Data Pipeline Design Document

Table of Contents

Orchestrating ETL data pipelines on Google Cloud Platform using Airflow and Python

1. Introduction

In this document, we'll dive into the design and implementation of our data pipelines for an Airline. These pipelines handle the critical task of processing and transforming various data sources—like passenger information (pax), in-flight sales data, and goods loading information—into a structured and accessible format for our data scientists and analysts.
We'll explore each component, the workflow, and the technology stack we use to ensure efficient and reliable data processing.

Purpose:
The purpose of this document is to provide a comprehensive overview of our data pipeline architecture. It's meant to guide you through the steps we've taken to build and maintain these pipelines, ensuring you understand the underlying design principles and can effectively troubleshoot or expand the system as needed.

Whether you're new to data processing workflows on Google Cloud Platform, or looking to set-up batch pipelines using Airflow, this document will serve as your go-to reference.

Scope:
This document covers the end-to-end design and implementation of our data pipelines, specifically focusing on the ingestion, transformation, and loading of pax, sales, and loading data.

We'll discuss the use of Google Cloud Storage (GCS) | Google Cloud Composer (Airflow) | BigQuery | Cloud Functions | and Snowflake.

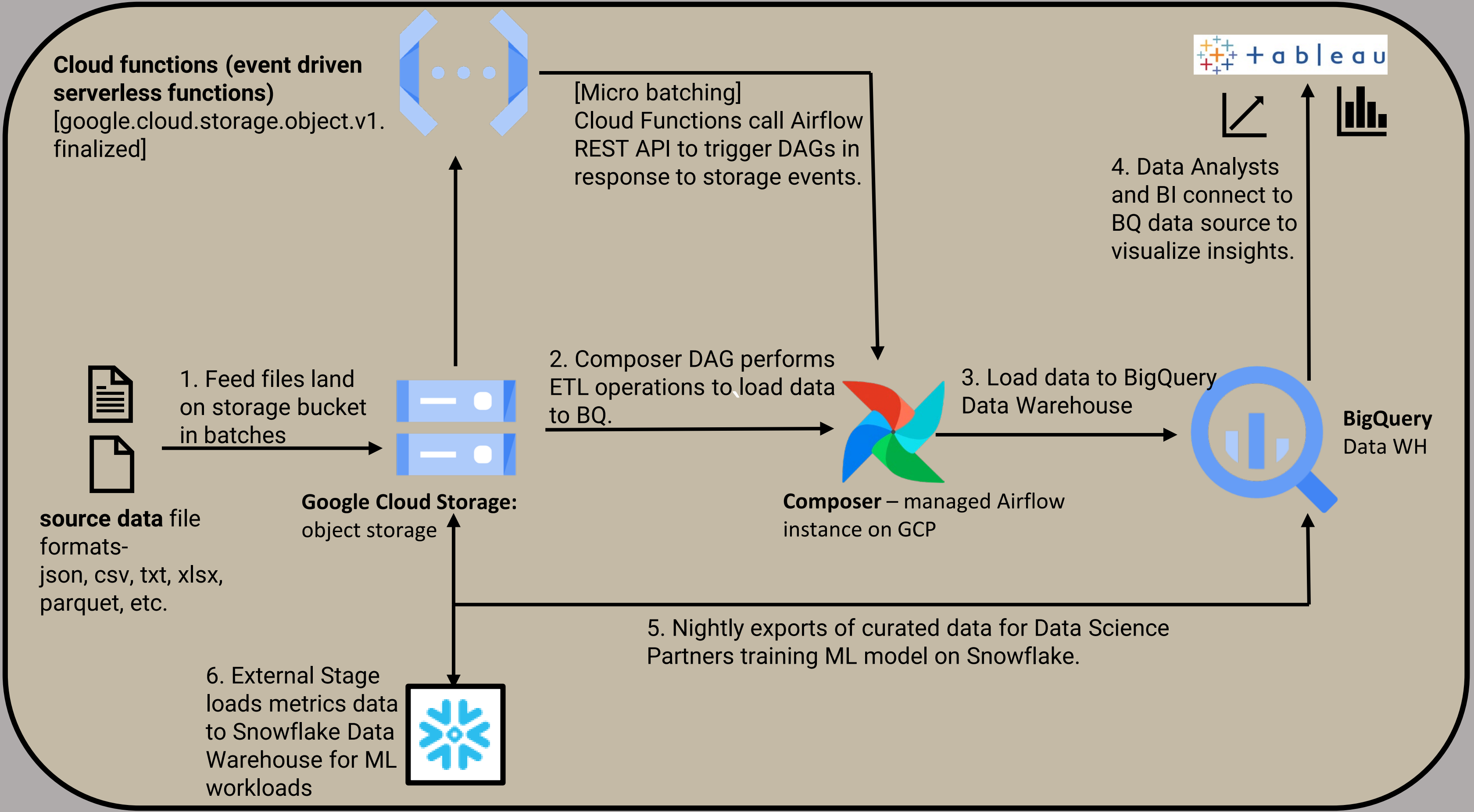
- The scope includes the daily and event-driven workflows.
- The data transformation processes using SQL queries.
- And the export mechanisms that integrate with Snowflake.

By the end of this document, you should have a clear understanding of how these components work together to deliver clean, reliable data to our stakeholders.

2. System Architecture

Process Flow:

- Every day, raw data files are uploaded to our Google Cloud Storage (GCS) buckets for sales, pax, and loading data.
- The nightly scheduled DAGs orchestrate the entire ETL process, extracting data from GCS, transforming it using SQL queries in BigQuery, and loading it into BigQuery tables.
- The transformation steps include cleaning the data and performing necessary calculations to prepare it for analysis.
- Once the data is processed, it's exported back to GCS if needed, and separate Snowflake pipeline loads this curated data into warehouse.
- Throughout this process, Cloud Logging keeps track of all activities, making it easy to monitor and debug.



High Level Architecture Diagram

Components and Technologies Used:

- Google Cloud Storage (GCS):
  Stores raw data files (CSV, Excel, JSON) for sales, pax, and loading data.

- Google Cloud Composer (Airflow):
  Orchestrates ETL processes, managing data workflows and scheduling DAGs.

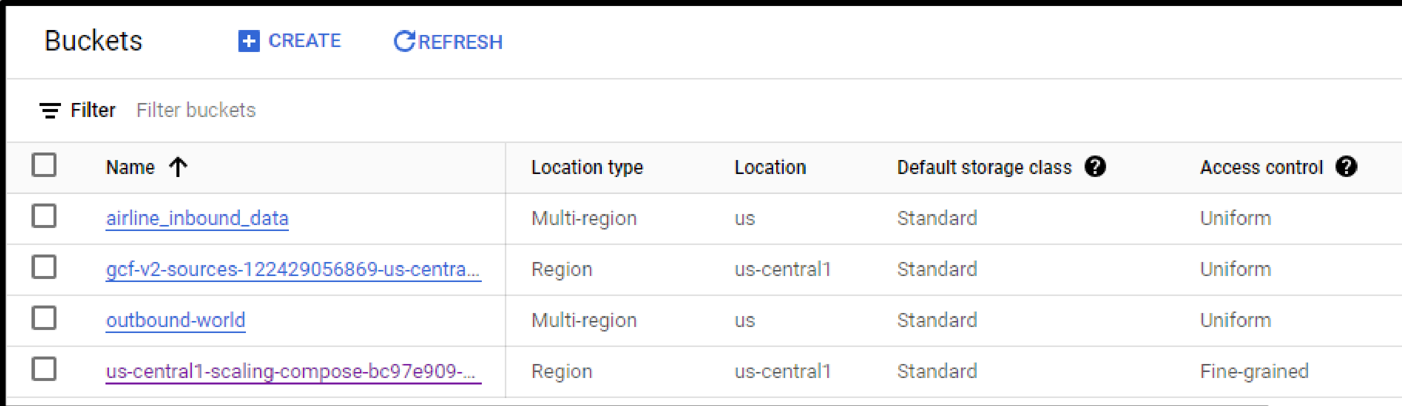
- BigQuery:
  Processes and transforms data using SQL, enabling efficient querying and analysis.

