID** field type in trips.
5. Click on **Create dataset**.
6. To create the desired table in the newly created dataset, navigate to **More > Query Settings**. This process will ensure you can access your table from Cloud Data Fusion.
7. Select the item for **Set a destination table for query results**. For **Dataset** input trip select from the dropdown. **Table Id** input zone\_id\_mapping. Click **Save**.



1. Enter the following query in the Query Editor and then click **Run**:

SELECT

zone\_id,

zone\_name,

borough

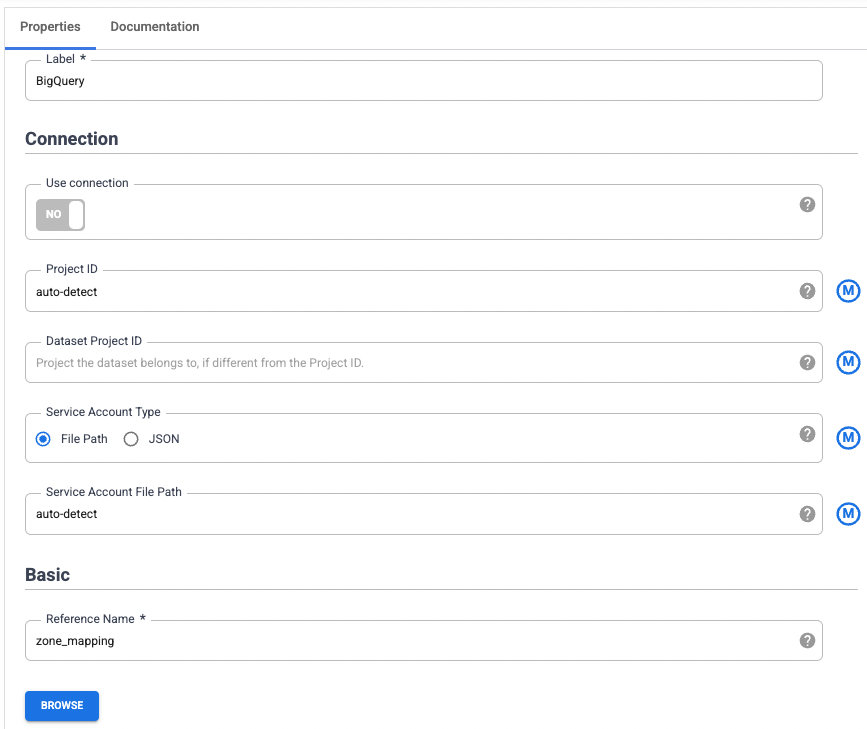
FROM

`bigquery-public-data.new\_york\_taxi\_trips.taxi\_zone\_geom`

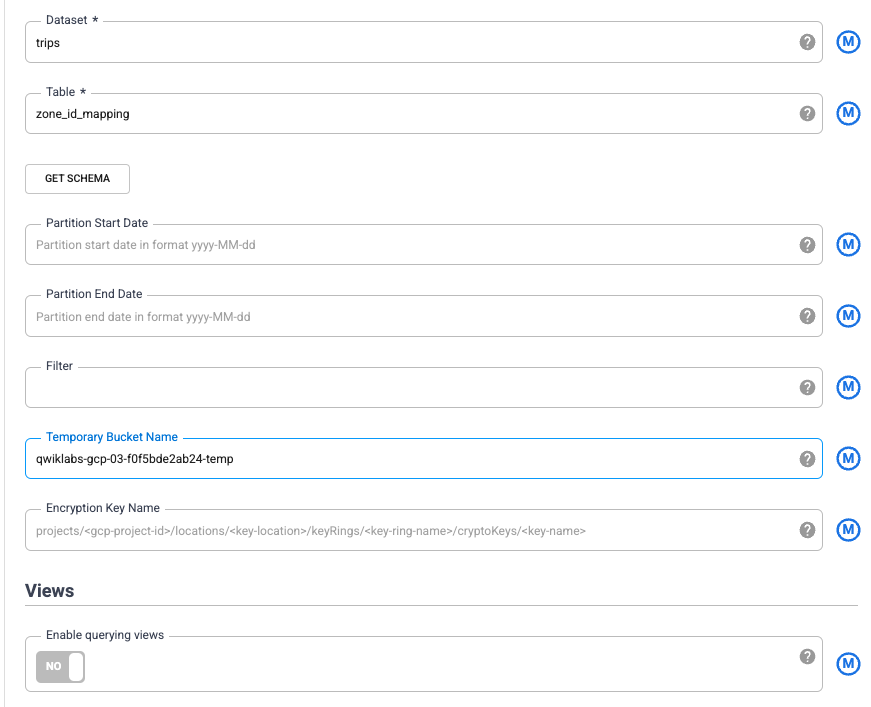
You can see that this table contains the mapping from zone\_id to its name and borough.



1. Now, you will add a source in your pipeline to access this BigQuery table. Return to tab where you have Cloud Data Fusion open, from the Plugin palette on the left, select **BigQuery** from the **Source** section. A BigQuery source node appears on the canvas with the two other nodes.
2. Hover over the new BigQuery source node and click **Properties**.
3. To configure the **Reference Name**, enter zone\_mapping, which is used to identify this data source for lineage purposes.



1. The BigQuery **Dataset** and **Table** configurations are the Dataset and Table you setup in BigQuery a few steps earlier: trips and zone\_id\_mapping. For **Temporary Bucket Name** input the name of your project followed by "-temp", which corresponds to the bucket you created in Task 2.

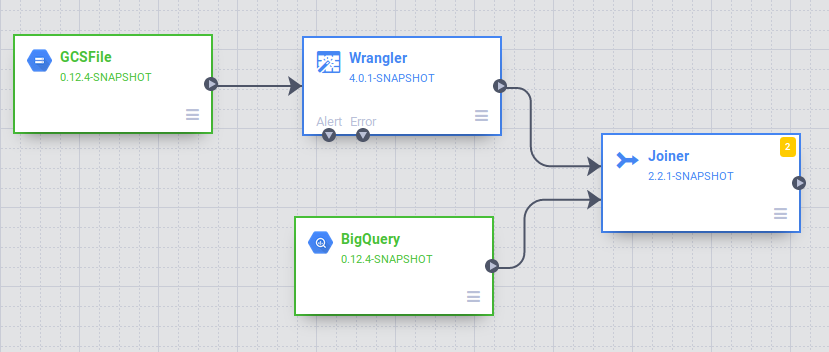


1. To populate the schema of this table from BigQuery, click **Get Schema**. The fields will appear on the right side of the wizard.
2. Click **Validate** on top right corner to check for any errors. To close the BigQuery Properties window click the **X** button in the top right corner.

**Task 6. Joining two sources**

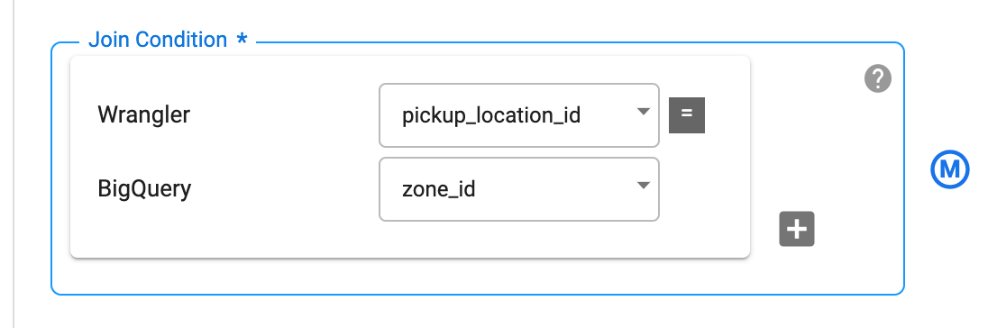
Now you can join the two data sources—taxi trip data and zone names—to generate more meaningful output.

1. Under the **Analytics** section in the Plugin Palette, choose **Joiner**. A **Joiner** node appears on the canvas.
2. To connect the Wrangler node and the BigQuery node to the Joiner node: Drag a connection arrow > on the right edge of the source node and drop on the destination node.

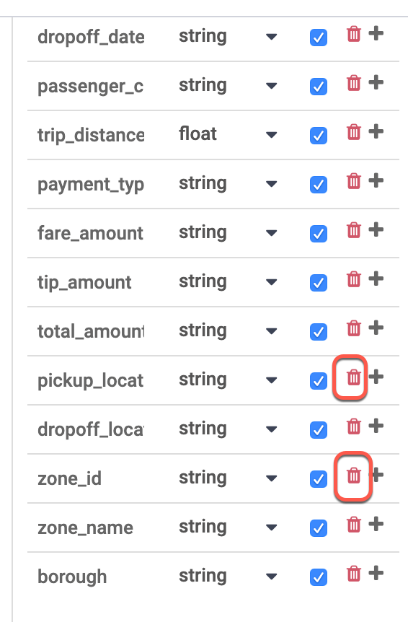


1. To configure the Joiner node, which is similar to a SQL JOIN syntax:

* Click **Properties** of **Joiner**.
* Leave the label as **Joiner**.
* Change the **Join Type** to **Inner**
* Set the **Join Condition** to join the pickup\_location\_id column in the Wrangler node to the zone\_id column in the BigQuery node.



