Resource-Efficient Data Pipeline Management with Airflow and Big Query Integration

THIRUVENI B

2320446

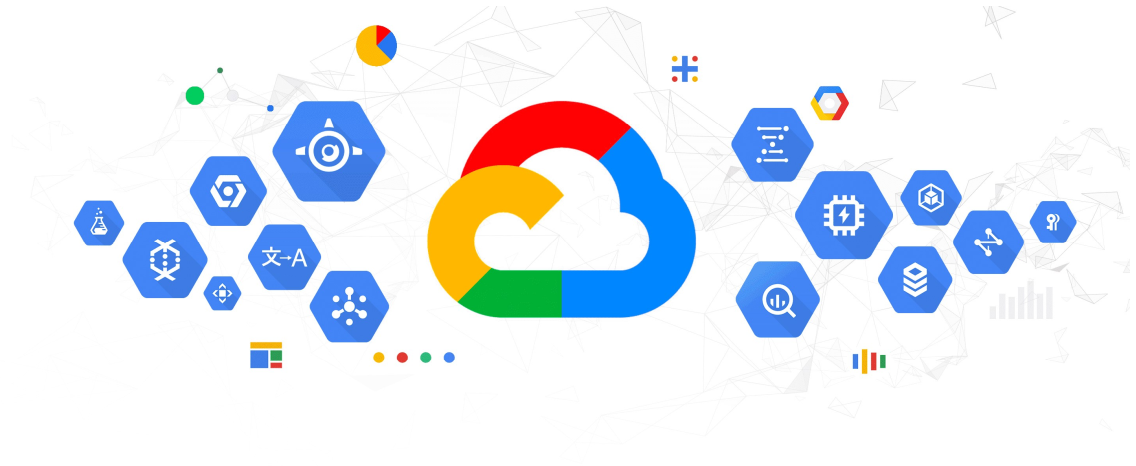
CSDAIA24GP003

ABSTRACT

This project presents a comprehensive approach to building a resource-efficient composer and orchestrating data pipelines using Airflow with Python code. The primary objective is to develop a streamlined workflow that minimizes resource usage while effectively managing data processing tasks. The project involves creating a composer with minimal resource requirements, establishing an Airflow Directed Acyclic Graph (DAG) to automate data pipeline operations, and integrating with Google Cloud Storage to manage files within a bucket. Additionally, the system includes scheduled tasks to check for new files in the bucket and load them into Big Query for further analysis. By leveraging Airflow's flexibility and scalability alongside Big Query's powerful data analytics capabilities, this project aims to provide a robust solution for managing data pipelines in a resource-constrained environment.

GOOGLE CLOUD PLATFORM (GCP)

Google Cloud Platform (GCP) offers a variety of services including computing, storage, databases, machine learning, and more. Key services provided by GCP include Compute Engine, Google Kubernetes Engine (GKE), Cloud Storage, Big Query, and Cloud Composer. GCP provides advantages such as global infrastructure, security, managed services, and integration with other Google services like Gmail, YouTube, and Google Search.



WHAT IS AIRFLOW!

Airflow is an open-source platform used for orchestrating complex workflows and data pipelines. Developed by Airbnb, it provides a programmable way to author, schedule, and monitor workflows. Airflow's core components include Directed Acyclic Graphs (DAGs) and Operators, which allow users to define workflows as code and execute tasks in a distributed environment.

COMPOSER

Cloud Composer is a fully managed service for Apache Airflow provided by Google Cloud Platform (GCP). Composer simplifies the deployment and management of Airflow workflows, offering scalability, reliability, and integration with other GCP services.

Why Composer for Workflow Orchestration

- Composer offers several advantages over self-hosted Airflow installations, including managed infrastructure, automatic updates, and integration with other GCP services.
- Compared to other workflow orchestration tools, Composer provides a familiar interface for Airflow users and seamless integration with GCP services.
- Composer enables organizations to focus on building and managing workflows without worrying about infrastructure maintenance or scalability issues.

Creating Buckets in Google Cloud Storage (GCS)

- Google Cloud Storage (GCS) is a scalable object storage service for storing and retrieving data on Google Cloud Platform.
- Buckets are containers for storing objects (files) in GCS, with each bucket having a unique name and configurable settings such as access control and storage class.

- Steps for creating buckets in GCS include selecting a unique name, choosing a storage class (e.g., Standard, Nearline, Coldline), and configuring access permissions.

Integrating Composer with Google Cloud Storage and Big Query

Composer can interact with Google Cloud Storage (GCS) and BigQuery using Airflow operators and hooks. Configuration steps for integrating Composer with GCS and BigQuery include setting up connections in the Airflow UI, specifying authentication credentials, and defining task dependencies. Access to GCS and Big Query resources is managed through service accounts and IAM roles in GCP, ensuring secure and controlled access to data.

Project Implementation Overview

- Project architecture includes Composer for workflow orchestration, GCS for storing input files, and BigQuery for data analysis and querying.
- Workflow tasks involve checking files in GCS at scheduled intervals and loading them into BigQuery for analysis.
- Components such as DAGs (Directed Acyclic Graphs), operators, and sensors are used in Airflow to define and execute workflow tasks.

AIRFLOW WEB UI

Checking DAG Web Server Status:

There are two main ways to check the health of your Airflow DAG web server:

1. **Airflow Web UI:** If your Airflow web server is running, you can access the UI and view the health status information. The URL will typically be http://<your-webserver-address>:<port>/admin/airflow/health. This page should display the status of various components like the scheduler, meta database, dag processor, and triggerer.
2. **Health Check Endpoint:** Airflow also provides a health check endpoint (if configured) that you can use to programmatically check the status. The specific URL and method for this endpoint might vary depending on your setup. You can consult your Airflow documentation for details.

Color Coding in the Web UI:

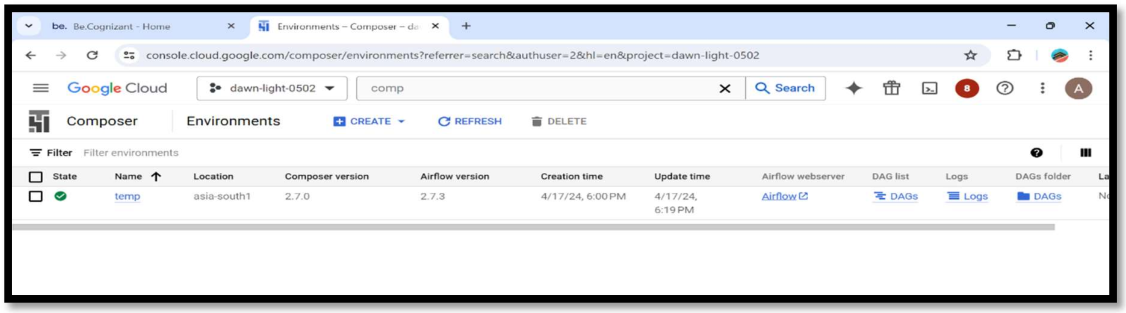
The Airflow web UI often uses color coding to represent the status of DAGs and other components. Here's a general explanation of the common colors:

- **Green:** This usually indicates a healthy and functioning state. For a DAG, it might mean it's up-to-date and ready to run.
- **Yellow:** This could represent a warning state. For a DAG, it might signify a scheduled run is upcoming or there are some configuration issues.
- **Red:** This typically indicates an error or unhealthy state. A DAG might be marked red if there are errors in its definition or recent runs failed.
- **Gray:** This could represent an inactive or unknown state.

The specific color-coding scheme might vary depending on your Airflow version and customization. The web UI usually provides tooltips or hover text that explains the meaning behind each color when you hover over a DAG or component.

STEP 1:

CREATING COMPOSER

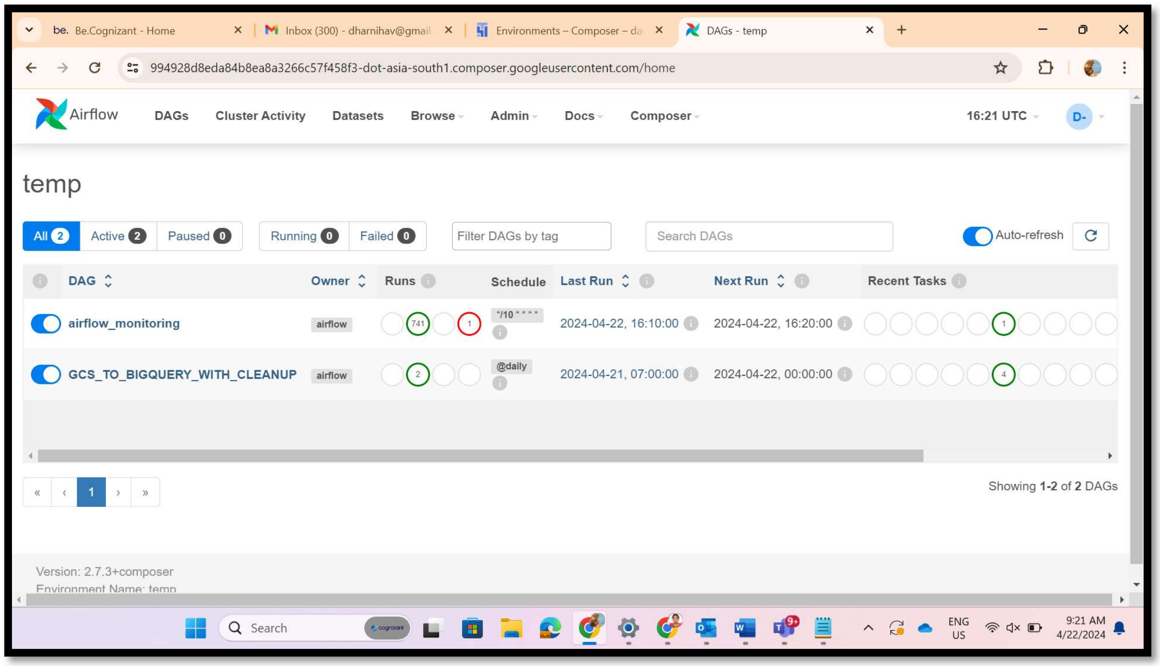

be. Be.Cognizant - Home
Environments – Composer – da
console.cloud.google.com/composer/environments?referrer=search&authuser=2&hl=en&project=dawn-light-0502
Google Cloud
dawn-light-0502
comp
Search
Composer
Environments
CREATE
REFRESH
DELETE
Filter
Filter environments