- Cloud Functions:
  Handles ad hoc micro batching by triggering DAGs upon file arrival in GCS.

- Cloud Logging:
  Monitors and logs all activities, aiding in debugging and performance tracking.

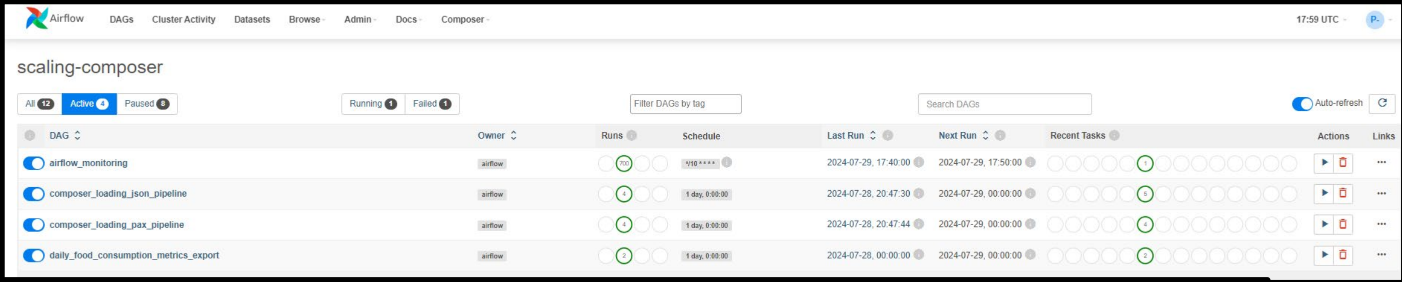
- Python:
  Used in Airflow DAGs for scripting and data manipulation tasks.

- SQL:
  Employed in BigQuery for data transformation, cleaning, and querying.

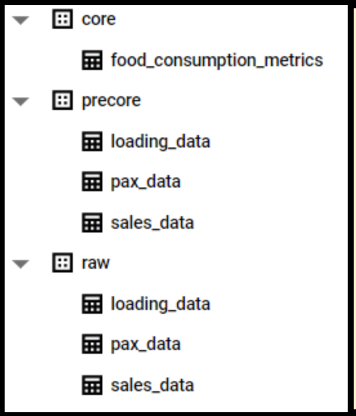


Name	Location type	Location	Default storage class	Access control
airline_inbound_data	Multi-region	us	Standard	Uniform
gcf-v2-sources-122429056869-us-centra...	Region	us-central1	Standard	Uniform
outbound-world	Multi-region	us	Standard	Uniform
us-central1-scaling-compose-bc97e909-...	Region	us-central1	Standard	Fine-grained

Cloud Storage buckets for : Daily source feeds | Artifact Registry | Data exports to external systems | composer bucket for dags, plugins, and logs.



Daily batch pipelines on **Airflow** for pax | sales | loading ETL workloads.

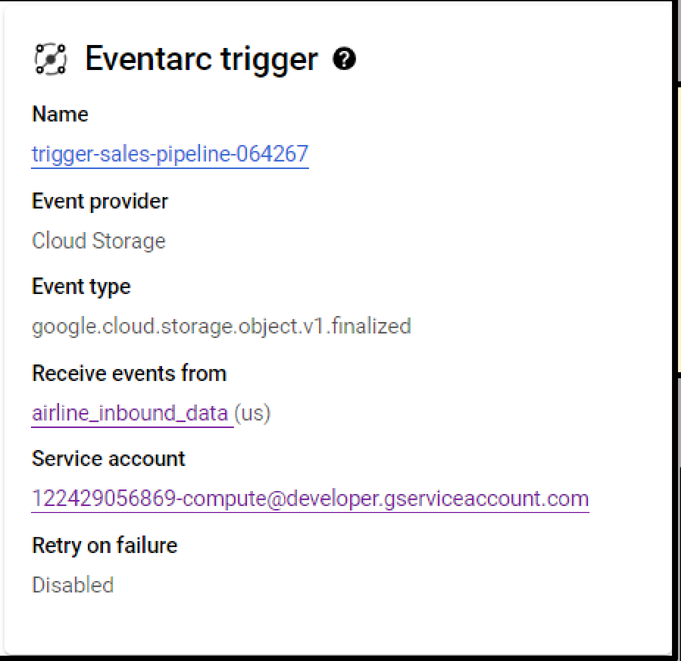


BigQuery datasets

raw: truncate and load tables for pax | sales | loading

precore:
preprocessed and cleaned data

core:
curated layer to calculate metrics

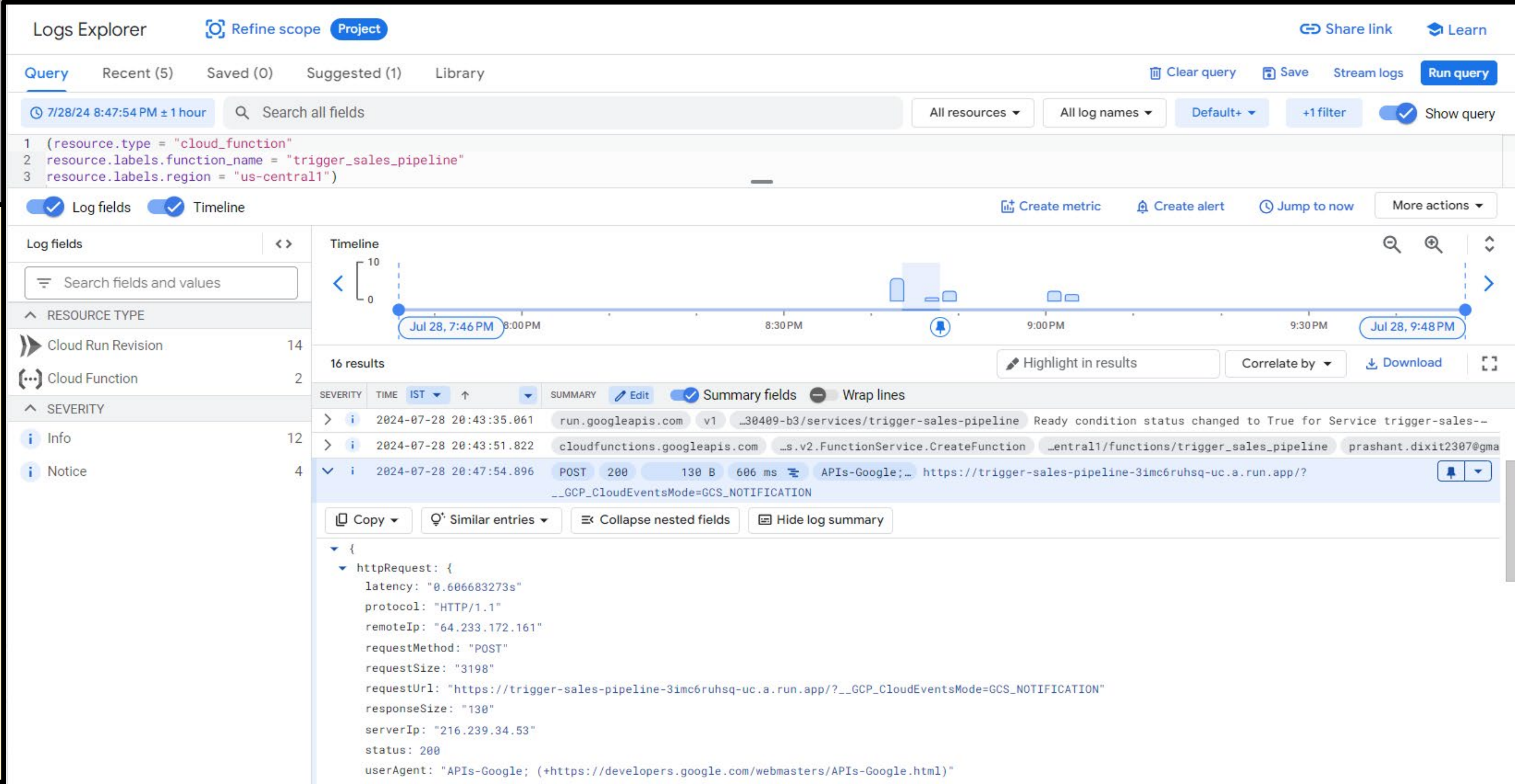


The **EventArc** trigger named ‘trigger-sales-pipeline-064267’ is set up to activate the ‘trigger-sales-pipeline’ **Cloud Function** whenever a new file is finalized/created in the airline_inbound_data bucket. It uses the default compute service account to handle these events from Google Cloud Storage.