* To generate the schema of the resultant join, click **Get Schema**.
* In the **Output Schema** table on the right, remove the zone\_id and pickup\_location\_id fields by hitting the red garbage can icon.

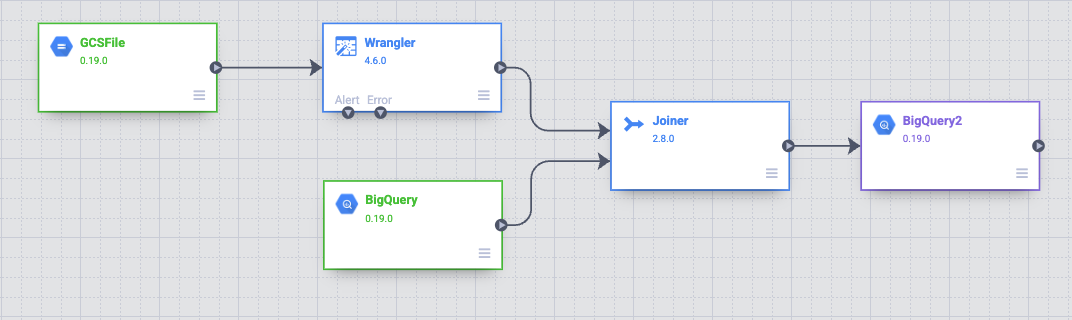


* Click **Validate** on top right corner to check for any errors. Close the window by clicking the **X** button in the top right corner.

**Task 7. Storing the output to BigQuery**

You will store the result of the pipeline into a BigQuery table. Where you store your data is called a sink.

1. In the **Sink** section of the Plugin Palette, choose **BigQuery**.
2. Connect the **Joiner** node to the **BigQuery** node. Drag a connection arrow > on the right edge of the source node and drop on the destination node.

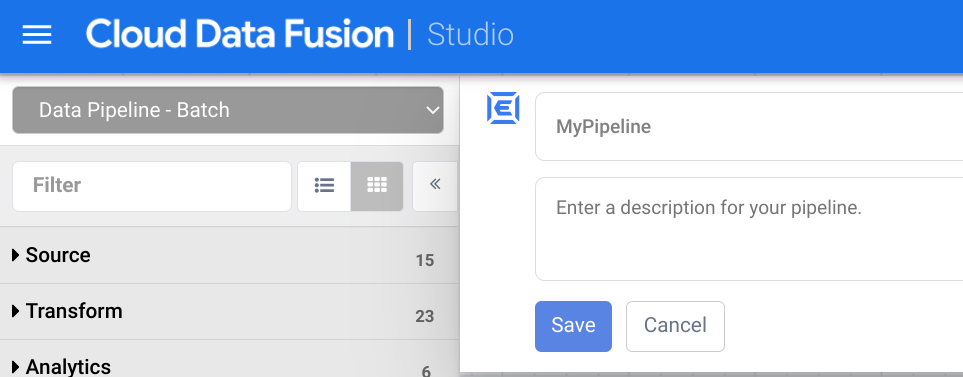


1. Open the BigQuery2 node by hovering on it and then clicking **Properties**. You will next configure the node as shown below. You will use a configuration that's similar to the existing BigQuery source. Provide bq\_insert for the **Reference Name** field and then use trips for the **Dataset** and the name of your project followed by "-temp" as **Temporary Bucket Name**. You will write to a new table that will be created for this pipeline execution. In **Table** field, enter trips\_pickup\_name.
2. Click **Validate** on top right corner to check for any errors. Close the window by clicking the **X** button in the top right corner.

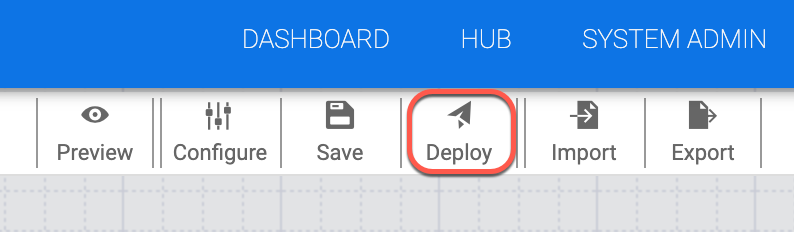
**Task 8. Deploying and running the pipeline.**

At this point you have created your first pipeline and can deploy and run the pipeline.

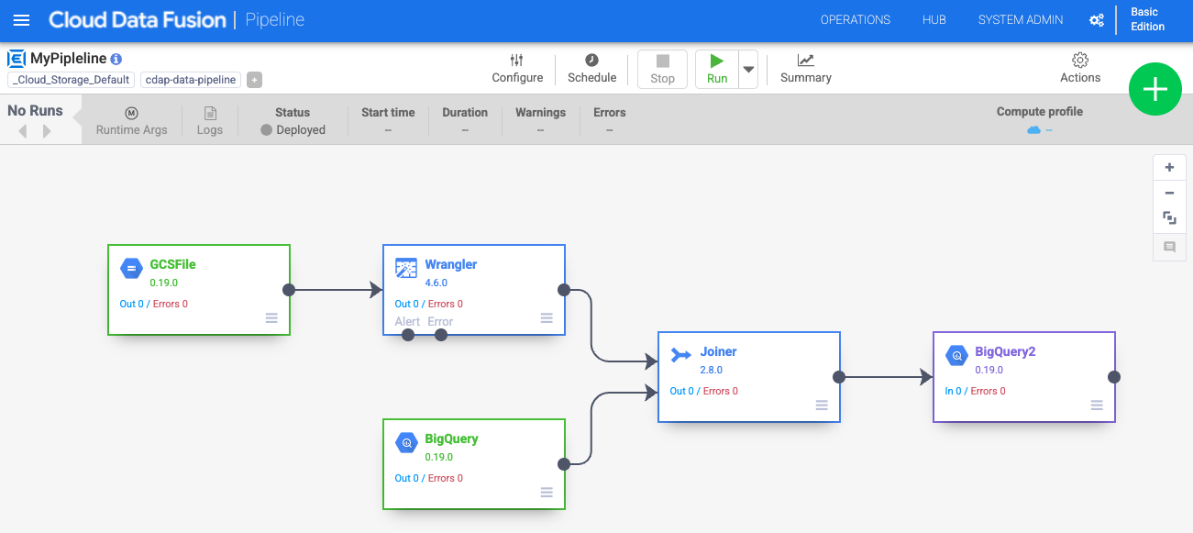
1. Name your pipeline in the upper left corner of the Data Fusion UI and click **Save**.



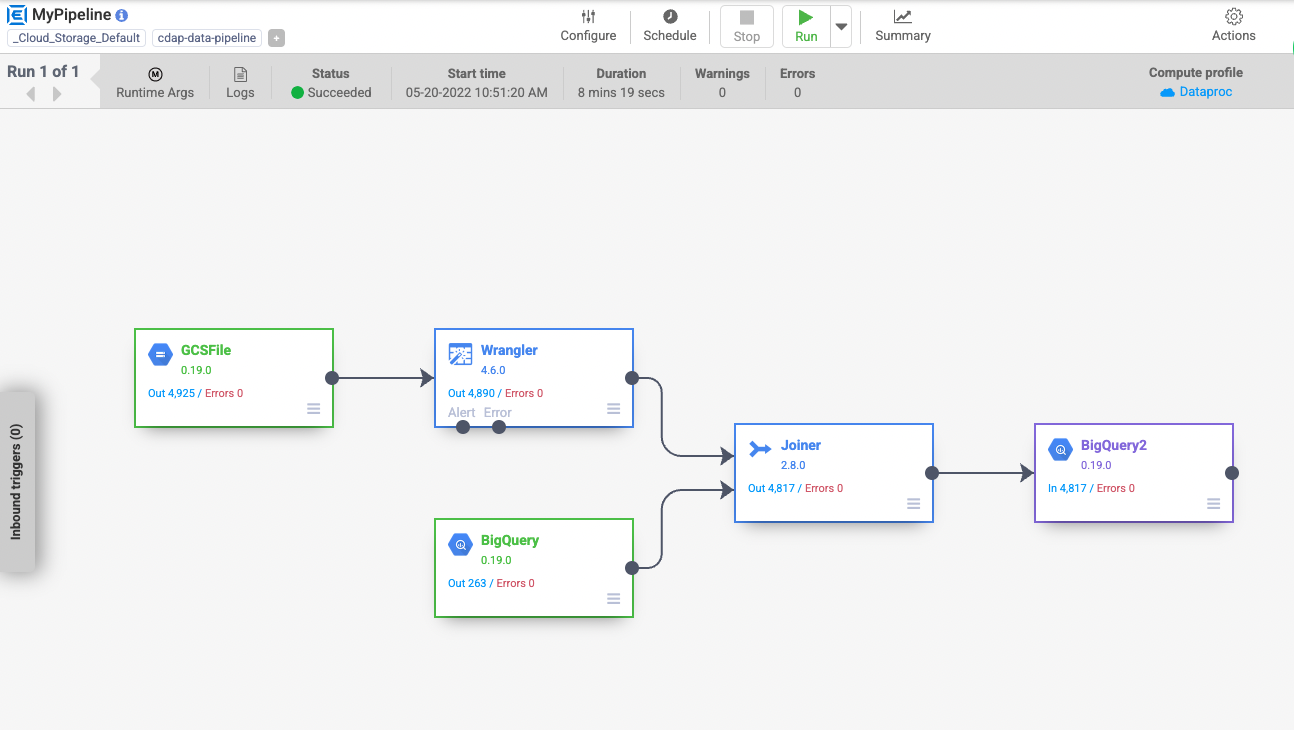
1. Now you will deploy the pipeline. In the upper-right corner of the page, click **Deploy**.



