Code Documentation

Group-44
Richard David (M24CSE019)
Sachin Singh (M24CSE033)
Vishwanath Singh (M24CSE030)

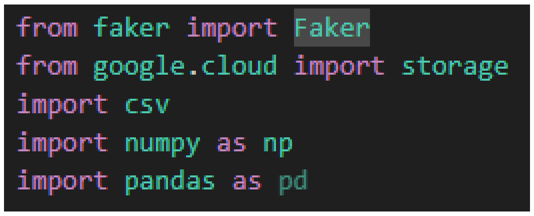
Sales Data Generation and Uploading to Google Cloud Storage (GCS)

extract.py

This Python script generates synthetic sales data for analysis purposes and uploads the generated data to Google Cloud Storage (GCS). It is important to note that, while real-time sales data from an actual business can be used in production scenarios, this script generates **synthetic data** as a placeholder to simulate sales data for development and testing purposes. This data can later be replaced with actual business sales data once available.

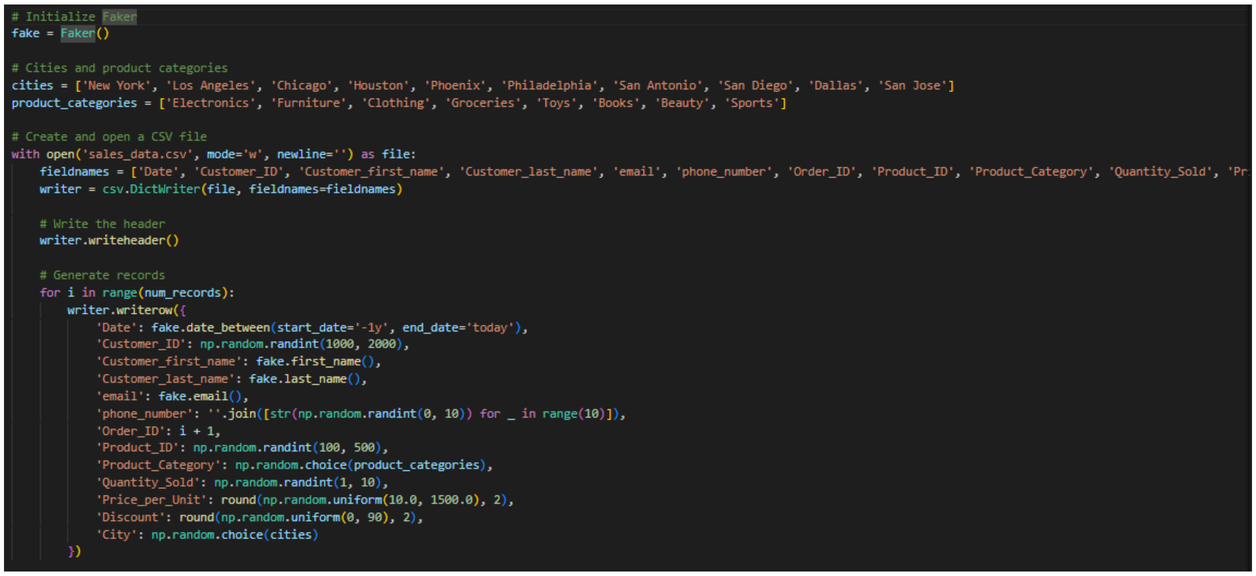
Key Components of the Script

1. Libraries Used


```
from faker import Faker
from google.cloud import storage
import csv
import numpy as np
import pandas as pd
```

- **Faker:** This library is utilized to generate realistic synthetic data, such as names, emails, phone numbers, and dates.
- **Google Cloud Storage (GCS):** This library facilitates interactions with GCS, allowing the script to upload files to the cloud.
- **csv:** This module is used for writing data into a CSV file.
- **numpy and pandas:** While primarily known for data analysis, numpy is used here for generating random numbers efficiently.

2. Data Generation Process



```
# Initialize Faker
fake = Faker()

# Cities and product categories
cities = ['New York', 'Los Angeles', 'Chicago', 'Houston', 'Phoenix', 'Philadelphia', 'San Antonio', 'San Diego', 'Dallas', 'San Jose']
product_categories = ['Electronics', 'Furniture', 'Clothing', 'Groceries', 'Toys', 'Books', 'Beauty', 'Sports']

# Create and open a CSV file
with open('sales_data.csv', mode='w', newline='') as file:
    fieldnames = ['Date', 'Customer_ID', 'Customer_first_name', 'Customer_last_name', 'email', 'phone_number', 'Order_ID', 'Product_ID', 'Product_Category', 'Quantity_Sold', 'Pr
    writer = csv.DictWriter(file, fieldnames=fieldnames)

    # Write the header
    writer.writeheader()

    # Generate records
    for i in range(num_records):
        writer.writerow({
            'Date': fake.date_between(start_date='-1y', end_date='today'),
            'Customer_ID': np.random.randint(1000, 2000),
            'Customer_first_name': fake.first_name(),
            'Customer_last_name': fake.last_name(),
            'email': fake.email(),
            'phone_number': ''.join([str(np.random.randint(0, 10)) for _ in range(10)]),
            'Order_ID': i + 1,
            'Product_ID': np.random.randint(100, 500),
            'Product_Category': np.random.choice(product_categories),
            'Quantity_Sold': np.random.randint(1, 10),
            'Price_per_Unit': round(np.random.uniform(10.0, 1500.0), 2),
            'Discount': round(np.random.uniform(0, 90), 2),
            'City': np.random.choice(cities)
        })
```