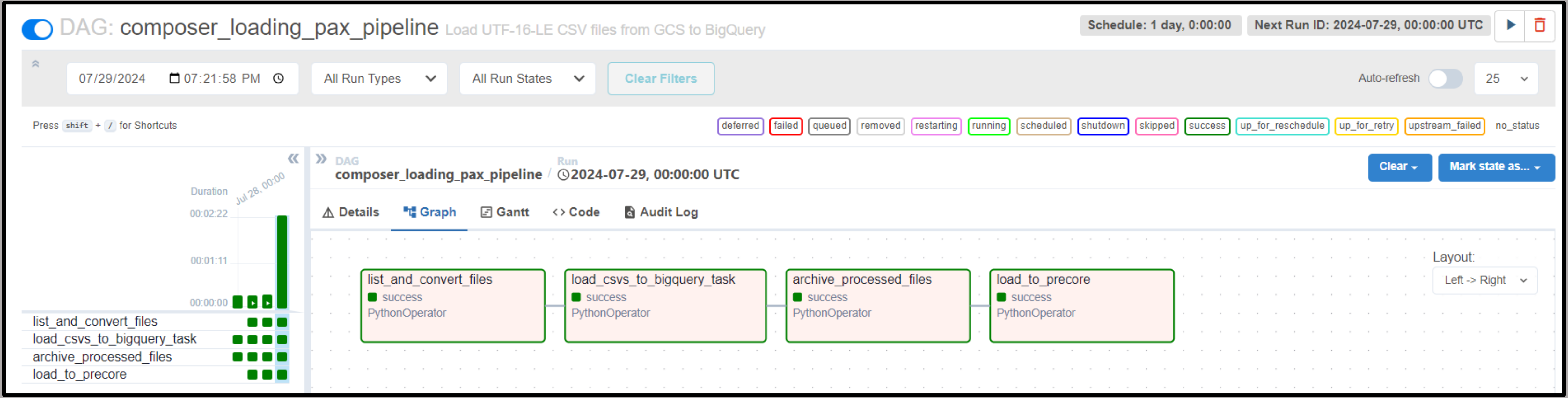
Eventarc lets you asynchronously deliver events between decoupled microservices while managing delivery, security, authorization, observability, and error-handling for you.

Cloud Logging plays a crucial role in monitoring and troubleshooting by capturing detailed logs of Cloud Function executions. For example, logs show requests to the ‘triggersalespipeline’ function, including HTTP request details, response sizes, and execution latencies, which helps ensure the function is operating correctly and identifies potential issues.

- Logs provide information on HTTP request methods, status codes, and response sizes for monitoring function performance.
- Resource labels identify the specific Cloud Run service and revision associated with the function, aiding in pinpointing issues.



3. Pipeline Details [pax, sales, loading]



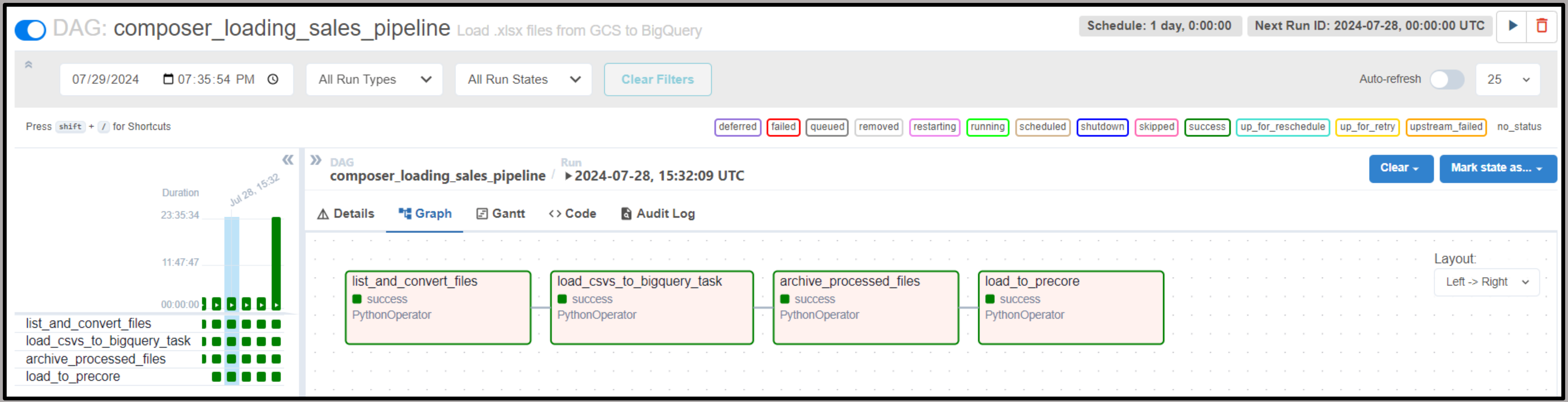
Source Data:
CSV files stored in GCS bucket `gs://airline_inbound_data/pax/` with utf-16-le encoding.
File pattern: pax_%m_%Y.csv [for ex: pax_03_2019.csv and pax_04_2019.csv]

Destination:
BigQuery table: `sunlit-analyst-430409-b3.raw.pax_data`.

Steps and Transformations:
Loading: CSV files are read directly from GCS using Google Cloud Storage Client in Composer DAGs.

Transformations: Data is cleaned and transformed using SQL queries in BigQuery. The transformation process includes converting the UTF16LE encoded text to a readable format and handling data type conversions.

Load Mode: Data is loaded into raw BigQuery tables in write_truncate mode, then processed to move transformed data into precore tables, including adding a timestamp field.



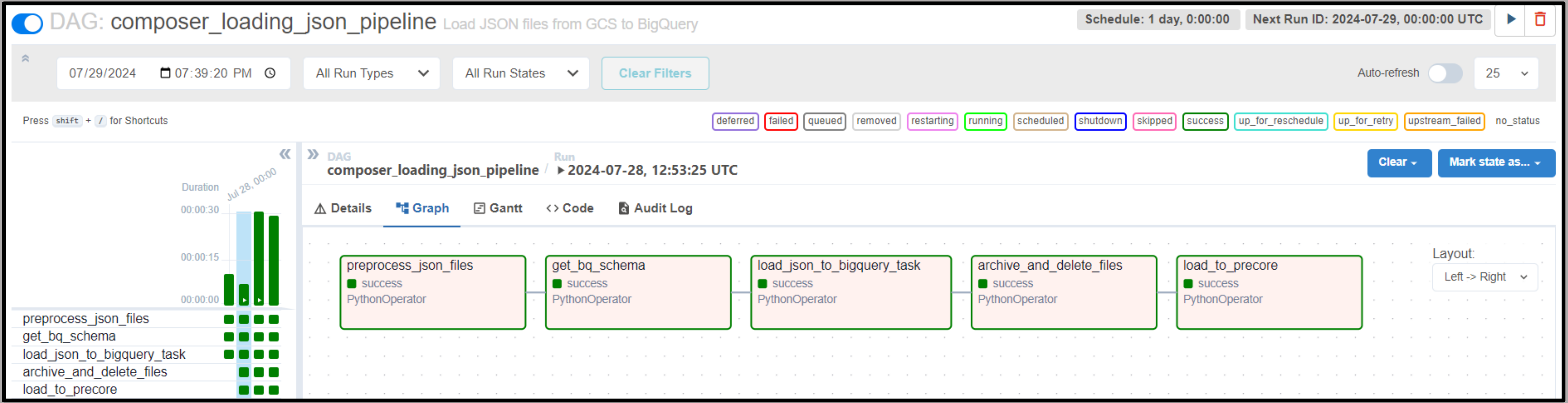
Source Data:
Excel files stored in GCS bucket `gs://airline_inbound_data/sales/`,
File pattern: Sales_Report_%B_%Y.xlsx [Sales_Report_April_2019.xlsx, Sales_Report_March_2019.xlsx]

Destination:
BigQuery table: `sunlit-analyst-430409-b3.raw.sales_data`.