1. On the next screen click **Run** to start processing data.



When you run a pipeline, Cloud Data Fusion provisions an ephemeral Cloud Dataproc cluster, runs the pipeline, and then tears down the cluster. This could take a few minutes. You can observe the status of the pipeline transition from *Provisioning* to *Starting* and from *Starting* to *Running* to *Succeeded* during this time.



**Note:** The pipeline transition may take 10-15 minutes to succeeded.

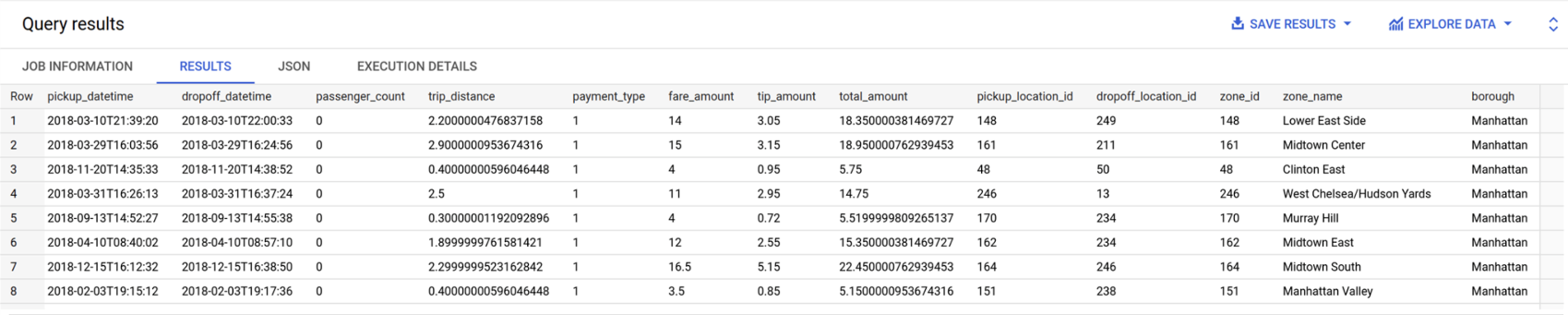
**Task 9. Viewing the results.**

To view the results after the pipeline runs:

* Return to the tab where you have BigQuery open. Run the query below to see the values in the trips\_pickup\_name table:
* SELECT
* \*
* FROM

`trips.trips\_pickup\_name`

BQ RESULTS



**End your lab**

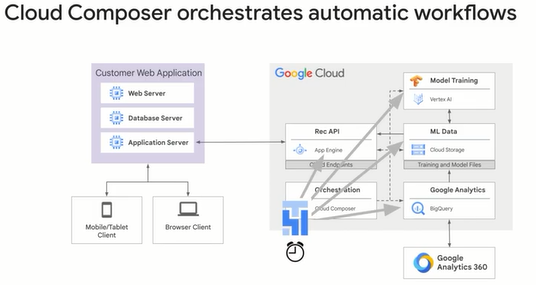
**Orchestrate work between Google Cloud services with Cloud Composer.**

The next big task for managing data pipelines is to orchestrate the work across multiple Google Cloud services.

For example, if you have three cloud data fusion pipelines, and two ML models that you wanted to run in a certain order, you need an orchestration engine.

In this module, we'll look at using Cloud Composer to help out with tasks like that.

Cloud Composer will control the Google Cloud services that we need to run.



What Cloud Composer is simply a serverless environment on which an open source workflow tool runs.

That workflow tool is called Apache airflow, which is an open source orchestration engine.

The heart of any workflow is D.A.G. As you saw with cloud data fusion, You're also building D.A.G.s with Apache Airflow as you see here.

What's happening in this particular D.A.G. are four tasks that update our training data, export it, retrain our model, and we deploy it.

You can tell your D.A.G. to do pretty much anything you need it to do.

Here, it's sending tasks to BigQuery, cloud storage, and Vertex AI.

But yours could orchestrate among four completely different services.



**Apache Airflow environment.**

Let's preview the actual Cloud Composer environment.

Once you use the command line or Google Cloud web UI to launch a Cloud Composer instance, you'll be met with a screen like this.

Keep in mind that you can have multiple Cloud Composer environments, and with each environment, you can have a separate Apache Airflow instance, which could have zero to many DAGs.

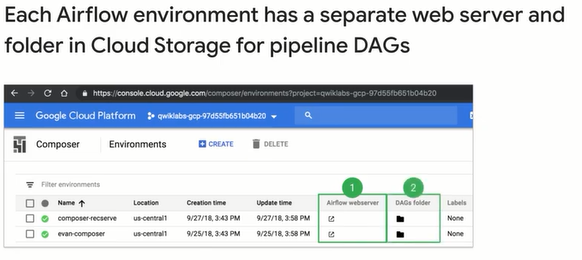
An important note here is that sometimes you'll be required to edit environment variables for your workflows, like specifying your specific Google Cloud project account.

Normally, you will not do that at the Cloud Composer level but on the actual Apache Airflow instance level.

Again, generally, you're only on the Cloud Composer page here to create new environments before you launch directly into the Airflow web server.

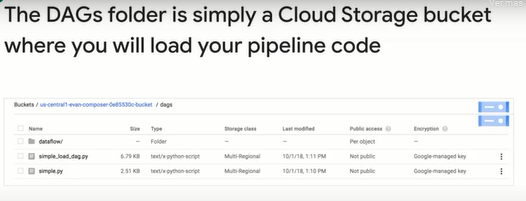
To access the Airflow admin UI, where you can monitor and interact with your workflows, you'll click on the link underneath Airflow webserver.

The second box you see is the DAGs folder, which is where the code of your actual workflows will be stored.



The DAGs folder for each airflow instance is simply a cloud storage bucket that is automatically created for you when you create your Cloud Composer instance.

Here is where you upload your DAG files written in Python, and bring your first workflow to life in Airflow.



**DAGs and Operators.**

Now that you're familiar with the basic environment setup, it's time to discuss your primary artifact, which is your DAG and the operators you're using to call whichever services you want to send tasks to.

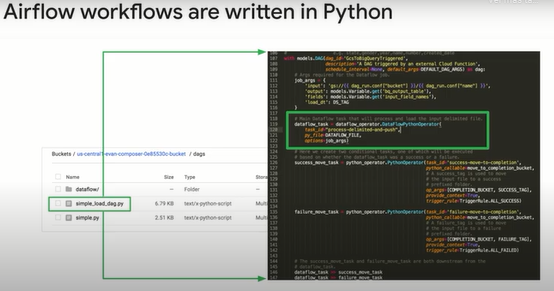
First, airflow workflows are written in Python, you'll have one Python file for each DAG.

For example, here we have simple \_load\_dag.py in our DAG folder Cloud Storage bucket, and you can see a preview of what the DAG file looks like.

Don't worry about reading the code, we'll go into that later.

It's sufficient enough for now to just know that there are a series of user created tasks in each DAG file that invoke predefined operators, like this task, which uses the data flow Python operator and is given the task ID of Process Delimited and Push.

We'll go over creating a DAG file and its components a little later.



Once you've uploaded the Python file to the DAG's folder, you can navigate back to the airflow web server, and under DAGs, you'll see the DAG you created with code represented visually as a directed graph with nodes and edges.

You'll remember that the Python code that defined a task we called Process Delimited and Push is now a node in our graph here.

Let's explore a bit more of the airflow Web UI.

You can see that this particular workflow is called GCS to BigQuery Triggered and it has three tasks when it runs.

One, process delimited and push.

I just happen to know from that Python file you saw earlier that this task invokes a Dataflow job to read in a new CSV file from a Cloud Storage bucket, processes it and writes the output to BigQuery.

Two, success move to completion, which moves the CSV file from an input Cloud Storage bucket to a process store completed Cloud Storage bucket for archiving.

Or three, if the pipeline fails partway, the file is moved to the completion bucket but tagged as failure.

This is an example of a DAG which isn't strictly sequential.

There, a decision is made to run one node or a different one based on the outcome of the parent note.