State	Name	Location	Composer version	Airflow version	Creation time	Update time	Airflow webserver	DAG list	Logs	DAGs folder
✓	temp	asia-south1	2.7.0	2.7.3	4/17/24, 6:00 PM	4/17/24, 6:19 PM	Airflow	DAGs	Logs	DAGs

Creating the composer in the google console using the method of GUI with minimum resources in the location of Mumbai (Asia-south 1) in the version of 2.7.3 as the airflow version and the web UI of the airflow and the Dag folder where we can upload the Dag file that is python file.

STEP 2:

AIRFLOW WEB UI



DAG	Owner	Runs	Schedule	Last Run	Next Run	Recent Tasks
airflow_monitoring	airflow	741, 1	*/10 * * * *	2024-04-22, 16:10:00	2024-04-22, 16:20:00	1
GCS_TO_BIGQUERY_WITH_CLEANUP	airflow	2	@daily	2024-04-21, 07:00:00	2024-04-22, 00:00:00	4

Showing 1-2 of 2 DAGs

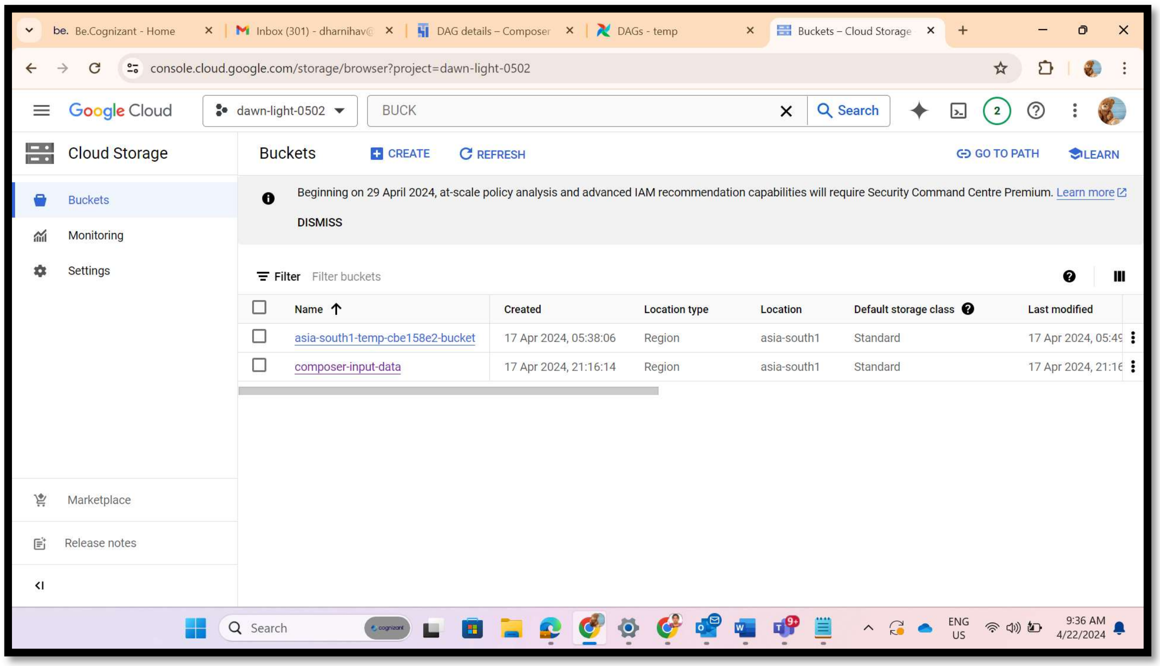
Version: 2.7.3+composer
Environment Name: temp

We can see the Airflow web page that mentions the number of runs with the default dags and also can see the graph of the dag that is running.

STEP 3:

CREATE BUCKET

Creating the bucket in the GUI method to upload the input files.

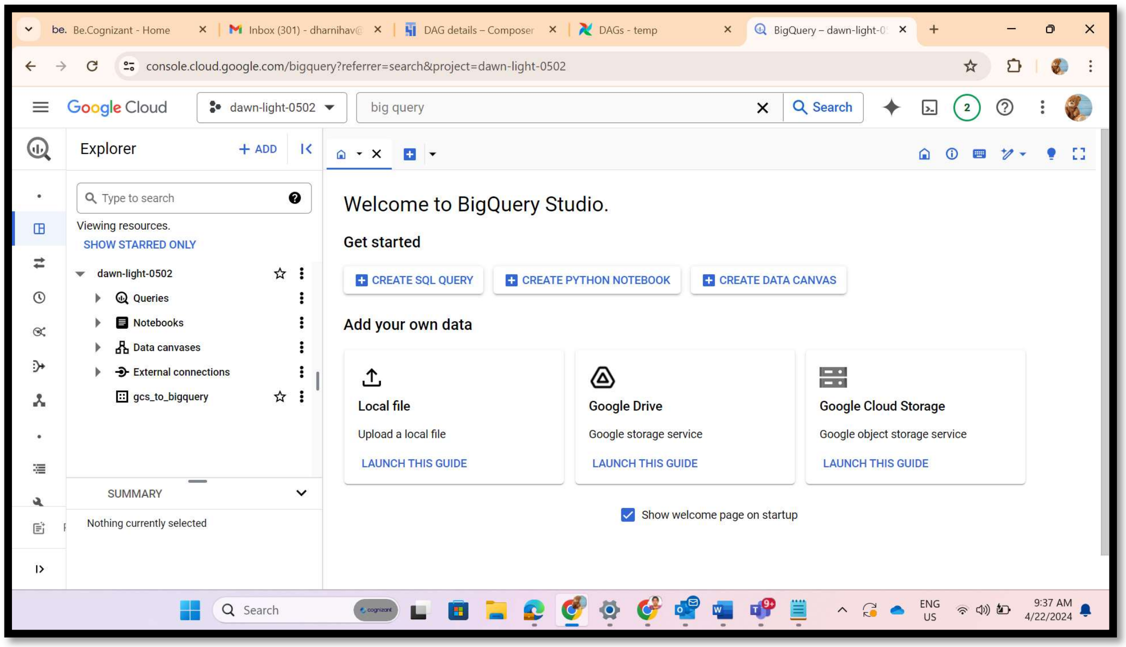


Beginning on 29 April 2024, at-scale policy analysis and advanced IAM recommendation capabilities will require Security Command Centre Premium. Learn more

Name	Created	Location type	Location	Default storage class	Last modified
asia-south1-temp-cbe158e2-bucket	17 Apr 2024, 05:38:06	Region	asia-south1	Standard	17 Apr 2024, 05:49
composer-input-data	17 Apr 2024, 21:16:14	Region	asia-south1	Standard	17 Apr 2024, 21:16

STEP 4

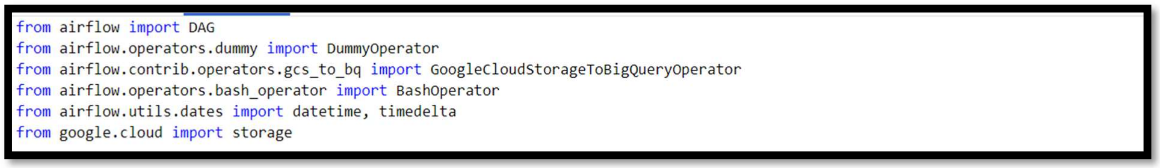
CREATE BIG QUERY DATASET named as ‘gcs_to_bigquery’.



STEP 5

WRITE DAG

Importing the necessary DAG and Operators from the Airflow library to define the workflows.