Steps and Transformations:
Loading: Excel files are loaded from GCS, with processing to handle Excel specific challenges, such as formatting issues and data type recognition.

Transformations: Data is transformed using SQL queries in BigQuery (data cleaning, formatting adjustments, and conversion into a consistent structure)

Load Mode: Data is initially loaded into raw BigQuery tables in write_truncate mode and then moved to precore tables with additional timestamp fields.



Source Data:
JSON files stored in GCS bucket `gs://airline_inbound_data/loading/`.
File pattern: loading_%m_%Y.json   [loading_03_2019.json, loading_04_2019.json].

Destination:
BigQuery table: `sunlit-analyst-430409-b3.raw.loading_data`

Steps and Transformations:
Loading: JSON files are read from GCS with special handling for problematic keys and escape characters, ensuring proper parsing.

Transformations: Data is cleaned and structured using SQL queries in BigQuery. Transformation involves resolving issues with JSON formatting, such as handling spaces in keys.

Load Mode: Data is first loaded into raw BigQuery tables in write truncate mode, then processed to move transformed data into `precore` tables, including adding a timestamp field.

4. Common Pipeline Components

Data Extraction using storage client

In the pipelines, I've used the Google Cloud Storage Client Library to efficiently interact with GCS buckets. By importing the library and instantiating a storage client, the process of accessing and managing files is streamlined. This allows to easily read data from various sources, such as CSVs, Excel files, and JSON files, directly from GCS, facilitating smooth data extraction and integration into BigQuery. The client library simplifies authentication and reduces the amount of code needed, making it easier to handle different file formats and manage data across the pipelines.

```python
def preprocess_json_files(bucket_name, input_folder, output_folder):
    client = storage.Client()
    bucket = client.get_bucket(bucket_name)
    blobs = bucket.list_blobs(prefix=input_folder)

    for blob in blobs:
        if blob.name.endswith('.json'):
            print(f"Processing file: {blob.name}")
            json_file_path = f'/tmp/{os.path.basename(blob.name)}'
            blob.download_to_filename(json_file_path)

            with open(json_file_path, 'r', encoding='utf-8') as f:
                records = json.load(f)

            processed_records = [map_fields(record) for record in records]
            processed_json_file_path = f'/tmp/processed_{os.path.basename(blob.name)}'

            with open(processed_json_file_path, 'w', encoding='utf-8') as f:
                for record in processed_records:
                    f.write(json.dumps(record) + '\n')

            processed_blob = bucket.blob(output_folder + os.path.basename(processed_json_file_path))
            processed_blob.upload_from_filename(processed_json_file_path)
            print(f"Uploaded processed file to: {processed_blob.name}")
```

Snippet from loading pipeline - mapping json records as per table schema

```python
def list_and_convert_csv_files(bucket_name, input_folder, converted_folder):
    print("Starting to list and convert CSV files from the GCS bucket...")
    client = storage.Client()
    bucket = client.get_bucket(bucket_name)
    blobs = client.list_blobs(bucket_name, prefix=input_folder)
    csv_files = [blob.name for blob in blobs if blob.name.endswith('.csv')]

    for csv_file in csv_files:
        print(f"Converting {csv_file} to UTF-8 encoding...")
        blob = bucket.blob(csv_file)
        csv_file_path = f'/tmp/{os.path.basename(csv_file)}'
        blob.download_to_filename(csv_file_path)
        df = pd.read_csv(csv_file_path, encoding='utf-16-le')

        converted_file_path = f'/tmp/{os.path.basename(csv_file)}'
        df.to_csv(converted_file_path, index=False, encoding='utf-8')
        print(f"Converted {csv_file} to UTF-8 CSV at {converted_file_path}")

        converted_file = csv_file.replace(input_folder, converted_folder)
        converted_blob = bucket.blob(converted_file)
        converted_blob.upload_from_filename(converted_file_path)
        print(f"Uploaded converted CSV file to GCS at {converted_file}")

    return csv_files
```

Snippet from pax pipeline – utf-16-le encoded csv processing

Loading Data into BigQuery

Here I've used Airflow's 'GCSToBigQueryOperator' to load the data into BigQuery raw tables.

The schema to be used for the BigQuery table may be specified in one of two ways. We may either directly pass the schema fields in or point the operator to a Google Cloud Storage object name. The object in Google Cloud Storage must be a JSON file with the schema fields in it.

We are loading multiple objects from a single bucket using the 'source_objects' parameter.

```python
def load_to_bq_callable(**kwargs):
    schema = fetch_schema_from_bigquery(BQ_PROJECT_NAME, BQ_DATASET_NAME, BQ_TABLE_NAME)
    load_to_bq_task = GCSToBigQueryOperator(
        task_id='load_csvs_to_bigquery',
        bucket=BUCKET_NAME,
        source_objects=[f'{CONVERTED_FOLDER}*.csv'],
        destination_project_dataset_table=f'{BQ_PROJECT_NAME}.{BQ_DATASET_NAME}.{BQ_TABLE_NAME}',
        schema_fields=schema,
        source_format='CSV',
        skip_leading_rows=1,
        write_disposition='WRITE_TRUNCATE',
        autodetect=False,
        dag=dag)
    load_to_bq_task.execute(context=kwargs)
    print("Loading CSV files to BigQuery...")
```

JSON Load Method:

- Uses NEWLINE_DELIMITED_JSON format for JSON files.
- Schema is retrieved from previous task via XCom and applied during the load.
- WRITE_TRUNCATE option overwrites existing data in the BigQuery raw table.

```python
def load_to_bq_callable(**kwargs):
    schema = kwargs['ti'].xcom_pull(key='bq_schema', task_ids='get_bq_schema')
    load_to_bq_task = GCSToBigQueryOperator(
        task_id='load_json_to_bigquery',
        bucket=BUCKET_NAME,
        source_objects=[f'{PROCESSED_JSON_FOLDER}*.json'],
        destination_project_dataset_table=f'{BQ_PROJECT_NAME}.{BQ_DATASET_NAME}.{BQ_TABLE_NAME}',
        schema_fields=schema,
        source_format='NEWLINE_DELIMITED_JSON',
        write_disposition='WRITE_TRUNCATE',
        autodetect=False,
        dag=dag,
    )
    load_to_bq_task.execute(context=kwargs)
    print("Loading JSON files to BigQuery...")
```

CSV Load Method:

- Uses CSV format for CSV files.
- Skips the header row with skip_leading_rows=1.
- Schema is fetched directly from BigQuery, and WRITE_TRUNCATE option is used for raw loads.

Data cleaning and transformations

In our data pipelines for `pax`, `sales`, and `loading`, data cleaning and transformation processes were crucial to ensure data quality and integrity. During initial data profiling, we identified several instances of junk data in the raw tables, such as invalid dates, unexpected text entries, and inconsistencies in data formats. To address these issues and prepare the data for analytical use, we implemented specific cleaning and transformation steps for each pipeline.