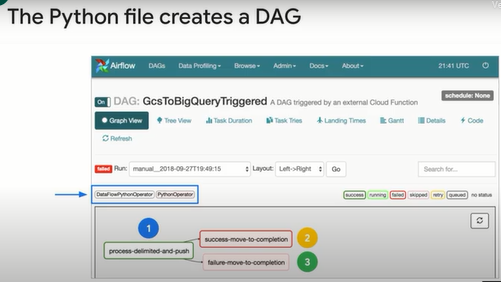
But regardless of the size and shape of your workflow DAG, one common thread for all workflows is the common operators used.

If the DAG itself is how to run the workflow, first do step one, then either move to step two or three, the operators specify what actually gets done as part of the task.

In this simple example, we're calling on the Dataflow Python operator and the general Python operator.

Those are by no means the only operators.

So let's pause here and look at all the operators at our disposal to achieve our goal of automatic retraining and deployment of our ML model.

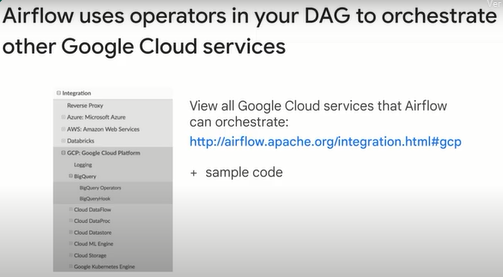


Airflow has many operators which you can invoke the tasks you want to complete.

Operators are usually atomic in a task, which means generally you only see one operator per task.

This list of all services that airflow can orchestrate to is taken directly from the Apache Airflow documentation.

Let's take a look at the ones that are likely most relevant to us as data engineers.



As you might have guessed, we'll certainly be making use of the BigQuery operators since our workflows depend on the data that is fed into them through Cloud storage and BigQuery.

Here's a list of the specific operators that we can invoke in a task to call on the BigQuery service for querying and other data related tasks.

You'll be mainly working with the first three in this course, but I encourage you to skim the resource link on all the operators so you can get a feel for what is possible.

Once we have our training data in a good place, the next logical step in our workflow is to retrain and redeploy our model.

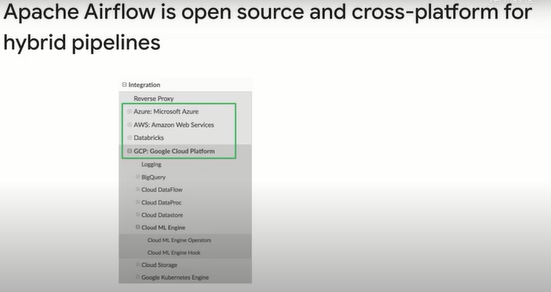
In the same DAG file, after the BigQuery operators complete, we can make a service call through a Vertex AI operator to kick off a new training job and manage our model like incrementing the version.

You might have noticed that your airflow DAG can have operators that send tasks out to other Cloud providers.

This is great for hybrid workflows where you have components across multiple Cloud platforms, or even on premise.

Apache Airflow is open source and continually adds more operators to other services.

So be sure to check out the list and the documentation periodically if you're waiting for a new service to be added.

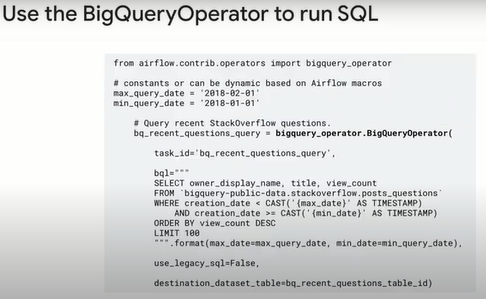


Here, you see four tasks: T1, T2, T3, and T4, and four operators corresponding to four Google Cloud services.

The names should look familiar, and you can probably start to guess what this pipeline does at a high level just by reading them in order.

The first two are concerned with getting fresh model data from a BigQuery dataset and into Cloud Storage for consumption by our ML model later in the pipeline.

In the lab you're going to work on later, the dataset will be the one you're already familiar with, the Google Analytics news articles sample dataset.



Let's see the parameters the BigQuery operator takes.

The BigQuery operator allows you to specify a SQL query to run against a BigQuery dataset.

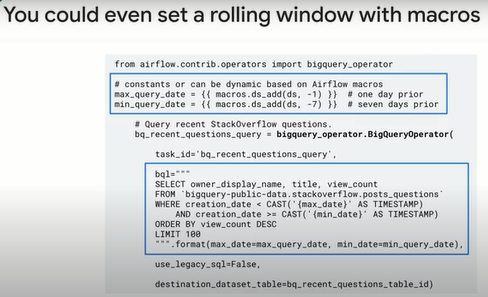
In this quick example, we're parsing in a query which returns the top 100 most popular Stack Overflow posts from the BigQuery public dataset for a specified date range you see there in the Where clause.

Notice anything different about the filters in the Where clause?

Yes, they are parameters.

In this case, for max date, and min date, you can parameterize pieces of the SQL statements like what we did here to only return posts for January 2018 with min query date, and max query date.

What is really powerful is that you can even make the parameters dynamic instead of the static ones shown here, and have them be based on the DAG schedule date, like macros.ds\_adds-7, which is a week before the DAG scheduled run date.



The next two operators handle retraining the model by submitting a job to the machine learning engine and then deploying the updated model to App Engine



.

At the end of almost all DAG files, you'll find the actual order in which we want these operators to run.

This is what the D in DAG is for, Directed.

For our example, T2 or task two won't run until T1 has completed.

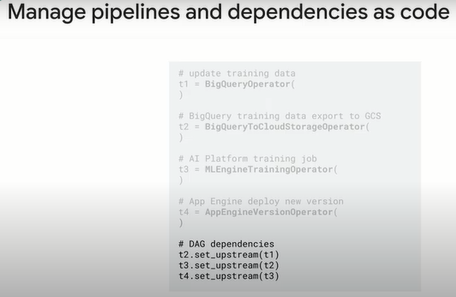
This is what gives our graph the dependencies of a workflow.

I can probably guess what you're thinking at this point, you could build some cool branching of multiple child nodes per upstream parent, and that is totally possible.

Just don't forget to comment your code so you know where one branch begins, where it's going, and what tasks are involved.

As a tip, after you load your DAG file into the DAG folder, you can see the visualization of your DAG in the airflow UI as a directed graph, a Gantt chart or a list if you want.

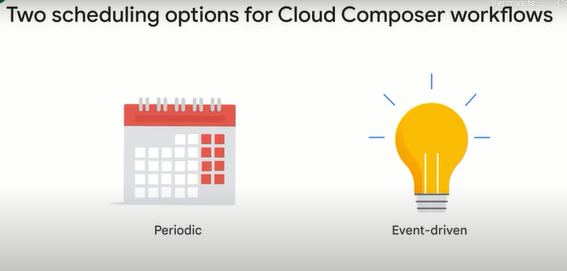
Reviewing the visual representation of the ordered tasks will help you confirm your tasks are ordered properly.



**Workflow scheduling**

Now that you're familiar with the Cloud Composer and Apache Airflow environments, and the basics of building a list of tasks for Google Cloud services in your DAG, it's time to discuss a really important topic: workflow scheduling. As we hinted at earlier, there are two different ways your workflow can be run

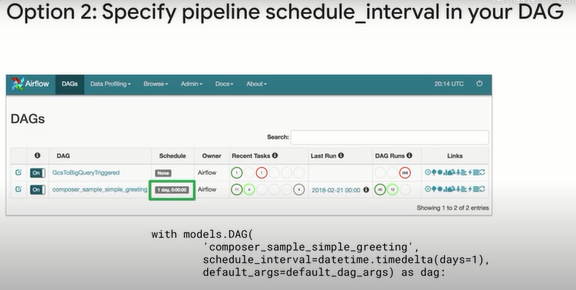
without you sitting there manually clicking run DAG. The first and most common is a set schedule or periodic run of a workflow, like once a day at 6:00am, or weekly on Saturdays. The second way is trigger based. Like if you wanted to run your workflow whenever new CSV data files were loaded into a Cloud Storage bucket,

or if new data came in from a Pub Sub topic you've subscribed to, then navigate to the DAGs tab to view the existing workflows that you have Python DAG files for. Here we have two DAGs, the bottom one composer sample simple greeting has a daily schedule. But why is this top DAG missing a schedule?

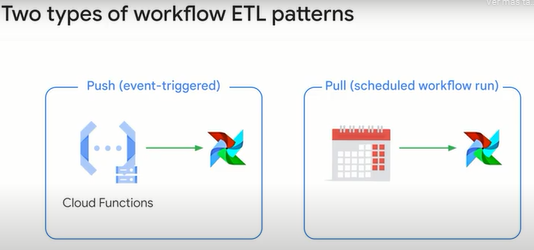
How would it ever get run? The answer is the fact that it's not on a set schedule at all. It's event driven. The driver of when this workflow runs is a Cloud function that we create. In the next lesson, we'll actually create our own Cloud function that watches a Cloud Storage bucket for new CSV files.