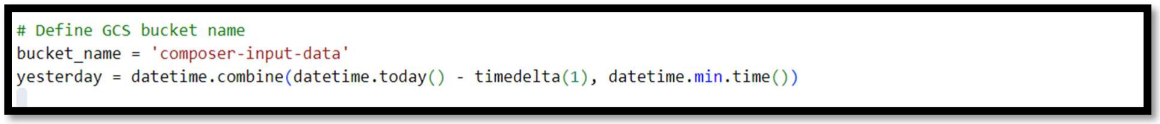


```
from airflow import DAG
from airflow.operators.dummy import DummyOperator
from airflow.contrib.operators.gcs_to_bq import GoogleCloudStorageToBigQueryOperator
from airflow.operators.bash_operator import BashOperator
from airflow.utils.dates import datetime, timedelta
from google.cloud import storage
```

Importing Libraries (Lines 1-5):

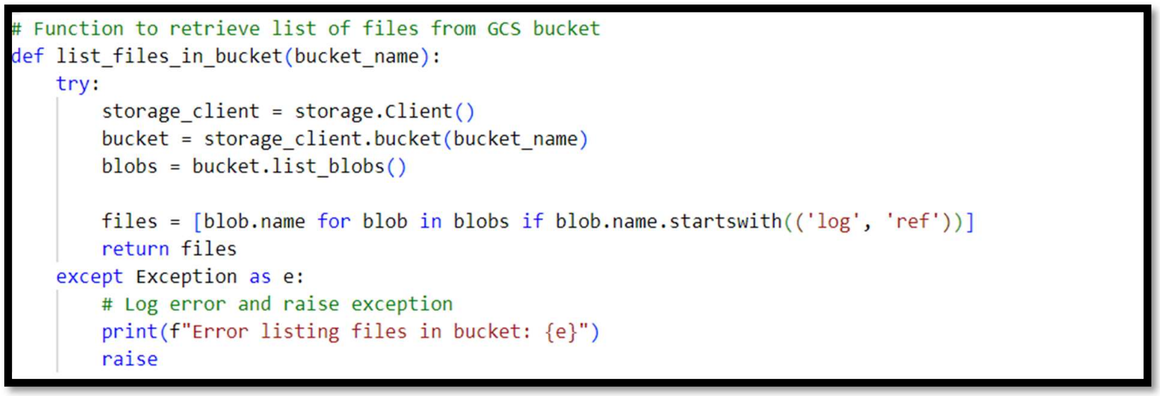
- Imports the DAG class from the Airflow library, which is used to define workflows.
- Imports the DummyOperator class, used to create placeholder tasks in the DAG that do not perform any actions but help with flow control.
- Imports the GoogleCloudStorageToBigQueryOperator class, used to transfer data from GCS to Big Query tables.
- Imports the BashOperator class, used to run bash commands within the DAG.
- Imports the datetime and timedelta classes from Airflow's date utility functions.
- Imports the storage library from Google Cloud to interact with GCS.

Defining the GCS bucket name:



```python
# Define GCS bucket name
bucket_name = 'composer-input-data'
yesterday = datetime.combine(datetime.today() - timedelta(1), datetime.min.time())
```

Defines a variable bucket name and assigns the string value 'composer-input-data'. This variable stores the name of the GCS bucket from which the data files will be loaded.



```python
# Function to retrieve list of files from GCS bucket
def list_files_in_bucket(bucket_name):
    try:
        storage_client = storage.Client()
        bucket = storage_client.bucket(bucket_name)
        blobs = bucket.list_blobs()

        files = [blob.name for blob in blobs if blob.name.startswith(('log', 'ref'))]
        return files
    except Exception as e:
        # Log error and raise exception
        print(f"Error listing files in bucket: {e}")
        raise
```

This code defines a function named list_files_in_bucket that retrieves a list of files from a specified GCS bucket, filtering for files with desired prefixes.

Function Definition:

- def list_files_in_bucket(bucket_name): This line defines a function named list_files_in_bucket that takes a single argument, bucket_name, which is a string representing the name of the GCS bucket to list files from.

Connecting to GCS:

- storage_client = storage.Client(): This line creates a client object called storage_client using the storage library from Google Cloud. This client object allows interaction with the GCS service.

Getting Bucket Object:

- bucket = storage_client.bucket(bucket_name):
  This line uses the storage_client object to get a specific bucket object from GCS. It calls the bucket method on the client, passing the bucket_name argument to retrieve the corresponding bucket object.

Listing Blobs:

- blobs = bucket.list_blobs(): This line calls the list_blobs method on the bucket object. This method retrieves a list of all blobs (objects) stored within the specified GCS bucket. Each blob represents a file in the bucket.

Filtering Files:

- files = [blob.name for blob in blobs if blob.name.startswith(('log', 'ref'))]: This line creates a list named files to store the filtered filenames. It uses a list comprehension to iterate through each blob (file) in the blobs list. Inside the loop:
    - blob.name: This accesses the filename of the current blob (file).
    - startswith(('log', 'ref')): This checks if the filename starts with either "log" or "ref" (assuming these are the prefixes for your data files) using string startswith method with a tuple containing both prefixes.
    - If the filename starts with "log" or "ref", the entire filename (blob.name) is appended to the files list.

Returning Results:

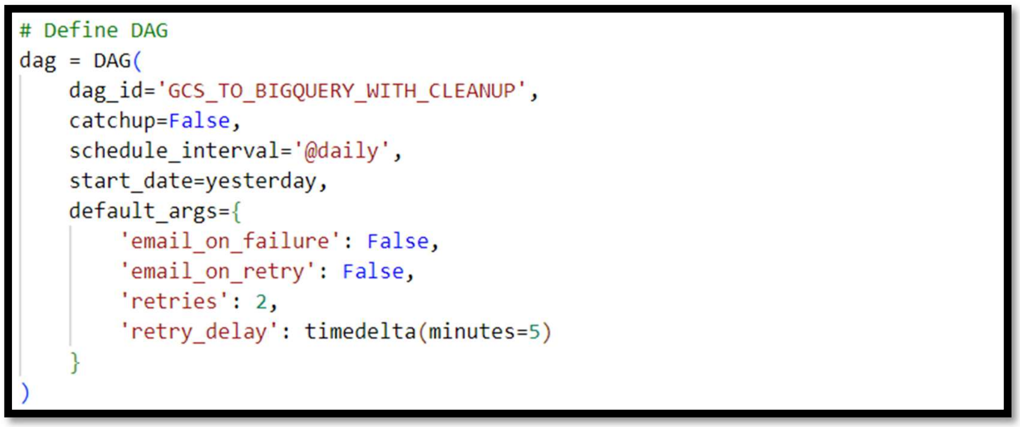
- return files: After iterating through all blobs and filtering, this line returns the files list containing the filenames that meet the starting prefix criteria ("log" or "ref").

Except Block:

- except Exception as e: This block defines an exception handler that catches any general exceptions (Exception) that might occur within the try block.
    - print (f"Error listing files in bucket: {e}"): If an exception is caught, this line prints an error message containing details about the exception using f-strings. The f"{...}" allows embedding the exception variable e within the message.
    - raise: This line re-raises the caught exception (e). This ensures that the error propagates further, and the calling code can handle it appropriately.

Overall, this function attempts to connect to GCS, list all files from the specified bucket, filter for files with desired prefixes ("log" or "ref"), and return a list of those filenames. If any errors occur during the process, it logs the error and re-raises it for further handling.

Defining the DAG to start and end the process:



```
# Define DAG
dag = DAG(
    dag_id='GCS_TO_BIGQUERY_WITH_CLEANUP',
    catchup=False,
    schedule_interval='@daily',
    start_date=yesterday,
    default_args={
        'email_on_failure': False,
        'email_on_retry': False,
        'retries': 2,
        'retry_delay': timedelta(minutes=5)
    }
)
```

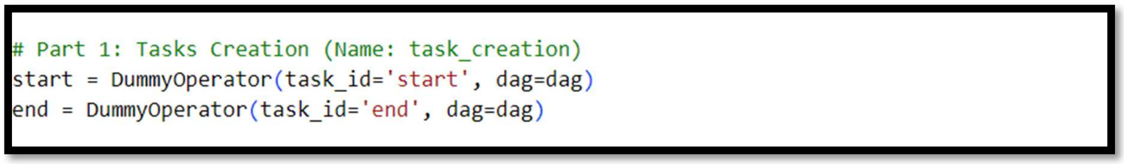
This code defines a Directed Acyclic Graph (DAG) in Airflow named GCS_TO_BIGQUERY_WITH_CLEANUP.

1. **dag = DAG(...)**: This line creates a DAG object named dag using the DAG class imported from Airflow. The DAG class is used to define workflows or pipelines that consist of tasks.
2. **dag_id ='GCS_TO_BIGQUERY_WITH_CLEANUP'<**: This argument specifies the ID of the DAG. Here, it's set to "GCS_TO_BIGQUERY_WITH_CLEANUP", which helps identify the DAG within Airflow.
3. **catchup=False**: This argument controls whether the DAG should backfill data for missed scheduled runs. Here, it's set to False, meaning the DAG will only run for the current day and future days based on the schedule. It won't attempt to process data for any previous days it might have missed.
4. **schedule_interval='@daily'**: This argument defines the scheduling interval for the DAG. Here, it's set to '@daily', which tells Airflow to run the DAG once every day. Airflow provides various scheduling options, and @daily is a common choice for daily tasks.
5. **start_date=yesterday**: This argument specifies the start date for the DAG. Here, it's set to the yesterday variable, which we saw previously calculates the date for the previous day (including midnight). This ensures the DAG starts processing data from yesterday onwards.
6. **default_args={ ... }**: This argument defines a dictionary of default arguments that will be applied to all tasks within the DAG unless explicitly overridden by individual tasks. Here, the dictionary includes:

- 'email_on_failure': False - Disables sending emails when tasks fail.
- 'email_on_retry': False - Disables sending emails when tasks are retried.
- 'retries': 2 - Sets the number of times failed tasks will be retried (here, twice).
- 'retry_delay': timedelta(minutes=5) - Defines the delay between retries in case a task fails (here, 5 minutes).