For the pax pipeline,
we found that the `Date` column, expected to contain dates in the format 'DD.MM.YYYY', occasionally had invalid entries such ('Gesamtsumme'). We filtered these out using a regular expression to match the correct date format, ensuring only valid date entries were included in the `precore` table.

In the sales pipeline, we encountered issues with various data formats and inconsistencies within the JSON files. We employed a series of transformations to parse these JSON files correctly, handling escape characters and spaces in keys.

The loading pipeline had similar challenges, particularly with the `Flight_Month_Year` column, which sometimes contained irrelevant text entries like 'ZG Orders and Invoices' instead of the expected month-year values (e.g., '42019', '82024').

```python
def load_to_precore_callable(**kwargs):
    client = bigquery.Client()
    query = f"""
    INSERT INTO `{PRECORE_PROJECT_NAME}.{PRECORE_DATASET_NAME}.{PRECORE_TABLE_NAME}`
    SELECT *, CURRENT_TIMESTAMP() as ingestion_time
    FROM `{BQ_PROJECT_NAME}.{BQ_DATASET_NAME}.{BQ_TABLE_NAME}`
    WHERE REGEXP_CONTAINS(Flight_Month_Year, r'^\d{1,2}\d{4}$')
    """
    client.query(query).result()
    print("Loaded data into precore table with an additional timestamp.")
```

Loading pipeline - load cleaned data to precore layer

```python
def load_to_precore_callable(**kwargs):
    client = bigquery.Client()
    query = f"""
    INSERT INTO `{PRECORE_PROJECT_NAME}.{PRECORE_DATASET_NAME}.{PRECORE_TABLE_NAME}`
    SELECT *, CURRENT_TIMESTAMP() as ingestion_time
    FROM `{BQ_PROJECT_NAME}.{BQ_DATASET_NAME}.{BQ_TABLE_NAME}`
    WHERE REGEXP_CONTAINS(Date, r'^\d{2}\.\d{2}\.\d{4}$')
    """
    client.query(query).result()
    print("Loaded data into precore table with an additional timestamp.")
```

pax pipeline - load cleaned data to precore layer

5. Micro Batching using Cloud functions

Deploying a cloud function

In our pipelines, we used Cloud Functions to trigger Cloud Composer DAGs whenever storage created/finalized events occurred. By following GCP documentation, we set up Cloud Functions to monitor changes in a Cloud Storage bucket. When a new file was added or an existing one was updated, the Cloud Function triggered the corresponding DAG through the Airflow REST API. This method allowed us to process data promptly and keep our pipelines responsive.

- Enabled necessary APIs, including the Airflow REST API.
- Configured Cloud Functions to trigger on changes in the Cloud Storage bucket.
- Uploaded and set up DAGs to be triggered by these functions.
- Ensured API access by configuring Webserver Access Control.
- Tested by uploading files and checking the Airflow interface for results.

requirements.txt

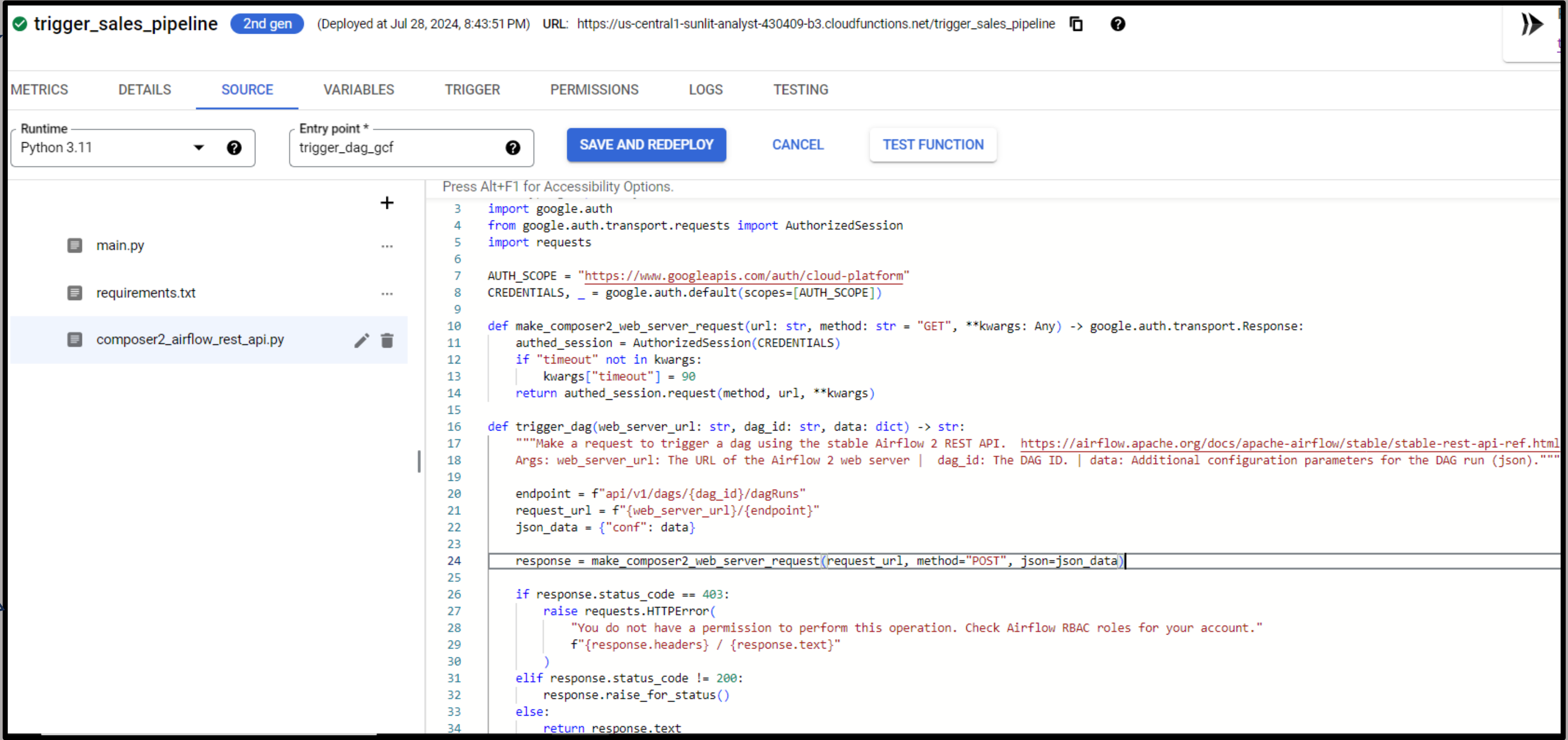
functions-framework==3.*
google-auth==2.19.1
requests==2.32.2

composer2_airflow_rest_api.py

Pass below url for composer web server and dag_id we want to trigger from main.py

web_server_url = ("https://c67920cb09d149e5885ebbd0cc6c6b07-dot-us-central1.composer.googleusercontent.com")

dag_id = 'composer_loading_sales_pipeline'



Trigger based on Storage Event
This JSON message is a log entry generated when a Cloud Storage event triggers a Cloud Run service, specifically for the "trigger-sales-pipeline."

Key Points:
1. Event Source: A POST request from the Google Cloud Storage event notification system.
2. Target: The Cloud Run service "trigger-sales-pipeline" in the "us-central1" region.
3. Event Details: The event was logged at "2024-07-28T15:17:54.896939Z" with a status code of 200 (success).
4. Context: The request triggered the Cloud Function configured to handle **sales** pipeline events.