The script generates **1,000 synthetic sales records**. For each record, several data fields are created, including customer information, product details, transaction data, and geographic locations. Here's a breakdown of the process:

- **Customer Information:** Includes randomly generated customer IDs, first and last names, emails, and phone numbers.
- **Product Information:** Each record includes a product ID, product category (e.g., electronics, clothing), quantity sold, and price per unit.
- **Sales Details:** A sales record includes an order ID, discount percentage, and the city where the sale occurred. The dates for the sales are randomly chosen within the past year.

The cities and product categories are hardcoded, but in a real-world scenario, these could be retrieved dynamically from a business database.

3. Data Fields

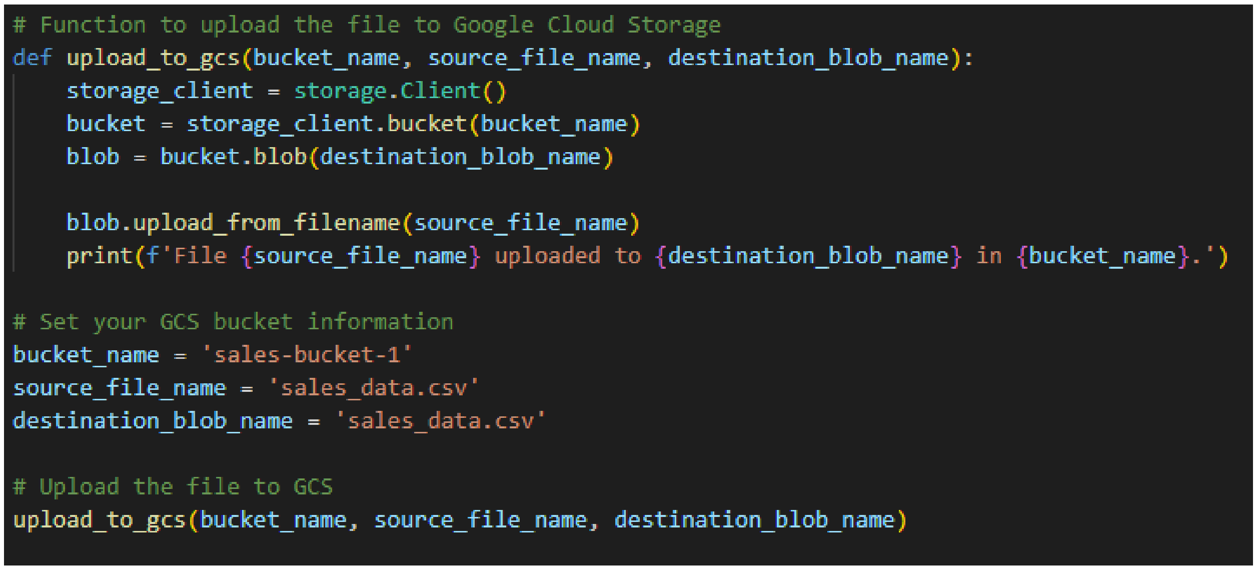
```
fieldnames = ['Date', 'Customer_ID', 'Customer_first_name', 'Customer_last_name', 'email', 'phone_number',
'Order_ID', 'Product_ID', 'Product_Category', 'Quantity_Sold', 'Price_per_Unit', 'Discount', 'City']
writer = csv.DictWriter(file, fieldnames=fieldnames)
```

Each synthetic sales record includes the following fields:

- **Date:** The sale date, randomly chosen from the last year.
- **Customer_ID:** A unique identifier for the customer.
- **Customer_first_name/Customer_last_name:** The customer's first and last names.
- **email:** The customer's email address.

- **phone_number:** The customer's phone number.
- **Order_ID:** A unique identifier for the sales order.
- **Product_ID:** A unique identifier for the product.
- **Product_Category:** The category the product belongs to (e.g., Electronics, Groceries).
- **Quantity_Sold:** The number of units sold in this transaction.
- **Price_per_Unit:** The price per unit of the product.
- **Discount:** Any applicable discount for the product.
- **City:** The city where the sale occurred.