Overall, this code snippet configures a DAG named GCS_TO_BIGQUERY_WITH_CLEANUP to run daily, starting from yesterday, with specific settings for retries and email notifications.

Creating the start and end task within the DAG:



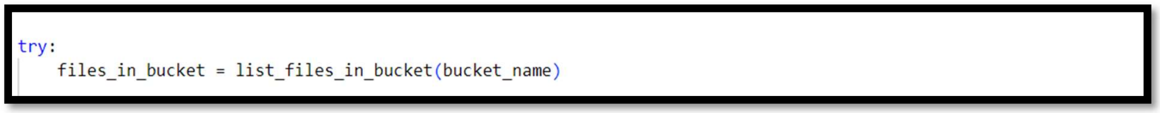
```
# Part 1: Tasks Creation (Name: task_creation)
start = DummyOperator(task_id='start', dag=dag)
end = DummyOperator(task_id='end', dag=dag)
```

This code creates two dummy operators named start and end within the DAG (dag) you defined earlier. Let's break it down:

1. **start = DummyOperator(task_id='start', dag=dag)**: Dummy operators are placeholder tasks that don't perform any specific actions but serve as control points within the DAG to define workflow structure.
   - task_id='start': This argument assigns the ID "start" to the dummy operator. Task IDs are unique identifiers within a DAG and help reference specific tasks.
   - dag=dag: This argument specifies the DAG object (dag) to which this operator belongs. This associates the start operator with the overall DAG you defined previously.
2. **end = DummyOperator (task_id='end', dag=dag)**: This line is similar to the previous one, but it creates another dummy operator named end. This operator likely signifies the successful completion of the DAG workflow.
   - task_id='end': This assigns the ID "end" to the dummy operator.
   - dag=dag: This again associates the end operator with the dag object.

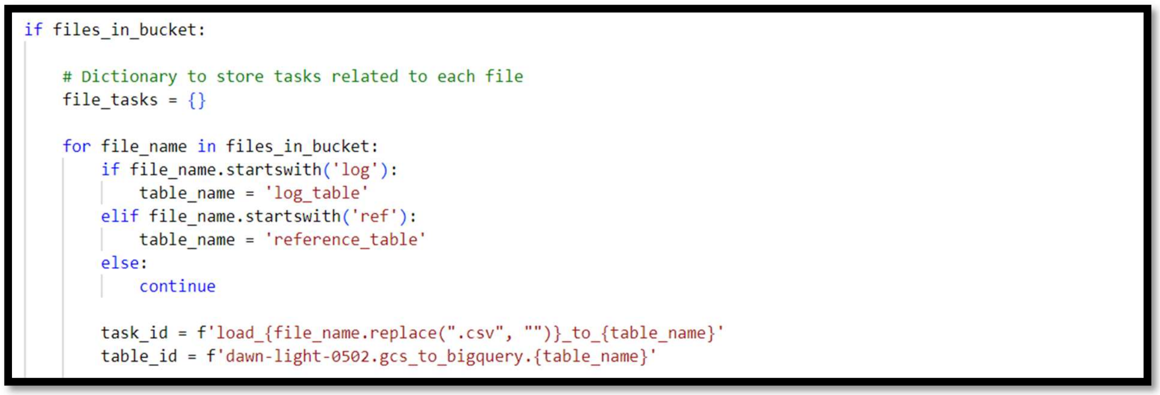
By creating these dummy operators, you're establishing the starting and ending points of your DAG workflow. These operators won't perform any actions, but they help visualize the flow of tasks within the DAG and potentially serve as control points for branching or conditional execution in more complex workflows.

List the files in bucket



```python
try:
    files_in_bucket = list_files_in_bucket(bucket_name)
```

This line of code attempts to call the list_files_in_bucket function we saw earlier.
So, this line essentially tries to call the list_files_in_bucket function to get a list of files from the GCS bucket and stores that list in the files_in_bucket variable for further processing within the DAG. The try block ensures that if any errors occur during this function call (e.g., issues connecting to GCS), the code execution doesn't halt abruptly, and the error can be handled gracefully.



```python
if files_in_bucket:

    # Dictionary to store tasks related to each file
    file_tasks = {}

    for file_name in files_in_bucket:
        if file_name.startswith('log'):
            table_name = 'log_table'
        elif file_name.startswith('ref'):
            table_name = 'reference_table'
        else:
            continue

        task_id = f'load_{file_name.replace(".csv", "")}_to_{table_name}'
        table_id = f'dawn-light-0502.gcs_to_bigquery.{table_name}'
```

This code block checks if there are any files in the GCS bucket and creates a dictionary to store tasks for each file if there are. Here's a breakdown:

1. **file_tasks** = {}: This line creates an empty dictionary named file_tasks. Dictionaries are collections of key-value pairs, where keys are unique identifiers and values can be any data type. Here, the dictionary will be used to store tasks associated with each file.
2. **if file_name.startswith('log'):**
   If the filename starts with "log":

  - table_name = 'log_table': A variable named table_name is assigned the value "log_table". This likely represents the BigQuery table where the data from this file will be loaded.

If the filename starts with "ref":

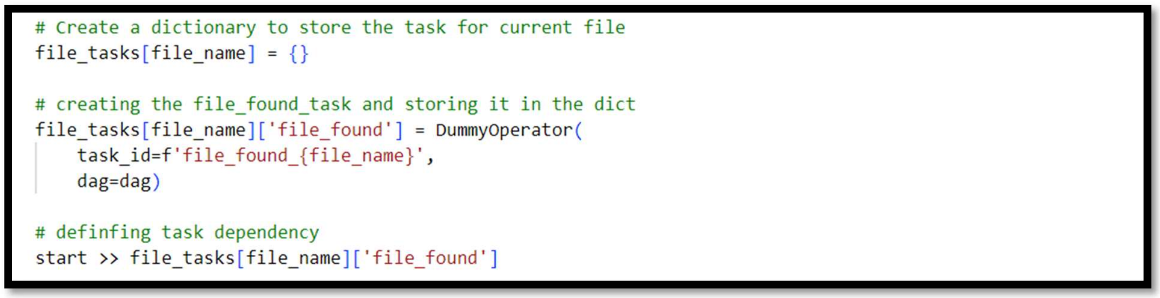
  - table_name = 'reference_table': Similar to the previous case, a variable named table_name is assigned the value "reference_table". This likely represents the BigQuery table for data from files with the "ref" prefix.

3. If the filename doesn't start with either "log" or "ref".The continue statement skips the remaining code within the current loop iteration and moves on to the next filename in the files_in_bucket list. Files that don't match the expected prefixes ("log" or "ref") are likely ignored in this case.

This code snippet essentially categorizes the files based on their prefixes ("log" or "ref") and assigns potential BigQuery table names based on those prefixes. It also creates an empty dictionary (file_tasks) to store information about tasks related to each file for further processing within the DAG.

Then constructs a string variable named task_id using f-strings (formatted string literals).Here, the literal string "load_" is placed first, followed by an expression within curly braces {}.This expression replaces the extension ".csv" in the file_name with an empty string, effectively removing the extension from the filename.This literal string is appended after the replaced filename.Overall, this line creates a unique task ID based on the filename. For example, if a filename is "log_data.csv", the task ID might become "load_log_data_to_log_table". This helps identify tasks associated with specific files.In summary, these lines dynamically generate unique task IDs and table IDs for each file processed within the DAG. These IDs help distinguish tasks and tables associated with different files during data transfer and loading operations.

Creating a dictionary to store the task for current file and defining the task dependency:



```
# Create a dictionary to store the task for current file
file_tasks[file_name] = {}

# creating the file_found_task and storing it in the dict
file_tasks[file_name]['file_found'] = DummyOperator(
    task_id=f'file_found_{file_name}',
    dag=dag)

# definfing task dependency
start >> file_tasks[file_name]['file_found']
```

This code block creates a dictionary entry for each file (file_name) and defines a dummy operator named "file_found" to indicate a file was found.

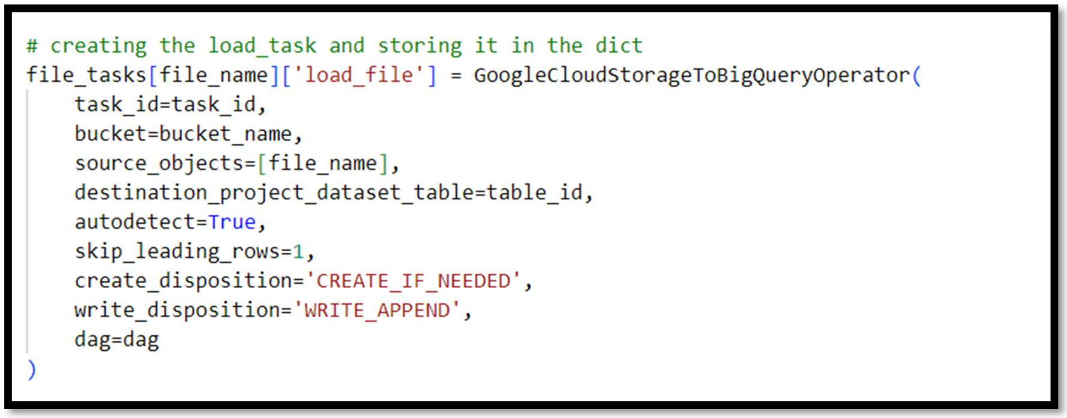
This creates a new dictionary entry within the file_tasks dictionary using the current file_name as the key. The value assigned is an empty dictionary, essentially creating a sub-dictionary within file_tasks to store tasks specific to this particular file.This creates a dummy operator named "file_found" and stores it within the sub-dictionary for the current file (file_tasks[file_name]).