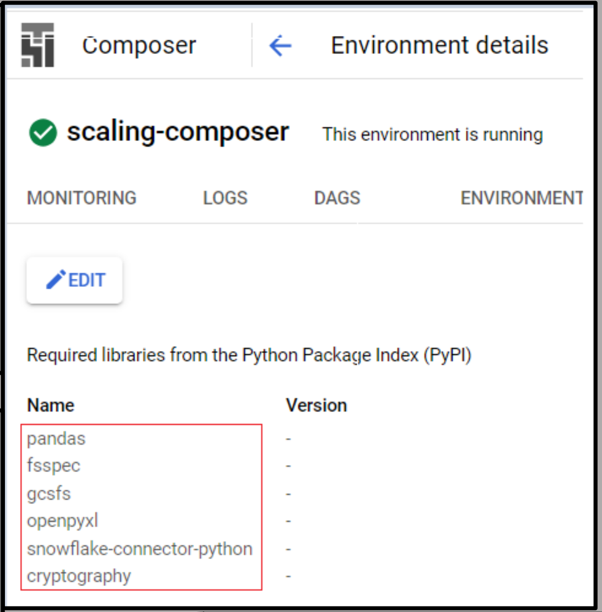
Summary: This log entry records a successful trigger of the sales pipeline via a Cloud Storage event notification, processed by a Cloud Run service in response to a storage event.


```json
{
  "insertId": "66a661230007f69a4cd11adb",
  "httpRequest": {
    "requestMethod": "POST",
    "requestUrl": "https://trigger-sales-pipeline-3imc6ruhsq-uc.a.run.app/?__GCP_CloudEventsMode=GCS_NOTIFICATION",
    "requestSize": "3198",
    "status": 200,
    "responseSize": "130",
    "userAgent": "APIs-Google; (+https://developers.google.com/webmasters/APIs-Google.html)",
    "remoteIp": "64.233.172.161",
    "serverIp": "216.239.34.53",
    "latency": "0.606683273s",
    "protocol": "HTTP/1.1"
  },
  "resource": {
    "type": "cloud_run_revision",
    "labels": {
      "location": "us-central1",
      "project_id": "sunlit-analyst-430409-b3",
      "service_name": "trigger-sales-pipeline",
      "configuration_name": "trigger-sales-pipeline",
      "revision_name": "trigger-sales-pipeline-00001-tux"
    }
  },
  "timestamp": "2024-07-28T15:17:54.896939Z",
  "severity": "INFO",
  "labels": {
    "instanceId": "0087244a8001d26599356b69828a21ec1062340ff618c8ae2a51edb3671c3dc6950579eaa862604f6614c07d2c92a84dde86c76b07d2b063bf8025e8c944aa7afc87",
    "goog-managed-by": "cloudfunctions"
  },
  "logName": "projects/sunlit-analyst-430409-b3/logs/run.googleapis.com%2Frequests",
  "trace": "projects/sunlit-analyst-430409-b3/traces/a89a3fcc5d39a9083737b61f6eec36a0",
  "receiveTimestamp": "2024-07-28T15:17:55.524780864Z",
  "spanId": "9770553742873758851",
  "traceSampled": true
}
```

6. Infrastructure setup and configuration

External package dependencies on Composer

- Cloud Composer images contains both preinstalled and custom PyPI packages. Preinstalled PyPI packages are packages that are included in the Cloud Composer image of our environment. Each Cloud Composer image contains PyPI packages that are specific for our version of Cloud Composer and Airflow.
- Custom PyPI packages are packages that we can install in our environment in addition to preinstalled packages.

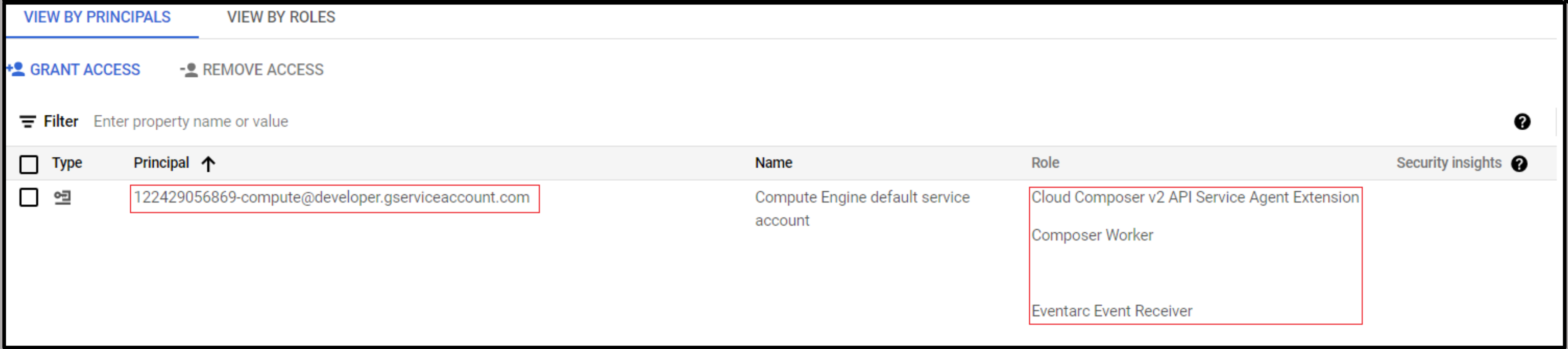


IAM considerations:

Cloud Composer uses Identity and Access Management (IAM) for access control.
We control access to different Cloud Composer features by granting roles and permissions both for IAM service accounts and for user accounts in your Google Cloud project.
Cloud Composer uses two types of IAM service accounts:

- Cloud Composer Service Agent account
- Environment's service account

In addition to these two types of service account, Google APIs Service Agent runs internal Google processes on your behalf.



Used default compute service account to manage permissions for Google Cloud Composer. This account has been granted permissions to access cloud storage, BigQuery and other GCP services.

For production environments, it's recommended that we set up a user-managed service account for Cloud Composer environments. Grant a role that is specific for Cloud Composer to this account. Afterwards, specify this service account when creating new environments.

7. Leveraging Snowflake for Machine Learning Workloads (optional)

In a modern enterprise, data is generated and stored across multiple cloud platforms, and this fragmented data storage poses a challenge for data scientists who need to access, aggregate, and analyze data from various sources to develop and deploy machine learning (ML) models effectively.

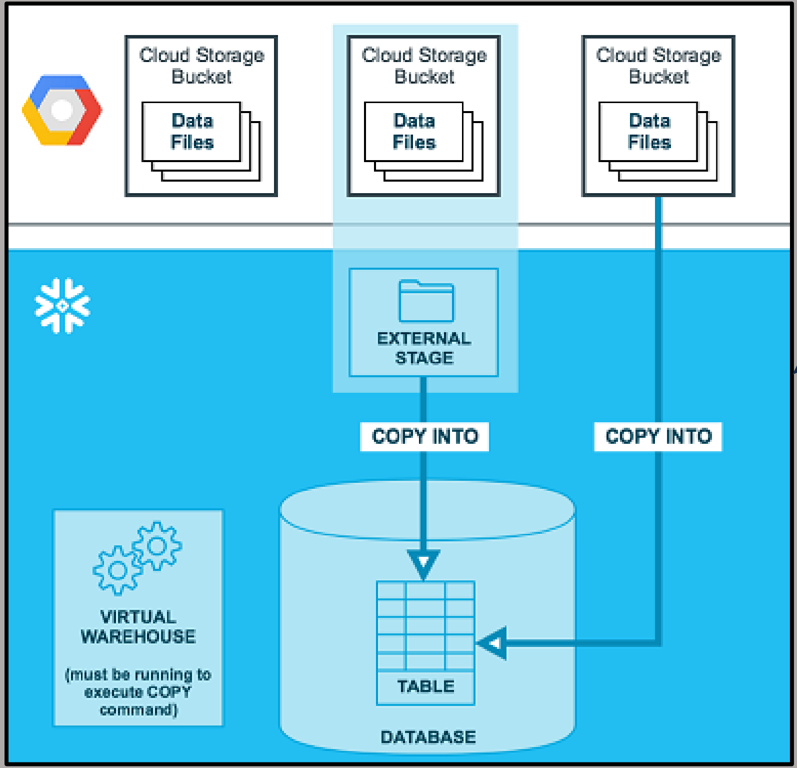
Why Snowflake as the Enterprise Cloud Data Warehouse ?