4. Uploading to Google Cloud Storage



```python
# Function to upload the file to Google Cloud Storage
def upload_to_gcs(bucket_name, source_file_name, destination_blob_name):
    storage_client = storage.Client()
    bucket = storage_client.bucket(bucket_name)
    blob = bucket.blob(destination_blob_name)

    blob.upload_from_filename(source_file_name)
    print(f'File {source_file_name} uploaded to {destination_blob_name} in {bucket_name}.')

# Set your GCS bucket information
bucket_name = 'sales-bucket-1'
source_file_name = 'sales_data.csv'
destination_blob_name = 'sales_data.csv'

# Upload the file to GCS
upload_to_gcs(bucket_name, source_file_name, destination_blob_name)
```

Once the sales data is generated and written to a CSV file (sales_data.csv), the script uses Google Cloud Storage (GCS) to upload the file to a specified GCS bucket. The function upload_to_gcs handles the uploading process:

- **bucket_name:** The name of the GCS bucket where the file will be uploaded.
- **source_file_name:** The name of the local file to be uploaded (in this case, sales_data.csv).
- **destination_blob_name:** The name to be given to the file once uploaded to GCS.

5. Usage Instructions

Setting Up Google Cloud

1. **Install Google Cloud SDK:** Ensure that Google Cloud SDK is installed and properly configured on your machine. You must authenticate your credentials using gcloud auth login.

2. **Set up a GCS Bucket:** Create a GCS bucket (or use an existing one) to store the generated CSV file.

6. Application in Real-World Scenarios

While this script generates **synthetic data**, it is designed to simulate real-world sales data. In actual use, businesses could connect the data extraction step to real-time sales systems, such as databases, APIs, or data lakes, to replace the synthetic data with actual sales data. Once real data is integrated, businesses can conduct meaningful analyses, including customer behavior, sales trends, and product performance.

For now, this synthetic data serves as a useful placeholder for building and testing the **data pipeline**, allowing teams to refine and optimize their system before live data becomes available.

This script provides a starting point for creating a data pipeline involving data generation, processing, and storage in Google Cloud Storage. Although real-time business data is not currently available, this synthetic data allows you to prepare and test your workflows. Later, the same structure can be adapted to real-world data, supporting comprehensive sales analysis, data-driven decision-making, and strategic planning.

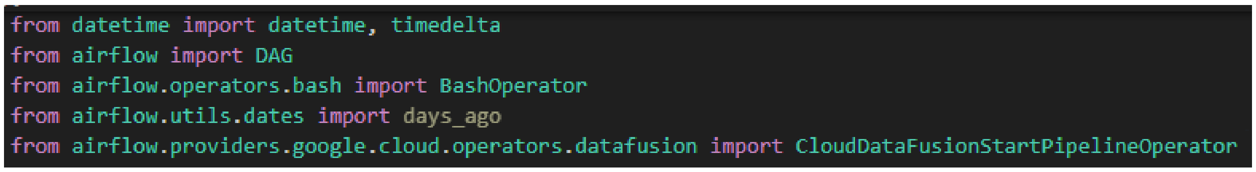
Airflow DAG for Sales Data Pipeline Orchestration

<u>dag.py</u>

This documentation outlines the steps involved in orchestrating a sales data pipeline using **Apache Airflow**. The DAG (Directed Acyclic Graph) defined in this script is responsible for running a Python script that extracts sales data and subsequently triggers a Cloud Data Fusion pipeline to transform and load the data.

Key Components

1. Libraries and Imports

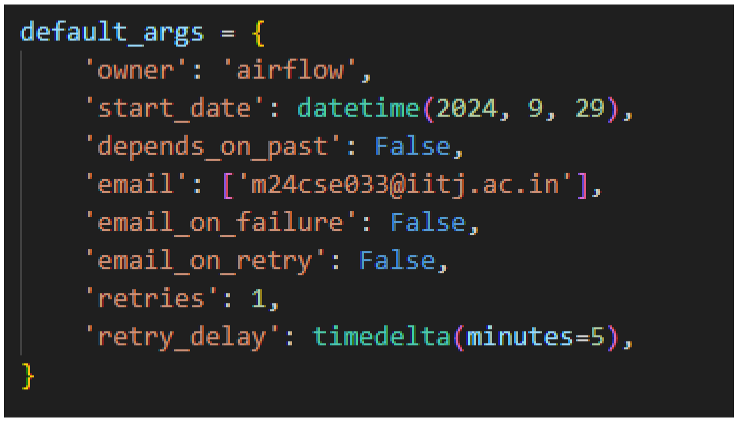


```
from datetime import datetime, timedelta
from airflow import DAG
from airflow.operators.bash import BashOperator
from airflow.utils.dates import days_ago
from airflow.providers.google.cloud.operators.datafusion import CloudDataFusionStartPipelineOperator
```

- **datetime:** Provides date and time functionalities, used to define the start date and time intervals in the DAG.
- **timedelta:** Used for specifying time delays (e.g., retry delays) within the DAG's configuration.
- **DAG (Directed Acyclic Graph):** The fundamental concept in Airflow, representing the workflow and task dependencies.