- DummyOperator(...): This part uses the DummyOperator class to create a dummy operator.
- task_id=f'file_found_{file_name}': The task ID is set dynamically using f-strings. It combines the literal string "file_found_" with the current file_name, resulting in unique IDs like "file_found_log_data.csv" (assuming the filename is "log_data.csv").
- dag=dag: This argument specifies the DAG object (dag) to which this operator belongs, associating it with the overall workflow.

2. **start >> file_tasks[file_name]['file_found']**: This line defines a task dependency within the DAG. It uses the greater than (>>) operator, which is a way to define how tasks relate to each other in Airflow. Here, it says that the start dummy operator (created earlier) needs to be completed successfully before the "file_found" task for the current file can run. This ensures the DAG checks for files only after the starting point (start) is reached.

Overall, this code snippet creates a structured way to store information about tasks related to each file found in the GCS bucket. It uses a nested dictionary (file_tasks) to categorize tasks by filename and defines a dummy operator "file_found" to indicate that a specific file was encountered within the bucket. Additionally, it establishes a dependency within the DAG workflow, ensuring the DAG checks for files only after the starting point is reached.

Creating the load_task and storing it in the dict:

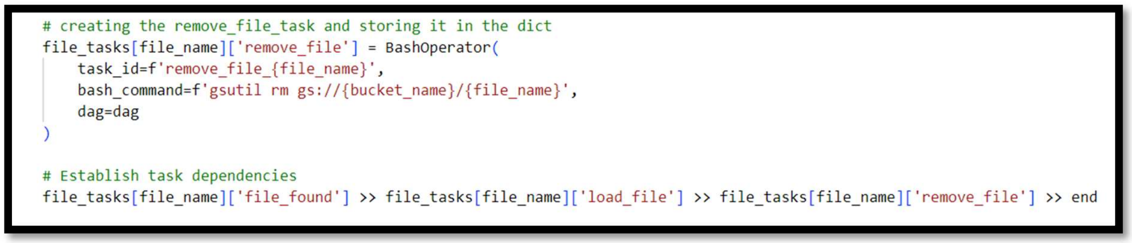


```
# creating the load_task and storing it in the dict
file_tasks[file_name]['load_file'] = GoogleCloudStorageToBigQueryOperator(
    task_id=task_id,
    bucket=bucket_name,
    source_objects=[file_name],
    destination_project_dataset_table=table_id,
    autodetect=True,
    skip_leading_rows=1,
    create_disposition='CREATE_IF_NEEDED',
    write_disposition='WRITE_APPEND',
    dag=dag
)
```

This code snippet is like a recipe for a data pipeline in Airflow. Here's a simpler explanation:

1. **Ingredients:**
    - file_name: Name of the file you want to load from Google Cloud Storage.
    - bucket_name: Name of the bucket where the file resides.
    - table_id: BigQuery table where you want to store the data.
2. **Steps:**
    - Airflow creates a task named task_id to handle this specific file.
    - The task reads the file from your specified bucket.
    - Airflow automatically figures out the file format (CSV, Avro, etc.).
    - It skips the first row of the file (assuming it's a header row).
    - If the BigQuery table doesn't exist, it creates it for you.
    - New data from the file gets appended to the existing data in the table, without deleting anything.

Basically, this code automates loading your data from Google Cloud Storage to a Big Query table, handling some common tasks for you.

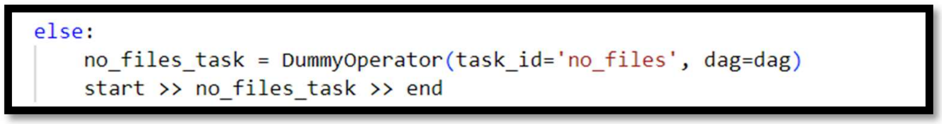

```
# creating the remove_file_task and storing it in the dict
file_tasks[file_name]['remove_file'] = BashOperator(
    task_id=f'remove_file_{file_name}',
    bash_command=f'gsutil rm gs://{bucket_name}/{file_name}',
    dag=dag
)

# Establish task dependencies
file_tasks[file_name]['file_found'] >> file_tasks[file_name]['load_file'] >> file_tasks[file_name]['remove_file'] >> end
```

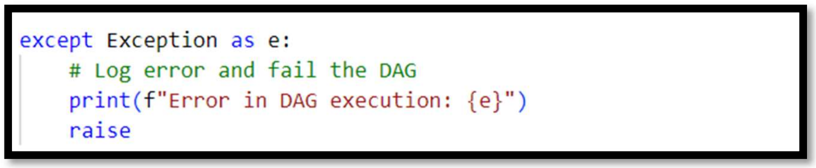
This code defines a task to remove the processed GCS file after loading its data to BigQuery. It uses the BashOperator to execute a gsutil rm command that removes the file from GCS. Additionally, it establishes dependencies between tasks using the greater than (>>) operator. These dependencies ensure the DAG follows a specific order:

1. Check for the file (file_found task).
2. Load data to BigQuery (load_file task).
3. Remove the GCS file (remove_file task).
4. Mark DAG completion (end task).

By chaining tasks with dependencies, the code ensures data is loaded successfully before deletion and establishes a clear execution flow within the DAG.


```
else:
    no_files_task = DummyOperator(task_id='no_files', dag=dag)
    start >> no_files_task >> end
```

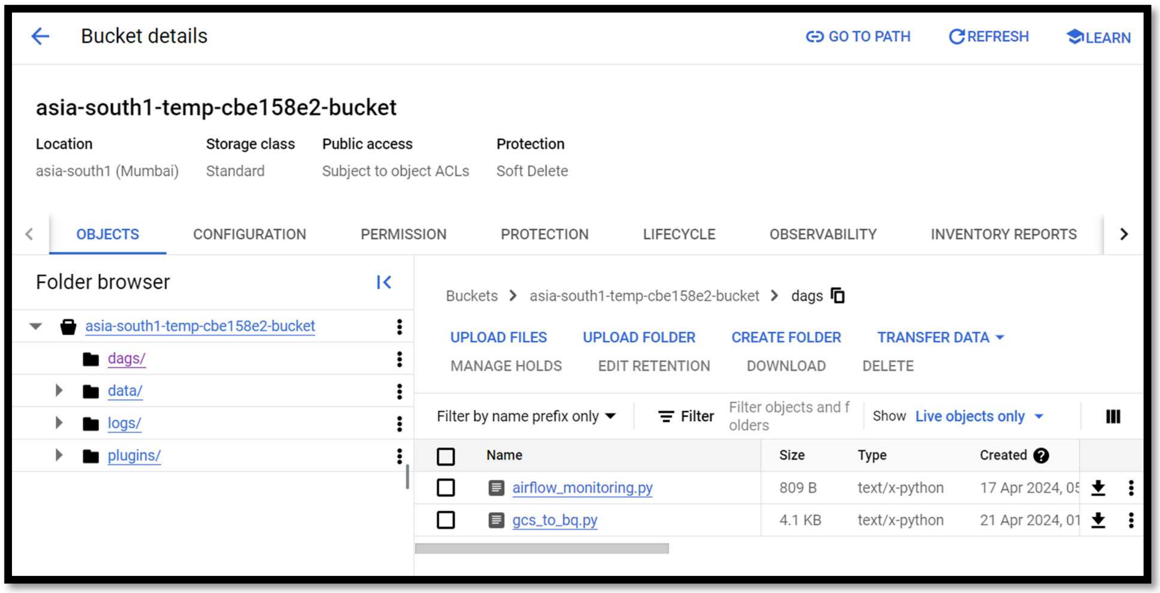
This Airflow DAG checks for files in a GCS bucket, loads their data to BigQuery (if found), and optionally removes them. It uses operators and dependencies to manage workflow and handle cases with or without files.


```
except Exception as e:
    # Log error and fail the DAG
    print(f"Error in DAG execution: {e}")
    raise
```

If an exception occurs within the try block (e.g., issues with GCS or BigQuery), it's logged with details and then re-raised. This ensures errors are caught, reported (logged), and the DAG execution fails for further attention. Re-raising allows Airflow's error handling mechanisms to take effect.