If you want to go the regular scheduled route, you can specify the schedule\_interval in your DAG code, like what you see here. By the way, clicking on the schedule of one day here in the UI won't allow you to edit it there, but instead, will take you to the history

of all the runs for that workflow.

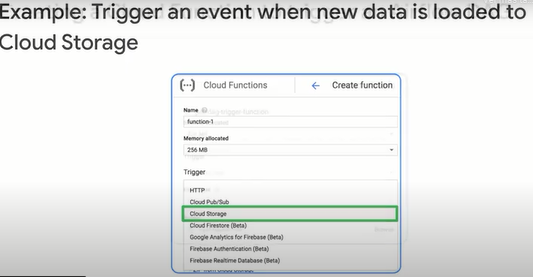


As you saw earlier in this course, there are two general patterns for ETL workflows: event triggered or push. As in, you push a new file to Cloud Storage and your workflow kicks off, or pull, which is where airflow at a set time could look in your Cloud storage folder, and take all the contents that are found for its workflow run.

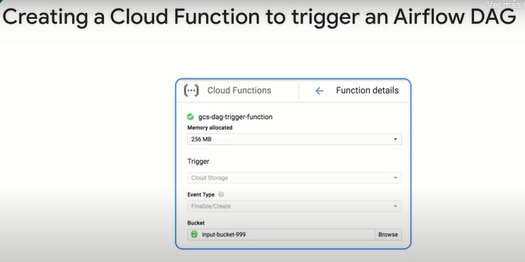


We can use Cloud Functions to create our event driven or push architecture workflow. I mentioned triggering on events within a Cloud Storage bucket, which you can also trigger based on HTTP requests, Pub Sub, FireStore, Firebase, and more as you see here.

Generally, push technology is great when wanting to distribute transactions as they happen. Stock tickers and other types of financial institution transactions are very important when it comes to push technology. How about disasters and notification? Again, important. For ML workflows, where your upstream data doesn't arrive at a regular pace, like get all the transactions at the end of each day, consider experimenting with a push architecture. Your final lab, since it's based on regular Google Analytics, news article data will be a pull architecture, but I've added in an optional lab for you to get practice with Cloud Functions and event-driven workflows for those interested. So let's talk through it more now. For our example, let's assume we have a CSV file or a set of files loaded to Cloud Storage. So we'll choose a Cloud storage trigger for our function.

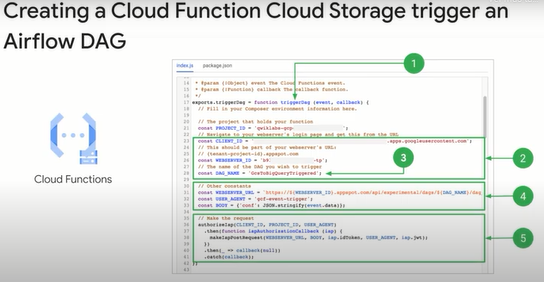


Then we specify an event type, finalize, create new files, and a bucket to watch.

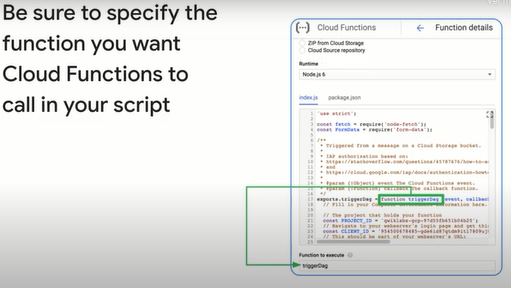


As part of the Cloud function, we need to create the actual function we want called in JavaScript. The good news is most of this code for triggering airflow DAGs in a function is all boilerplate for you to copy from as a starting point.

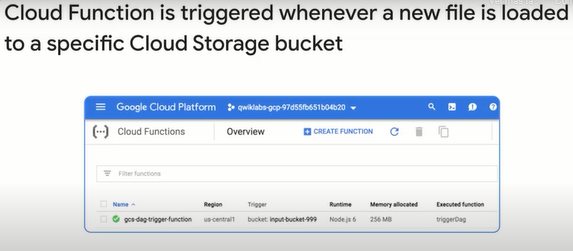
Here, we specify a name for our function called Trigger DAG, then we tell it where your airflow environment is to be triggered and which DAG in that airflow environment. In this case, it's looking for one called GCS to BigQuery triggered. Keep in mind, you can have multiple workflows or DAGs in a single airflow environment, so be sure you specify the correct DAG underscore name to trigger. Then we have a few constants that we are provided, which construct the airflow URL that we're going to trigger a post request to, as well as who's making the request and what the body of the request is. Lastly, the Trigger DAG function makes the actual request against the airflow server to kick off a workflow DAG.



Once you have the Cloud function code ready in your index.js file and the metadata about the function in package.json, which contains code dependency and versioning information, you still need to specify which function you actually want executed.



In this case, we created one called Trigger DAG. So we just copy that down. I'll also save you about 20 minutes of frustration and tell you that the function to execute box is case sensitive. So all capital letters, D-A-G is different from capital D, lowercase A and G.



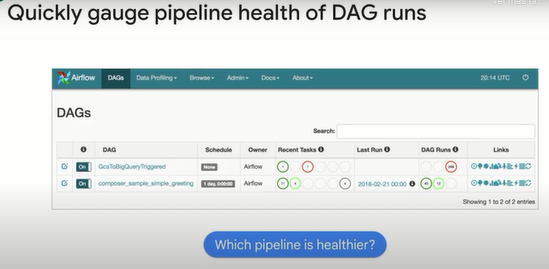
And there you have it.

Your Cloud function has been created, and is actively watching your Cloud storage bucket for file uploads. But how can you be sure everything is working as intended? For that, check out the next topic on monitoring and logging.

**Monitoring and Logging**

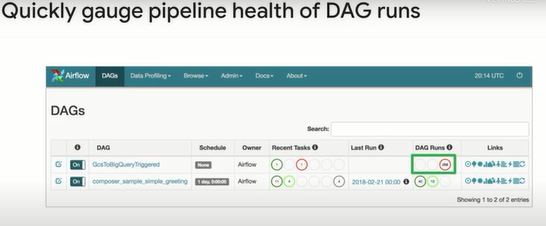
By this point, we've got our environment set up with our DAGs running at a predefined schedule or with triggered events. The last topic we'll cover before you practice what you've learned in your labs is how to monitor and troubleshoot your cloud functions and Airflow workflows. One of the most common reasons you'll want to investigate the historical runs of your DAGs is in the event that your workflow simply stops working. Note that you can have it auto retry for a set number of attempts in case it's a transient bug, but sometimes you can't get your workflow to run at all in the beginning.

In the DAG runs, you can monitor when your pipelines run and in what state, like success, running or failure. The quickest way to get to this page is clicking on the schedule for any of your DAGs from the main DAGs page. Here we have five successful runs over 5 days for this DAG, so this one seems to be running just fine. Back on the main page for DAGs, we see some red, which indicates trouble with some of our recent DAG runs.



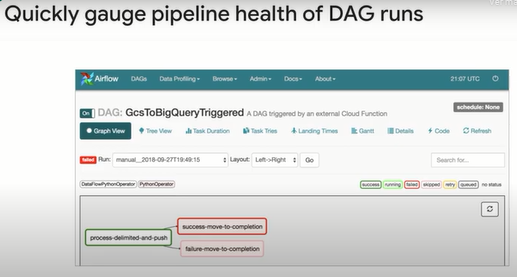
Speaking of DAG runs, you'll note the three circles below, which indicate how many runs passed, are currently active or have failed.

It certainly doesn't look good for 268 runs failed and zero passed for this first DAG.



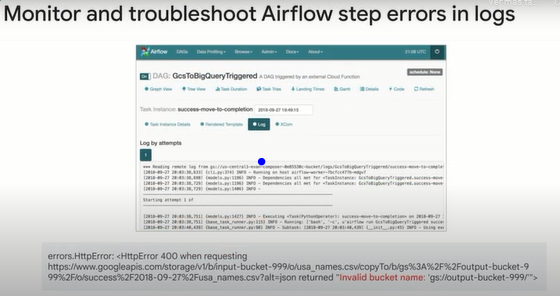
Let's see what happened.

We click on the name of the DAG to get to the visual representation. It looks like the first task is succeeding, judging by the green border, but the next task, success-move-to-completion is failing. Note that the lighter pink color for the failure-move-to-completion node means that node was skipped. So reading in to this a bit, the CSV file was correctly processed by Dataflow in the first task, but there was some issue moving the CSV file to a different cloud storage bucket as part of task two.

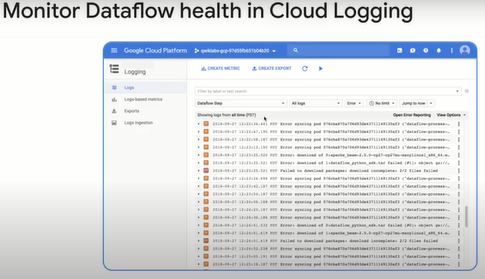


To troubleshoot, click on the node for a particular task and then click logs.