- Unified Data Platform (SaaS for all public clouds)
- Scalability and Performance Data (autoscaling, performance comes In T-shirt Sizes Like S, M, L, XL…)
- Sharing and Collaboration (modern features like Cloning, Data Shares, AI & ML Products, extensive range of connectors)
- Advanced Analytics Capabilities (in-built change data capture (CDC), serverless automations through Tasks & Streams, Partner Connect, BI, CI/CD)

To address the data integration challenge and enable effective ML workloads, we set up a pipeline that exports data from Google Cloud Storage (GCS) to Snowflake. This allows data scientists to access and analyze data stored in GCP seamlessly

To address the data integration challenge and enable effective ML workloads, we set up a pipeline that exports data from Google Cloud Storage (GCS) to Snowflake. This allows data scientists to access and analyze data stored in GCP seamlessly.

1. Data Aggregation in BigQuery
2. Export Data to GCS
3. Setup External Stage in Snowflake
4. Load Data into Snowflake
5. Data Transformation and Analysis in Snowflake
6. Machine Learning Workloads



As illustrated in the diagram below, loading data from a Cloud Storage bucket is performed in two steps:

Step 1
Snowflake assumes the data files have already been staged in a Cloud Storage bucket. If they haven't been staged yet, use the upload interfaces/utilities provided by Google to stage the files.

Step 2
Use the COPY INTO <table> command to load the contents of the staged file(s) into a Snowflake database table. You can load directly from the bucket, but Snowflake recommends creating an external stage that references the bucket and using the external stage instead.

- AIRLINE
  - INFORMATION_SCHEMA
  - INSIGHTS
    - Tables
      - FOOD_CONSUMPTION_METRICS
      - MY_FORECASTS_2024_07_31
    - Views
      - FOOD_CONSUMPTION_METRICS_V1
    - Stages
      - EIDZRKE5N4ED8XHB (Stage)
      - G_DM0HIHSJ972IU (Stage)
  - PUBLIC
- APP_DB
- SNOWFLAKE
- SNOWFLAKE_SAMPLE_DATA

Project structure and Objects on Snowflake DB

Loading Data into a Snowflake from an external GCP bucket involves following

1. Create a Cloud Storage Integration in Snowflake

```
CREATE STORAGE INTEGRATION GCP_INTEGRATION
TYPE = EXTERNAL_STAGE
STORAGE_PROVIDER = 'GCS'
ENABLED = TRUE
STORAGE_ALLOWED_LOCATIONS = ('gcs://outbound-world/metrics');
```

2. Retrieve the Cloud Storage Service Account for your Snowflake Account

```
DESC STORAGE INTEGRATION GCP_INTEGRATION;
--STORAGE_GCP_SERVICE_ACCOUNT=kooaoyasmn@azcentralindia-1-5514.iam.gserviceaccour
```

3. Grant the Service Account Permissions to Access Bucket Objects

Authorize this service account on gcp IAM & admin portal with required roles - snow_role with specific permissions for storage listed in the image)

snow_role EDIT ROLE CREATE FROM ROLE

ID	projects/sunlit-analyst-430409-b3/roles/CustomRole
Role launch stage	Alpha

Description

load data to snowflake

5 assigned permissions

storage.buckets.get
storage.objects.create
storage.objects.delete
storage.objects.get
storage.objects.list

4. Create an external stage

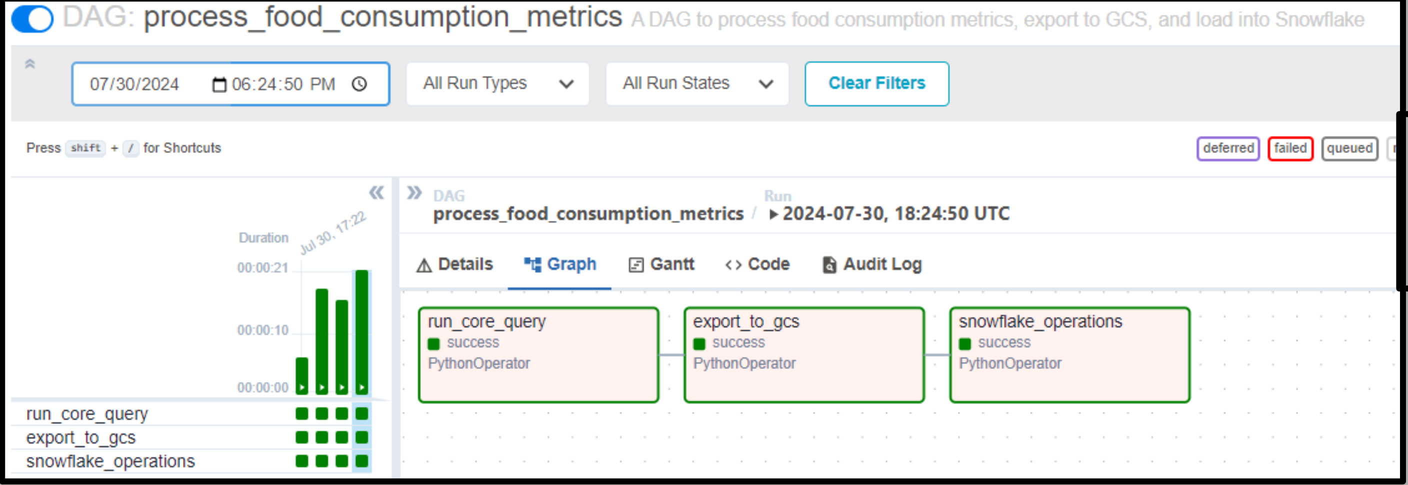
```
CREATE OR REPLACE STAGE APP_DB.APP_SCHEMA.GCS_STAGE
URL = 'gcs://outbound-world/metrics'
STORAGE_INTEGRATION = GCP_INTEGRATION;
```

5. Run Copy Into statement to perform Data Load

```
COPY INTO AIRLINE.INSIGHTS.FOOD_CONSUMPTION_METRICS
FROM @GCS_STAGE
FILE_FORMAT =
   (TYPE = 'CSV'
     FIELD_OPTIONALLY_ENCLOSED_BY = '"'
     FIELD_DELIMITER = ','
     SKIP_HEADER = 1)
PATTERN = '.*snow-exports_.*\\.csv';
```

copy into statement output

	file	status	rows_parsed	rows_loaded	error_limit	errors_seen
1	gcs://outbound-world/metrics/snow-exports_07302024.csv	LOADED	711	711	1	0