STEP 6

Deploy Dag into the dag folder which is present in the composer. The Dag is uploaded as the python file named as ‘gcs_to_bq.py’.



asia-south1-temp-cbe158e2-bucket

Location	Storage class	Public access	Protection
asia-south1 (Mumbai)	Standard	Subject to object ACLs	Soft Delete

asia-south1-temp-cbe158e2-bucket
- dags/
- data/
- logs/
- plugins/

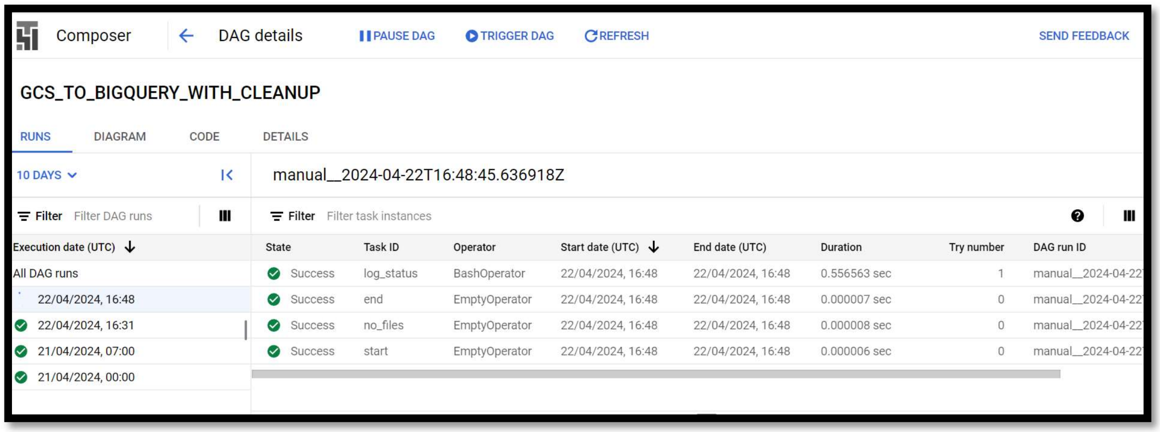
Buckets > asia-south1-temp-cbe158e2-bucket > dags

Name	Size	Type	Created
airflow_monitoring.py	809 B	text/x-python	17 Apr 2024, 05
gcs_to_bq.py	4.1 KB	text/x-python	21 Apr 2024, 01

STEP 7

TRIGGER DAG

We triggered the Dag and there is no files uploaded. So, that task get started and it is checked for the files and the task is ended.



GCS_TO_BIGQUERY_WITH_CLEANUP

manual__2024-04-22T16:48:45.636918Z

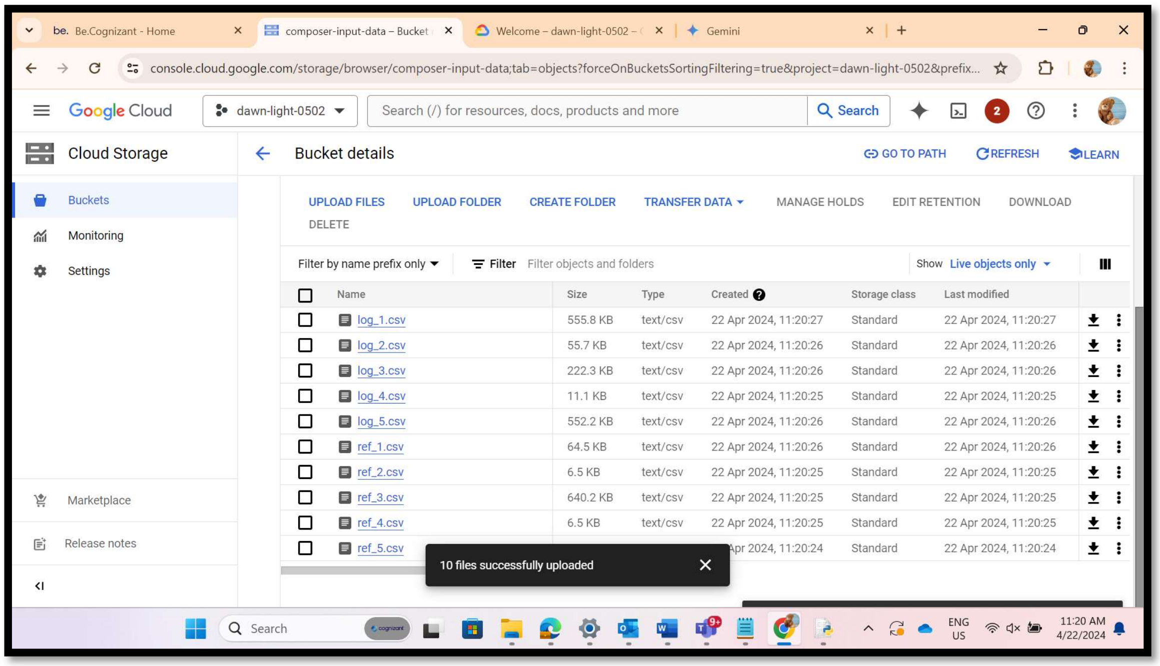
Execution date (UTC)
All DAG runs
22/04/2024, 16:48
22/04/2024, 16:31
21/04/2024, 07:00
21/04/2024, 00:00

State	Task ID	Operator	Start date (UTC)	End date (UTC)	Duration	Try number	DAG run ID
Success	log_status	BashOperator	22/04/2024, 16:48	22/04/2024, 16:48	0.556563 sec	1	manual__2024-04-22
Success	end	EmptyOperator	22/04/2024, 16:48	22/04/2024, 16:48	0.000007 sec	0	manual__2024-04-22
Success	no_files	EmptyOperator	22/04/2024, 16:48	22/04/2024, 16:48	0.000008 sec	0	manual__2024-04-22
Success	start	EmptyOperator	22/04/2024, 16:48	22/04/2024, 16:48	0.000006 sec	0	manual__2024-04-22

STEP 8

UPLOAD FILES

In our case, we have taken 10 files as the input and uploaded the files in the input bucket.

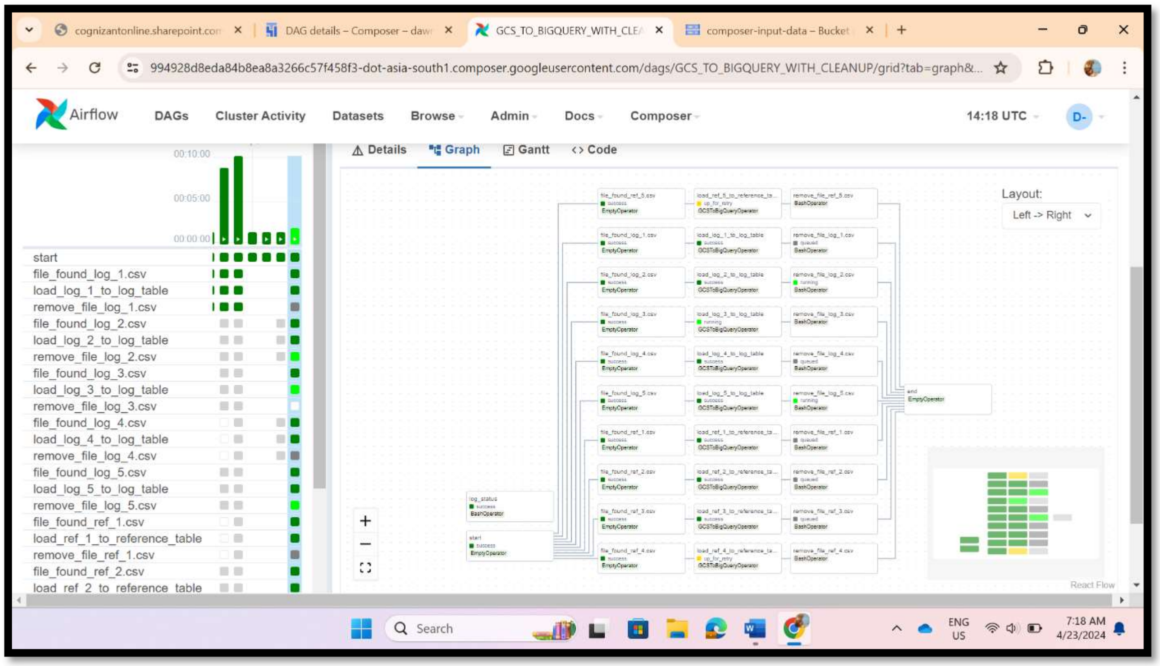

be. Be.Cognizant - Home
composer-input-data – Bucket
Welcome – dawn-light-0502 –
Gemini
console.cloud.google.com/storage/browser/composer-input-data;tab=objects?forceOnBucketsSortingFiltering=true&project=dawn-light-0502&prefix...
Google Cloud
dawn-light-0502
Search (/) for resources, docs, products and more
Search
Cloud Storage
Bucket details
GO TO PATH
REFRESH
LEARN
Buckets
Monitoring
Settings
Marketplace
Release notes
UPLOAD FILES
UPLOAD FOLDER
CREATE FOLDER
TRANSFER DATA
MANAGE HOLDS
EDIT RETENTION
DOWNLOAD
DELETE
Filter by name prefix only
Filter
Filter objects and folders
Show Live objects only