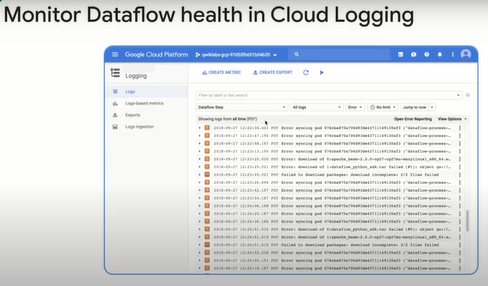
Here you will find the logs for that specific Airflow run. I search for the word error and then start my diagnosis there. Here, this was a pretty simple error where it was trying to copy a file from an input bucket to an output bucket, and the output bucket didn't exist or was named poorly.



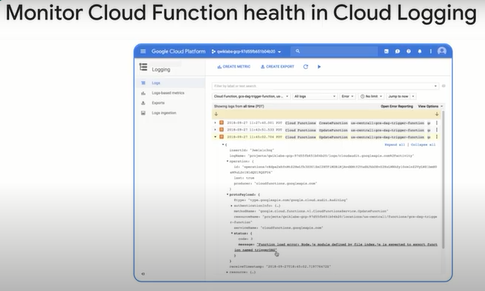
Another tool in your toolkit for diagnosing Airflow failures is the general Google Cloud logs.



Since Airflow launches other Google Cloud services through tasks, you can see and filter for errors for those services in Cloud Logging as you would debugging any other normal application. Here I filtered for Dataflow step errors to troubleshoot why my workflow is failing. It turns out that I had not changed the name of the output bucket for the CSV file, so after the file was processed by Dataflow as part of step one, it dumped the completed file back into the input bucket, which triggered another Dataflow job for processing and so on.

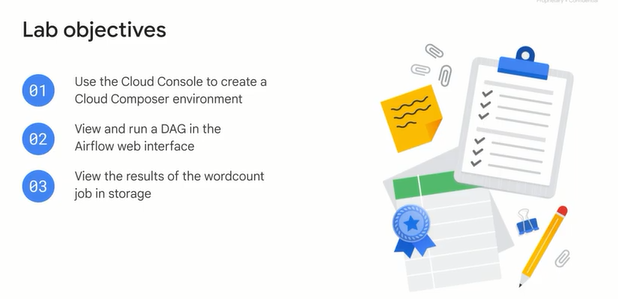


You might be wondering, if there's an error with my cloud function, my Airflow instance would never have been triggered or issued any logs at all since it was unaware we were trying to trigger it, and you're exactly right. If you're using cloud functions, be sure to check the normal Google Cloud Logs for errors and warnings in addition to your Airflow logs. In this example, each time I upload a CSV file to my cloud storage bucket hoping to trigger a cloud function and then my DAG, I get an error message that includes expected to export function named trigger DAG.



Remember way back when I said cloud functions were case sensitive? Looking for a function with capital DAG doesn't exist if it's capital D, lowercase A and G, so be sure to be mindful when setting up your cloud functions for the first time.

**Lab: An Introduction to Cloud Composer.**



Student Resource: <https://www.youtube.com/watch?v=GeNFEtt-D4k>

**Task 1. Ensure that the Kubernetes Engine API is successfully enabled**

To ensure access to the necessary APIs, restart the connection to the Kubernetes Engine API.

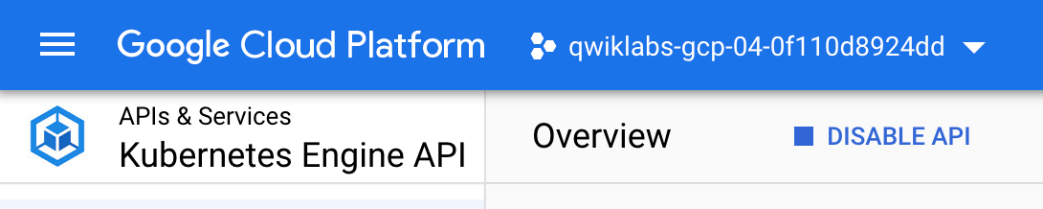
1. In the Google Cloud Console, enter **Kubernetes Engine API** in the top search bar. Click on the result for **Kubernetes Engine API**.
2. Click **Manage**.
3. Click **Disable API**.

If asked to confirm, click **Disable**.

If prompted Do you want to disable Kubernetes Engine API and its dependent APIs?, Click **Disable**.

1. Click **Enable**.

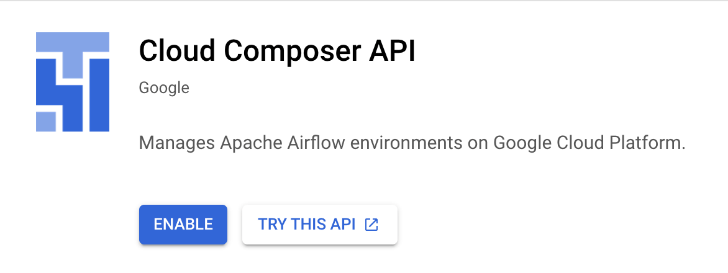
When the API has been enabled again, the page will show the option to disable.



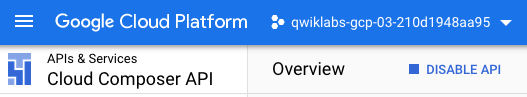
**Task 2. Ensure that the Cloud Composer API is successfully enabled**

Restart the connection to the Cloud Composer API. In the prior step, restarting the Kubernetes Engine API forced the Cloud Composer API to be disabled.

1. In the Google Cloud Console, enter **Cloud Composer API** in the top search bar. Click on the result for **Cloud Composer API**.
2. Click **Enable**.



When the API has been enabled, the page will show the option to disable.

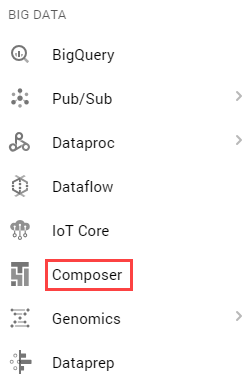


**Task 3. Create Cloud Composer environment**

In this section, you create a Cloud Composer environment.

Before proceeding further, make sure that you have performed earlier tasks to ensure that the required APIs are successfully enabled. If not, then please perform those tasks otherwise Cloud Composer environment creation will fail.

1. Go to **Navigation menu** > **Composer**:



1. Click **Create Environment** and select **Composer 1**. Set the following for your environment:

|  |  |
| --- | --- |
| **Property** | **Value** |
| **Name** | highcpu |
| **Location** | us-central1 |
| **Zone** | us-central1-a |
| **Machine type** | n1-highcpu-4 |

Leave all other settings as default.

1. Click **Create**.

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 can take 10-20 minutes for the environment to complete the setup process. Continue with the lab while the environment spins up.

Click **Check my progress** to verify the objective.

**Create a Cloud Storage bucket**

Create a Cloud Storage bucket in your project. This buckets will be used as output for the Hadoop job from Dataproc.

1. Go to **Navigation menu** > **Cloud Storage** > **Browser** and then click **Create bucket**.
2. Give your bucket a universally unique name, then click **Create**.

Remember the Cloud Storage bucket name as you'll use it as an Airflow variable later in the lab.

Click **Check my progress** to verify the objective.

**Task 4. Airflow and core concepts**

While waiting for your Composer environment to get created, review some terms that are used with Airflow.

[Airflow](https://airflow.apache.org/) is a platform to programmatically author, schedule and monitor workflows.

Use Airflow to author workflows as directed acyclic graphs (DAGs) of tasks. The airflow scheduler executes your tasks on an array of workers while following the specified dependencies.

**Core concepts**

[DAG](https://airflow.apache.org/docs/apache-airflow/stable/concepts/dags.html)

A Directed Acyclic Graph is a collection of all the tasks you want to run, organized in a way that reflects their relationships and dependencies.

[Operator](https://airflow.apache.org/docs/apache-airflow/stable/concepts/operators.html)

The description of a single task, it is usually atomic. For example, the *BashOperator* is used to execute bash command.

[Task](https://airflow.apache.org/docs/apache-airflow/stable/concepts/tasks.html)

A parameterised instance of an Operator; a node in the DAG.

[Task Instance](https://airflow.apache.org/docs/apache-airflow/stable/concepts/tasks.html#task-instances)

A specific run of a task; characterized as: a DAG, a Task, and a point in time. It has an indicative state: *running*, *success*, *failed*, *skipped*, ...

You can read more about the concepts [here](https://airflow.apache.org/concepts.html).

**Task 5. Defining the workflow**

Now let's discuss the workflow you'll be using. Cloud Composer workflows are comprised of [DAGs (Directed Acyclic Graphs)](https://airflow.apache.org/docs/apache-airflow/stable/concepts/dags.html" \t "_blank). DAGs are defined in standard Python files that are placed in Airflow's DAG\_FOLDER. Airflow will execute the code in each file to dynamically build the DAG objects. You can have as many DAGs as you want, each describing an arbitrary number of tasks. In general, each one should correspond to a single logical workflow.