Daily pipeline to load data to Snowflake from cloud storage

This daily pipeline calculates the metric in core layer, exports the data to cloud storage bucket, and then finally the data is written to snowflake db table *AIRLINE.INSIGHTS.FOOD_CONSUMPTION_METRICS* using an external stage.

```python
# 1. Exports core table data to GCS staging location in outbound bucket
# 2. Creates files with prefix_[date_part].csv pattern
def export_to_gcs():
    from google.cloud import storage
    import datetime

    client = bigquery.Client()
    bucket_name = 'outbound-world'
    file_name = f'snow-exports_{datetime.datetime.now().strftime("%m%d%Y")}.csv'

    query = "SELECT * FROM `sunlit-analyst-430409-b3.core.food_consumption_metrics`"
    df = client.query(query).to_dataframe()

    local_file_path = '/tmp/metrics.csv'
    df.to_csv(local_file_path, index=False)

    storage_client = storage.Client()
    bucket = storage_client.bucket(bucket_name)
    blob = bucket.blob(f'metrics/{file_name}')
    blob.upload_from_filename(local_file_path)
```

```python
def get_snwflk_ctx():
    local_rsa_key_path = '/tmp/rsa_key.p8'
    rsa_key_path = 'gs://airline_inbound_data/requirements/rsa_key.p8'
    storage_client = storage.Client()
    bucket = storage_client.bucket('airline_inbound_data')
    blob = bucket.blob('requirements/rsa_key.p8')
    blob.download_to_filename(local_rsa_key_path)

    with open(local_rsa_key_path, "rb") as key:
        p_key = serialization.load_pem_private_key(key.read(), password=None, backend=default_backend())

    pkb = p_key.private_bytes(encoding=serialization.Encoding.DER,
                              format=serialization.PrivateFormat.PKCS8,
                              encryption_algorithm=serialization.NoEncryption())

    ctx = snowflake.connector.connect(
        user='VISUALINSIGHTS',
        password='DataMatters123',
        account='gtorzpp-yw14931',
        private_key=pkb,
        warehouse='COMPUTE_WH',
        database='APP_DB',
        schema='APP_SCHEMA')
    return ctx

conn = get_snwflk_ctx()

copy_sql = f"""
COPY INTO AIRLINE.INSIGHTS.FOOD_CONSUMPTION_METRICS
FROM @GCS_STAGE
FILE_FORMAT = (TYPE = 'CSV' FIELD_OPTIONALLY_ENCLOSED_BY = '"' FIELD_DELIMITER = ',' SKIP_HEADER = 1)
PATTERN = '.*snow-exports_.*\\.csv';
"""
df = pd.read_sql_query(copy_sql, conn)
print(df)
conn.close()
```

Streamlit is an open-source Python library that makes it easy to create and share custom web apps for machine learning and data science. By using Streamlit we can quickly build and deploy powerful data applications. Streamlit in Snowflake helps developers securely build, deploy, and share Streamlit apps on Snowflake's data cloud. Using Streamlit in Snowflake, we can build applications that process and use data in Snowflake without moving data or application code to an external system.

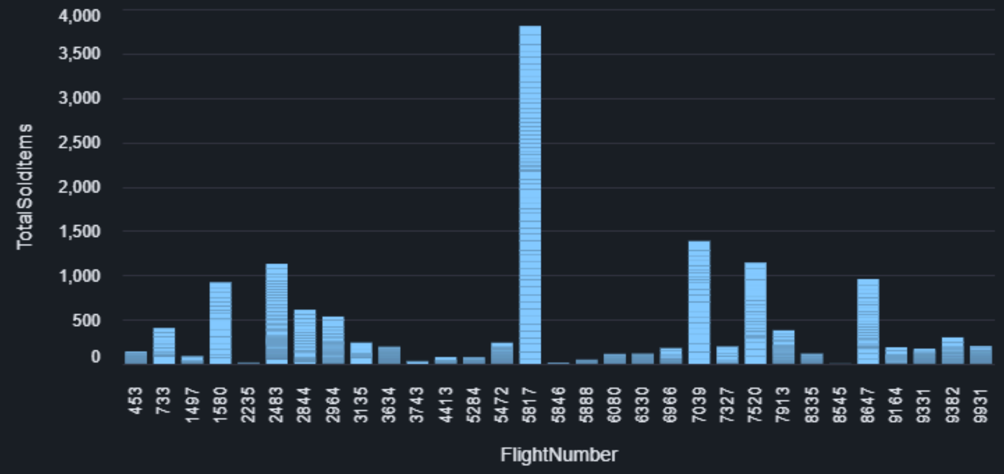
Streamlit Apps Airline_ML_Workloads

Share Run

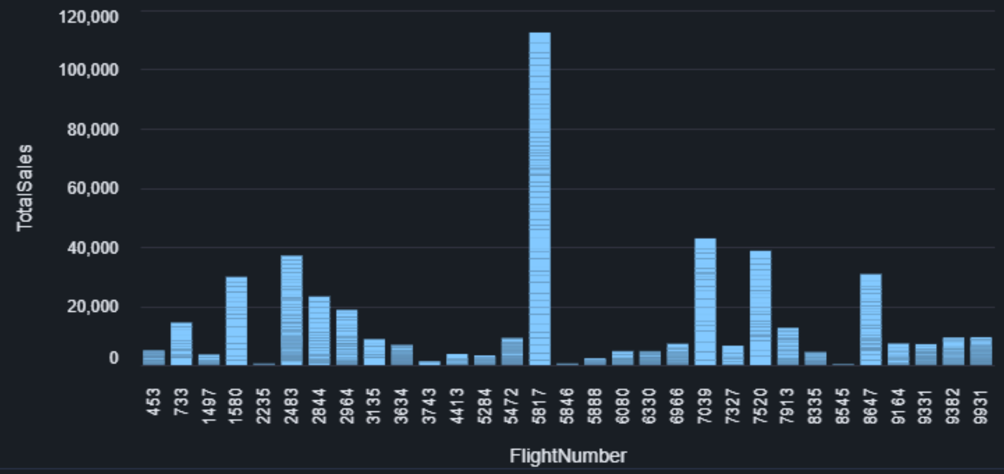
Packages

```python
# Import required packages
import streamlit as st
from snowflake.snowpark.context import get_active_session
from snowflake.snowpark.functions import col
import pandas as pd

# Set the title of the Streamlit app
st.title("Food Consumption Metrics Dashboard")

# Initialize the Snowflake session
session = get_active_session()

# Define a function to query data from Snowflake
def get_food_consumption_data():
    query = session.table("AIRLINE.INSIGHTS.FOOD_CONSUMPTION_METRICS")
    df = query.select(
        col("FLIGHTNUMBER"),
        col("FLIGHTDATE"),
        col("TOTALPASSENGERS"),
        col("TOTALSOLDITEMS"),
        col("TOTALSALES"),
        col("AVERAGESOLDITEMSPERPASSENGER"),
        col("AVERAGESALESPERPASSENGER")
    ).to_pandas()
    return df

# Get the food consumption data from Snowflake
food_consumption_data = get_food_consumption_data()

# Display the data in a table
st.subheader("Food Consumption Metrics")
st.dataframe(food_consumption_data, use_container_width=True)

# Create a bar chart for TotalSoldItems
st.subheader("Total Sold Items per Flight")
st.bar_chart(data=food_consumption_data.rename(columns={"FLIGHTNUMBER": "FlightNumber", "TO

# Create a bar chart for TotalSales
st.subheader("Total Sales per Flight")
st.bar_chart(data=food_consumption_data.rename(columns={"FLIGHTNUMBER": "FlightNumber", "TO

# Create a bar chart for AverageSoldItemsPerPassenger
st.subheader("Average Sold Items per Passenger")
```

Total Sold Items per Flight



Total Sales per Flight



Feedback

8. Summary and Appendices

Summary:

1. Data Loading: Loaded sales, passenger (pax), and loading data from Google Cloud Storage (GCS) to BigQuery tables for March and April 2019.
2. Data Transformation: Created SQL queries to transform data in BigQuery, including aggregating sales and passenger data and adding a timestamp field.
3. Pipeline Design: Developed Airflow DAGs in Google Cloud Composer for data ingestion, transformation, & loading into BigQuery.
4. Data Export: Built a pipeline to export processed metrics from BigQuery to GCS & integrated it with Snowflake for further analysis, handling RSA key connections.
5. Architecture: Infrastructure setup and configurations, discussed IAM considerations and external packaged dependencies and service accounts.
6. Future plans: Enhance the Streamlit app using Snowpark to visualize metrics data directly from Snowflake, keeping it separate from Composer workflows.

Reference Links:

Cloud Composer: https://cloud.google.com/composer?hl=en
Cloud Storage: https://cloud.google.com/storage?hl=en
Trigger DAGs with Cloud functions: https://cloud.google.com/composer/docs/composer-3/triggering-with-gcf
Snowflake-GCP integration: https://docs.snowflake.com/en/user-guide/data-load-gcs-config
Streamlit - https://docs.snowflake.com/developer-guide/streamlit/about-streamlit

---

Developed by Prashant Dixit.