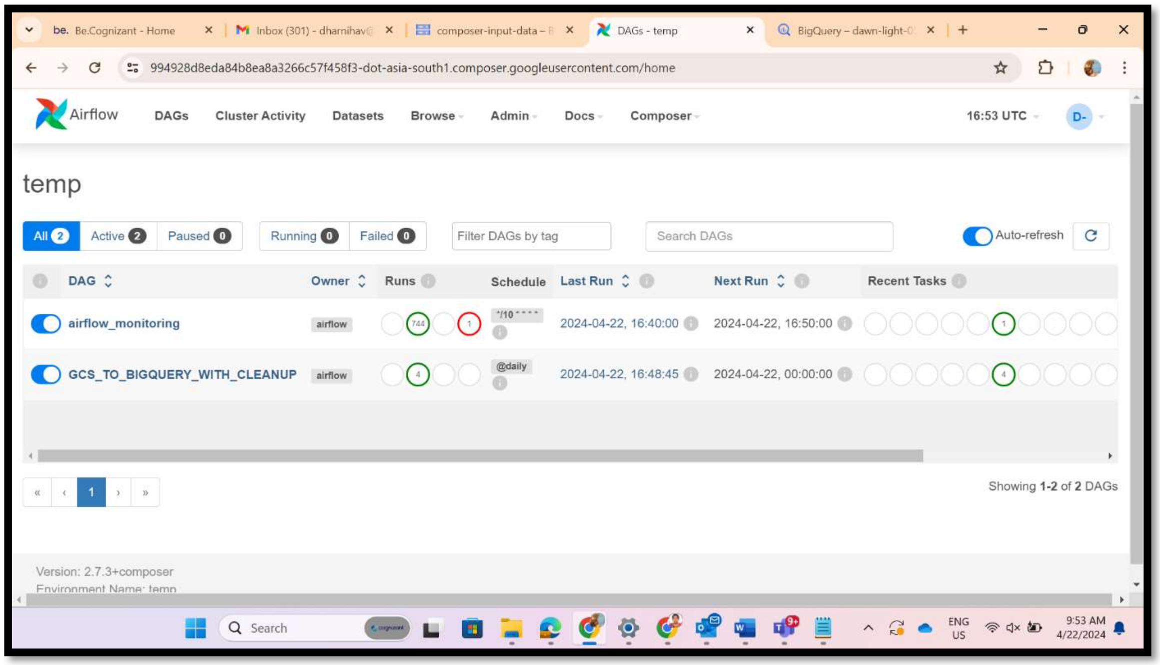
Name	Size	Type	Created	Storage class	Last modified
log_1.csv	555.8 KB	text/csv	22 Apr 2024, 11:20:27	Standard	22 Apr 2024, 11:20:27
log_2.csv	55.7 KB	text/csv	22 Apr 2024, 11:20:26	Standard	22 Apr 2024, 11:20:26
log_3.csv	222.3 KB	text/csv	22 Apr 2024, 11:20:26	Standard	22 Apr 2024, 11:20:26
log_4.csv	11.1 KB	text/csv	22 Apr 2024, 11:20:25	Standard	22 Apr 2024, 11:20:25
log_5.csv	552.2 KB	text/csv	22 Apr 2024, 11:20:26	Standard	22 Apr 2024, 11:20:26
ref_1.csv	64.5 KB	text/csv	22 Apr 2024, 11:20:26	Standard	22 Apr 2024, 11:20:26
ref_2.csv	6.5 KB	text/csv	22 Apr 2024, 11:20:25	Standard	22 Apr 2024, 11:20:25
ref_3.csv	640.2 KB	text/csv	22 Apr 2024, 11:20:25	Standard	22 Apr 2024, 11:20:25
ref_4.csv	6.5 KB	text/csv	22 Apr 2024, 11:20:25	Standard	22 Apr 2024, 11:20:25
ref_5.csv			Apr 2024, 11:20:24	Standard	22 Apr 2024, 11:20:24

10 files successfully uploaded

AIRFLOW WEB SERVER FOR REFERENCE

While tiggering the Dag, we can see the graph in the airflow web UI and as we seen before there will be the status of the task which is defined by the colour codes.




994928d8eda84b8ea8a3266c57f458f3-dot-asia-south1.composer.googleusercontent.com/home
Airflow
DAGs
Cluster Activity
Datasets
Browse
Admin
Docs
Composer
16:53 UTC

temp

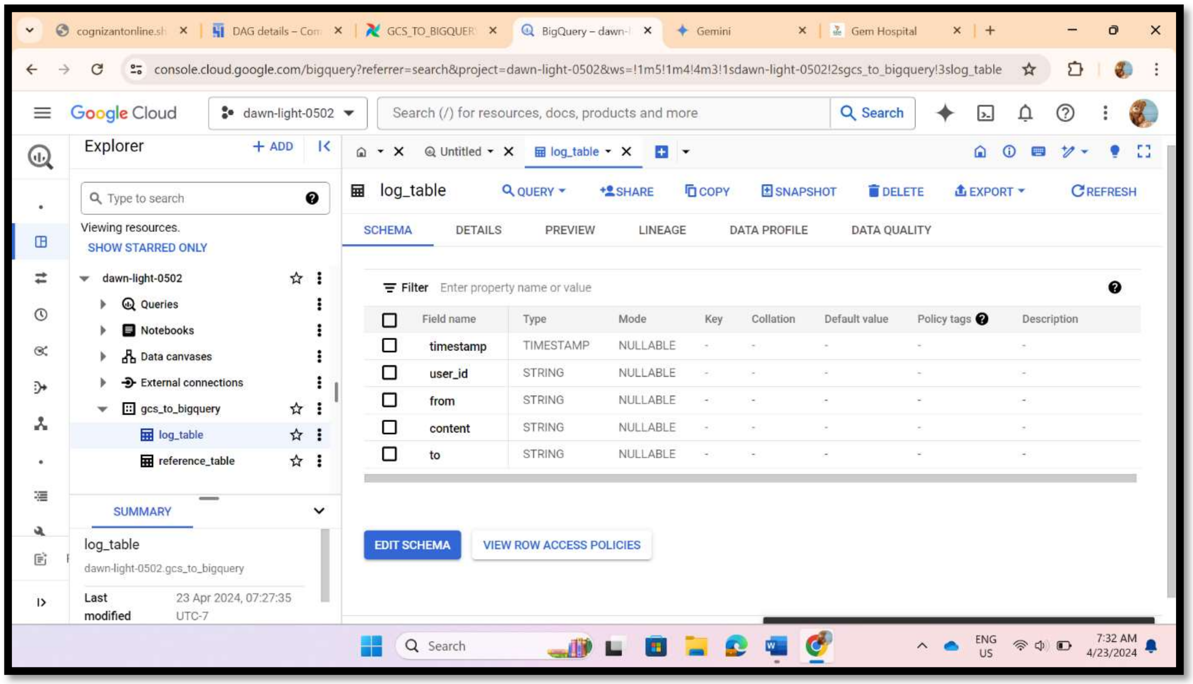
All 2 | Active 2 | Paused 0 | Running 0 | Failed 0 | Filter DAGs by tag | Search DAGs | Auto-refresh

DAG	Owner	Runs	Schedule	Last Run	Next Run	Recent Tasks
airflow_monitoring	airflow	744 1	*/10 * * * *	2024-04-22, 16:40:00	2024-04-22, 16:50:00	1
GCS_TO_BIGQUERY_WITH_CLEANUP	airflow	4	@daily	2024-04-22, 16:48:45	2024-04-22, 00:00:00	4

Showing 1-2 of 2 DAGs

Version: 2.7.3+composer
Environment Name: temp

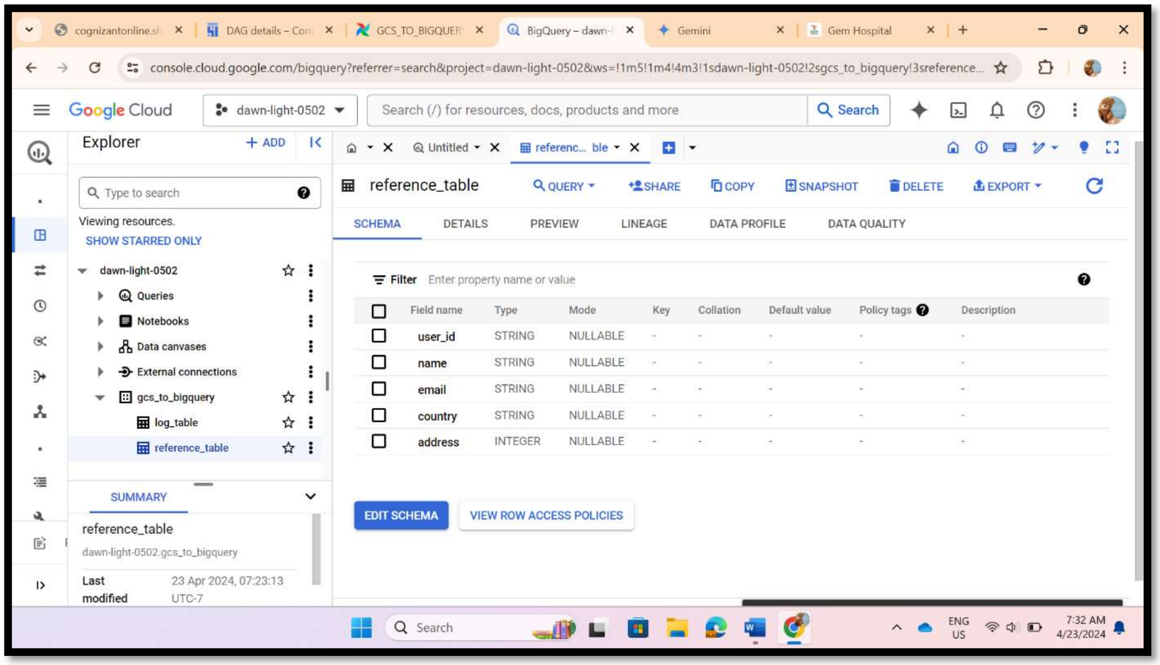
Schema of the table that is created in the big query:



log_table

Field name	Type	Mode	Key	Collation	Default value	Policy tags	Description
timestamp	TIMESTAMP	NULLABLE	-	-	-	-	-
user_id	STRING	NULLABLE	-	-	-	-	-
from	STRING	NULLABLE	-	-	-	-	-
content	STRING	NULLABLE	-	-	-	-	-
to	STRING	NULLABLE	-	-	-	-	-

log_table
dawn-light-0502.gcs_to_bigquery
Last modified 23 Apr 2024, 07:27:35 UTC-7