Below is the hadoop\_tutorial.py workflow code, also referred to as the DAG:

"""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 used as output for the Hadoop jobs from Dataproc.

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.

wordcount\_args = ['wordcount', 'gs://pub/shakespeare/rose.txt', 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')

}

with models.DAG(

'composer\_sample\_quickstart',

# Continue to run DAG once per day

schedule\_interval=datetime.timedelta(days=1),

default\_args=default\_dag\_args) as dag:

# 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,

region='us-central1',

zone=models.Variable.get('gce\_zone'),

image\_version='2.0',

master\_machine\_type='n1-standard-2',

worker\_machine\_type='n1-standard-2')

# Run the Hadoop wordcount example installed on the Cloud Dataproc cluster

# master node.

run\_dataproc\_hadoop = dataproc\_operator.DataProcHadoopOperator(

task\_id='run\_dataproc\_hadoop',

region='us-central1',

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',

region='us-central1',

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)

# Define DAG dependencies.

create\_dataproc\_cluster >> run\_dataproc\_hadoop >> delete\_dataproc\_cluster

To orchestrate the three workflow tasks, the DAG imports the following operators:

1. DataprocClusterCreateOperator: Creates a Cloud Dataproc cluster.
2. DataProcHadoopOperator: Submits a Hadoop wordcount job and writes results to a Cloud Storage bucket.
3. DataprocClusterDeleteOperator: Deletes the cluster to avoid incurring ongoing Compute Engine charges.

The tasks run sequentially, which you can see in this section of the file:

# Define DAG dependencies.

create\_dataproc\_cluster >> run\_dataproc\_hadoop >> delete\_dataproc\_cluster

The name of the DAG is quickstart, and the DAG runs once each day.

with models.DAG(

'composer\_sample\_quickstart',

# Continue to run DAG once per day

schedule\_interval=datetime.timedelta(days=1),

default\_args=default\_dag\_args) as dag:

Because the start\_date that is passed in to default\_dag\_args is set to yesterday, Cloud Composer schedules the workflow to start immediately after the DAG uploads.

**Task 6. Viewing environment information**

1. Go back to **Composer** to check the status of your environment.
2. Once your environment has been created, click the name of the environment (highcpu) to see its details.

On the **Environment details** you'll see information such as the Airflow web interface URL, Kubernetes Engine cluster ID, and a link to the DAGs folder, which is stored in your bucket.

**Note:** Cloud Composer only schedules the workflows in the /dags folder.

**Task 7. Using the Airflow UI**

To access the Airflow web interface using the GCP Console:

1. Go back to the **Environments** page.
2. In the **Airflow webserver** column for the environment, click **Airflow**.
3. Click on your lab credentials.
4. The Airflow web interface opens in a new browser window.

**Task 8. Setting Airflow variables**

Airflow variables are an Airflow-specific concept that is distinct from [environment variables](https://cloud.google.com/composer/docs/how-to/managing/environment-variables" \t "_blank).

1. Select **Admin** > **Variables**. Under *List Variable*, to add a new record click **+** icon.



1. Create the following Airflow variables: gcp\_project, gcs\_bucket, and gce\_zone and click **Save** after each variable.

|  |  |  |
| --- | --- | --- |
| **Key** | **Val** | **Details** |
| gcp\_project | <your project-id> | The Google Cloud Platform project you're using for this quickstart. |
| gcs\_bucket | gs://<my-bucket> | Replace <my-bucket> with the name of the Cloud Storage bucket you made earlier. This bucket stores the output from the Hadoop jobs from Dataproc. |
| gce\_zone | us-central1-a | This is the Compute Engine zone where your Cloud Dataproc cluster will be created. To chose a different zone, see [Available regions & zones.](https://cloud.google.com/compute/docs/regions-zones/regions-zones" \l "available" \t "_blank) |

Your Variables table should look like this when you're finished:



**Task 9. Uploading the DAG to Cloud Storage**

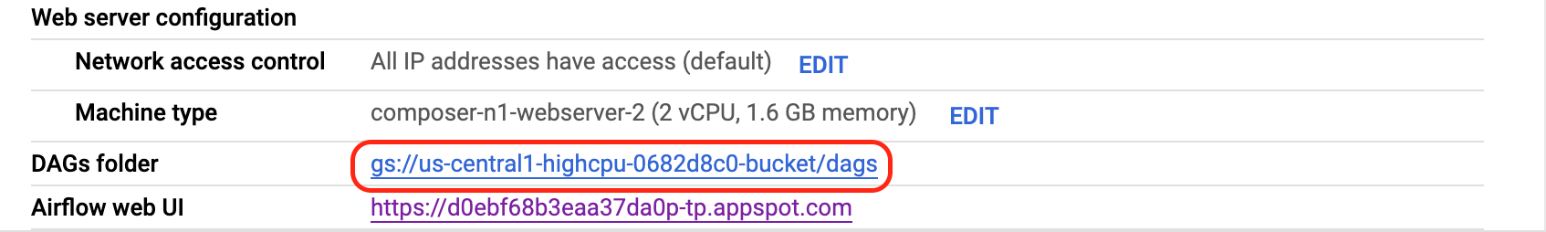
To upload the DAG:

1. In Cloud Shell, upload a copy of the hadoop\_tutorial.py file to the Cloud Storage bucket that was automatically created when you created the environment.

Replace <DAGs\_folder\_path> in the following command:

gsutil cp gs://cloud-training/datawarehousing/lab\_assets/hadoop\_tutorial.py <DAGs\_folder\_path>

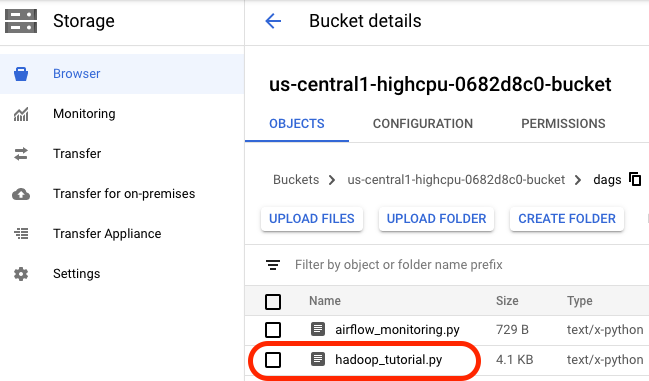
You can get the path by going to **Composer**. Click on the environment you created earlier and then click on the **Environment Configuration** tab to see the details of the environment. Find DAGs folder and copy the path.



The revised command to upload the file will look similar to the one below:

gsutil cp gs://cloud-training/datawarehousing/lab\_assets/hadoop\_tutorial.py gs://us-central1-highcpu-0682d8c0-bucket/dags

Once the file has been successfully uploaded to the DAGs directory, open dags folder in the bucket and you will see the file in the **Objects** tab of the Bucket details.



When a DAG file is added to the DAGs folder, Cloud Composer adds the DAG to Airflow and schedules it automatically. DAG changes occur within 3-5 minutes.

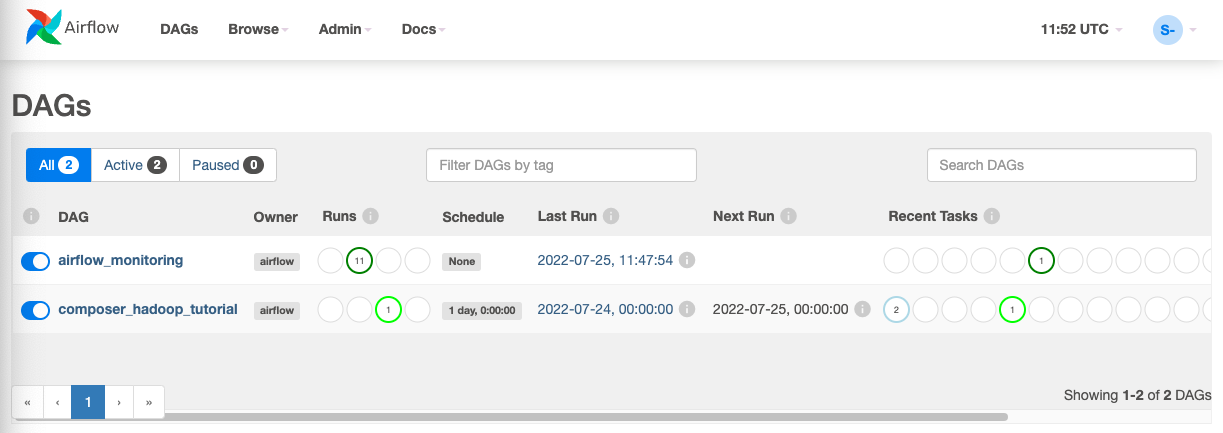
You can see the task status of the composer\_hadoop\_tutorial DAG in the Airflow web interface.

Click **Check my progress** to verify the objective.

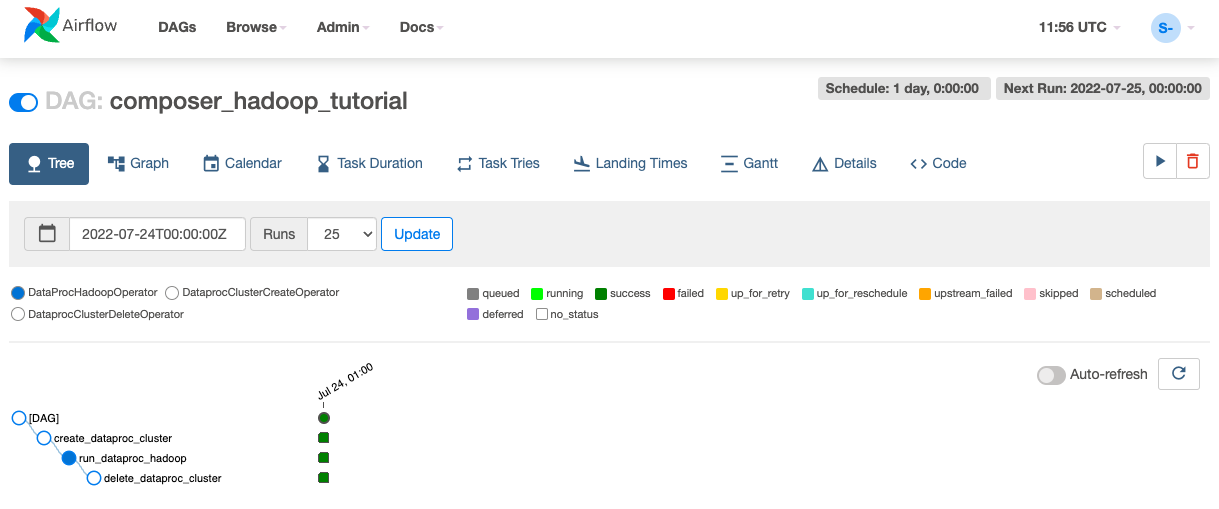
***Exploring DAG runs***

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.

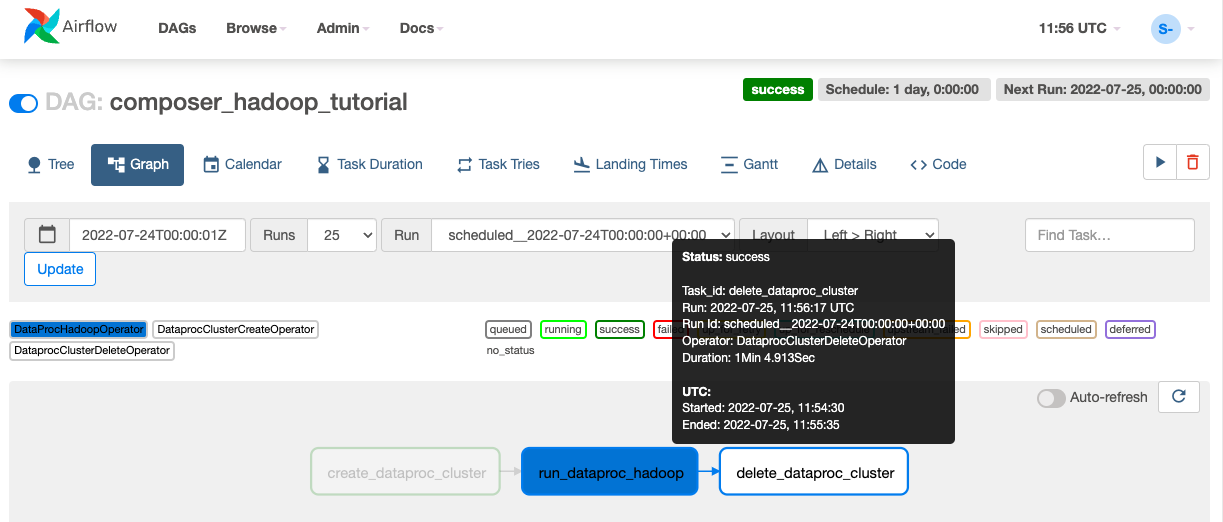
Make sure that you're on the DAGs tab in the Airflow web interface. It takes several minutes for this process to complete. Refresh your browser to make sure you're looking at the latest information.



1. In Airflow, click **composer\_hadoop\_tutorial** to open the DAG details page. This page includes several representations of the workflow tasks and dependencies.



1. In the toolbar, click **Graph**. 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.).



1. Click the "Refresh" link to make sure you're looking at the most recent information. The borders of the processes change colors as the state of the process changes

**Note**: If your Dataproc cluster already exists, you can run the workflow again to reach the success state by clicking **create\_dataproc\_cluster** graphic and then click **Clear** to reset the three tasks and click **OK** to confirm.

1. Once the status for **create\_dataproc\_cluster** has changed to "running", go to **Navigation menu** > **Dataproc**, then click on:

* **Clusters** to monitor cluster creation and deletion. 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.
* **Jobs** to monitor the Apache Hadoop wordcount job. Click the Job ID to see job log output.

1. Once Dataproc gets to a state of "Running", return to Airflow and click **Refresh** to see that the cluster is complete.

When the run\_dataproc\_hadoop process is complete, go to **Navigation menu** > **Cloud Storage** > **Browser** and click on the name of your bucket to see the results of the wordcount in the wordcount folder.

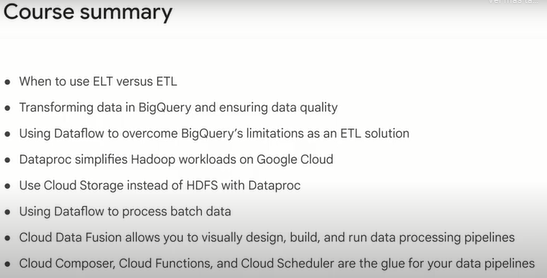
1. Once all the steps are complete in the DAG, each step has a dark green border. Additionally the Dataproc cluster that was created is now deleted.

**Congratulations!**

**Course Summary.**

You've made it to the end of this course on building batch data pipelines.

Let's recap what you've learned.



In this course, we covered the different methods of loading data into your data lakes and warehouses.

Remember ELT versus ETL and when to use each.

ELT of extract, load, transform is a common pattern for when the transformations you want on the data set are minor and then can be handled after you load the data.

An example is loading the data into BigQuery first and then doing SQL on the raw data and storing the transformed version as a new table.

On the other hand, ETL, or transforming before loading, is common when you have data that needs more complex transformations, where the sheer volume of data makes it better to do those transformations before loading.

That's where you would want to use a batch pipeline like Dataflow.

Next, we discussed how to transform data in BigQuery using SQL, and you saw the common operations you can perform to ensure your structured data is ready for insights.

Lastly, you practiced building batch pipelines in the serverless way with Dataflow.

In our Hadoop module, we discussed dataproc in detail.

You learned that you can lift and shift your existing Hadoop workloads to the cloud with no code changes, and they will just work.

However, once in the cloud, there were additional optimizations you could make, like using cloud storage for cluster storage instead of HDFS for greater efficiency and cost savings.

We then used Dataflow to build batch data pipelines using Dataflow templates and by writing them ourselves.

Recall that the fundamental unit of logical data in the pipeline is the PCollection, which stands for a parallel collection.

Dataflow will automatically split up your data set into many pieces and farm out the processing across as many worker VMs as it needs to complete the job.

Dataflow is a serverless application, which means you will have some control over the maximum number of workers, the actual processing work and the autoscaling of workers up and down as the demand requires.

In our module on managing data pipelines with Google Cloud, we discussed two new tools: Cloud Data Fusion and Cloud Composer.

Recall that Cloud Data Fusion allows data analysts and ETL developers to wrangle data and build pipelines in a visual way.

The technology then builds and executes the pipelines on a runner.

With Cloud Data Fusion, you also get access to the lineage of each data field, which is the series of any transformation logic that happened before and after the field reaches your data set.

The second tool we covered is Cloud Composer as a workflow orchestrator.

Cloud Composer is managed Apache Airflow and allows you to, at a high level, command Google Cloud services in a DAG to perform complex operations.

These DAGs can be user scheduled or event driven with cloud functions.

Recall the example where when a new CSV was uploaded to our cloud storage bucket, a cloud function was triggered to start a Dataflow pipeline for processing and then sync the data into BigQuery.

Congratulations on completing building batch data pipelines on Google Cloud.

Building resilient streaming analytics systems on Google Cloud is the third course of the data engineering on Google Cloud core series and is covered next.

We hope to see you there.

**Quizz: Manage Data Pipelines with Cloud Data Fusion and Cloud Composer.**

Cloud Data Fusion is the ideal solution when you need

a data warehousing solution

to reuse spark pipelines

**to build visual pipelines**

low-latency and high throughput processing of streaming data