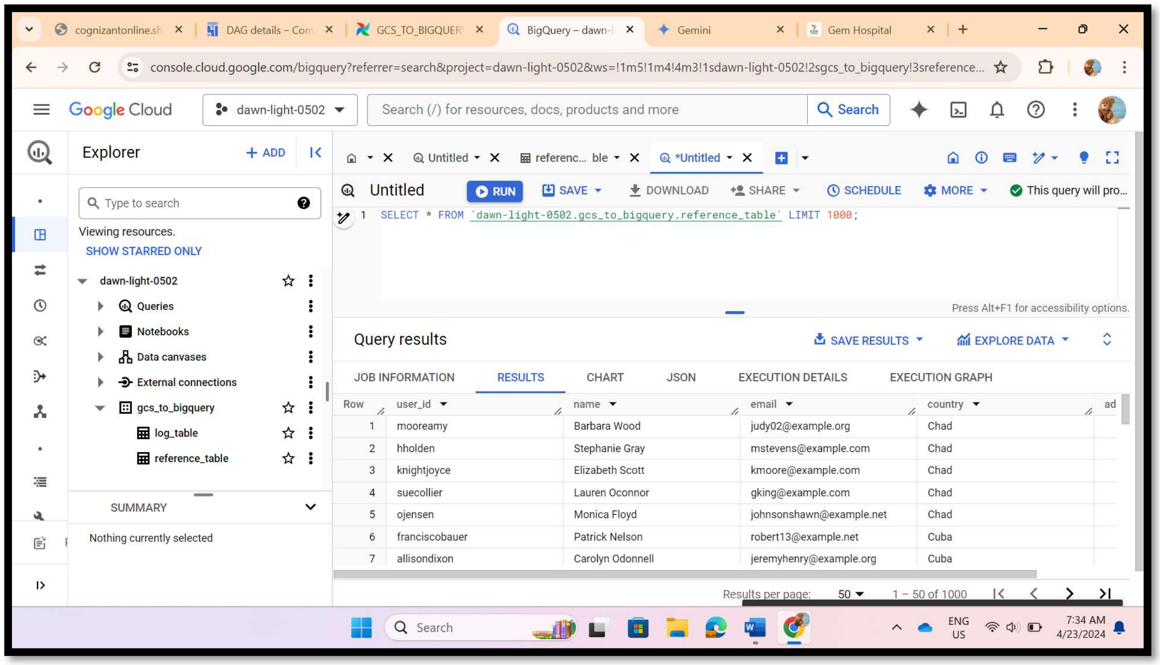
reference_table

Field name	Type	Mode	Key	Collation	Default value	Policy tags	Description
user_id	STRING	NULLABLE	-	-	-	-	-
name	STRING	NULLABLE	-	-	-	-	-
email	STRING	NULLABLE	-	-	-	-	-
country	STRING	NULLABLE	-	-	-	-	-
address	INTEGER	NULLABLE	-	-	-	-	-

reference_table
dawn-light-0502.gcs_to_bigquery
Last modified 23 Apr 2024, 07:23:13 UTC-7

We can see the schema of the table that is divided as the two able namely log_table, reference_table as we mentioned in the Dag.

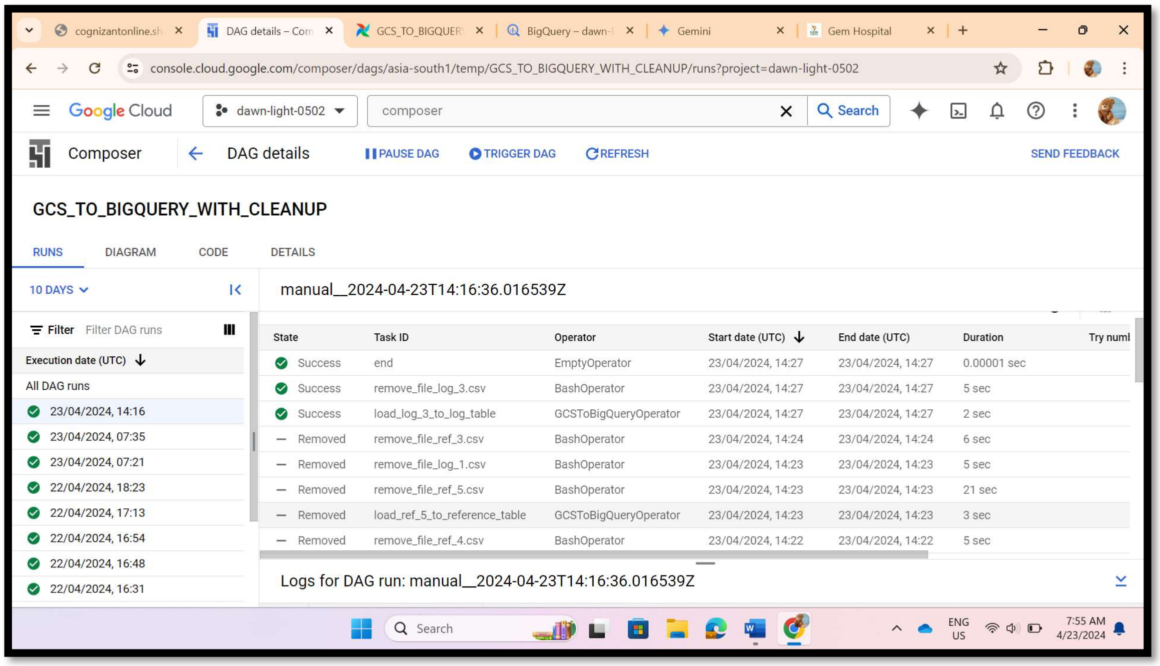
Querying the data in big query:



```
SELECT * FROM `dawn-light-0502.gcs_to_bigquery.reference_table` LIMIT 1000;
```

Row	user_id	name	email	country
1	mooreamy	Barbara Wood	judy02@example.org	Chad
2	hholden	Stephanie Gray	mstevens@example.com	Chad
3	knightjoyce	Elizabeth Scott	kmoore@example.com	Chad
4	suecollier	Lauren Oconnor	gking@example.com	Chad
5	ojensen	Monica Floyd	johnsonshawn@example.net	Chad
6	franciscobauer	Patrick Nelson	robert13@example.net	Cuba
7	allisondixon	Carolyn Odonnell	jeremyhenry@example.org	Cuba

We can see the Trigger is completed and all tasks are done.


console.cloud.google.com/composer/dags/asia-south1/temp/GCS_TO_BIGQUERY_WITH_CLEANUP/runs?project=dawn-light-0502
Google Cloud
dawn-light-0502
Composer
DAG details
PAUSE DAG
TRIGGER DAG
REFRESH
SEND FEEDBACK
GCS_TO_BIGQUERY_WITH_CLEANUP
RUNS
DIAGRAM
CODE
DETAILS
10 DAYS
Logs for DAG run: manual__2024-04-23T14:16:36.016539Z
7:55 AM
4/23/2024

Execution date (UTC)
All DAG runs
✓ 23/04/2024, 14:16
✓ 23/04/2024, 07:35
✓ 23/04/2024, 07:21
✓ 22/04/2024, 18:23
✓ 22/04/2024, 17:13
✓ 22/04/2024, 16:54
✓ 22/04/2024, 16:48
✓ 22/04/2024, 16:31

manual__2024-04-23T14:16:36.016539Z

State	Task ID	Operator	Start date (UTC)	End date (UTC)	Duration	Try numl
Success	end	EmptyOperator	23/04/2024, 14:27	23/04/2024, 14:27	0.00001 sec	
Success	remove_file_log_3.csv	BashOperator	23/04/2024, 14:27	23/04/2024, 14:27	5 sec	
Success	load_log_3_to_log_table	GCSToBigQueryOperator	23/04/2024, 14:27	23/04/2024, 14:27	2 sec	
Removed	remove_file_ref_3.csv	BashOperator	23/04/2024, 14:24	23/04/2024, 14:24	6 sec	
Removed	remove_file_log_1.csv	BashOperator	23/04/2024, 14:23	23/04/2024, 14:23	5 sec	
Removed	remove_file_ref_5.csv	BashOperator	23/04/2024, 14:23	23/04/2024, 14:23	21 sec	
Removed	load_ref_5_to_reference_table	GCSToBigQueryOperator	23/04/2024, 14:23	23/04/2024, 14:23	3 sec	
Removed	remove_file_ref_4.csv	BashOperator	23/04/2024, 14:22	23/04/2024, 14:22	5 sec	

CONCLUSION

The utilization of Google Cloud Platform services, specifically Cloud Composer for workflow orchestration, Google Cloud Storage for file storage, and BigQuery for data analysis, provides a robust and scalable solution for our project requirements. The seamless integration between Composer, GCS, and BigQuery allows for efficient scheduling, management, and analysis of data pipelines. Moving forward, considerations for enhancing the project may include implementing monitoring, logging, and error handling mechanisms to ensure the reliability and performance of our workflows. Overall, Composer empowers us to focus on building and managing workflows effectively, leveraging the benefits of cloud computing and managed services offered by Google Cloud Platform.