- **BashOperator:** Executes bash commands or scripts as part of an Airflow task.
- **CloudDataFusionStartPipelineOperator:** Used to trigger a Cloud Data Fusion pipeline from Airflow. This operator is provided by Google Cloud.

2. DAG Configuration

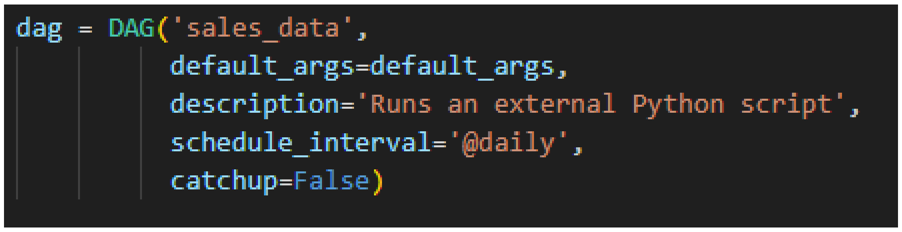


```
default_args = {
    'owner': 'airflow',
    'start_date': datetime(2024, 9, 29),
    'depends_on_past': False,
    'email': ['m24cse033@iitj.ac.in'],
    'email_on_failure': False,
    'email_on_retry': False,
    'retries': 1,
    'retry_delay': timedelta(minutes=5),
}
```

The DAG is defined with default parameters that include the following:

- **owner:** Specifies the owner of the DAG (set to 'airflow' by default).
- **start_date:** The date from which the DAG should start running. In this case, it is set to **September 29, 2024**.
- **depends_on_past:** If set to True, the current task will run only after the previous one is successful. Here, it is False to allow independent task execution.
- **email:** Specifies an email address where notifications should be sent in case of failures or retries. (The email address is 'm24cse033@iitj.ac.in').
- **retries:** Defines how many times a task should be retried upon failure. Set to 1 in this case.
- **retry_delay:** Specifies the delay before retrying a failed task. Set to 5 minutes using timedelta(minutes=5).

3. DAG Schedule and Catchup

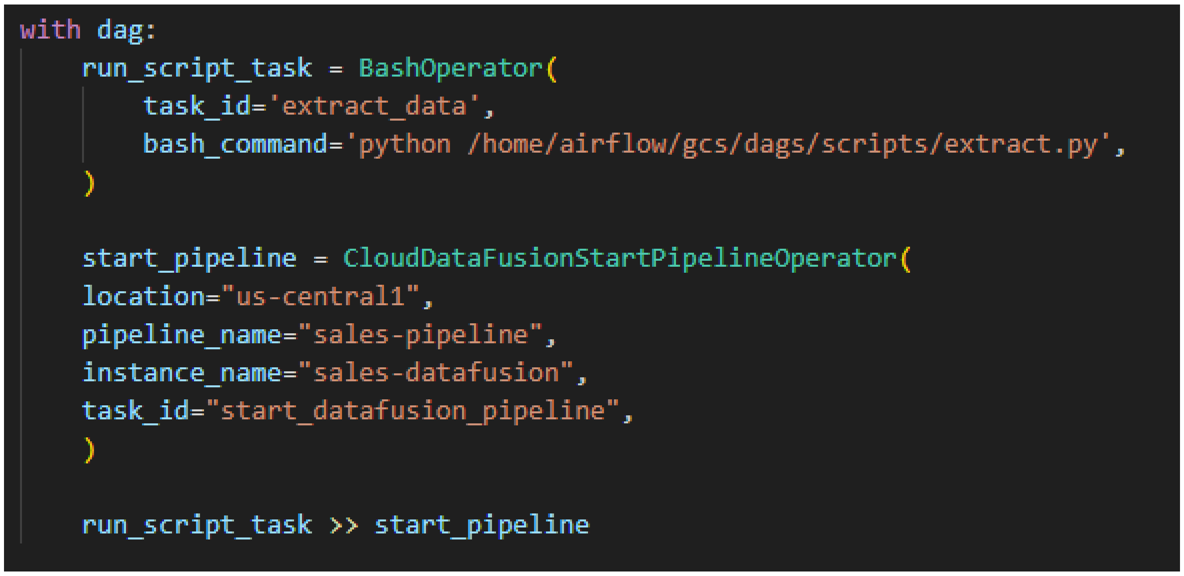


```
dag = DAG('sales_data',
          default_args=default_args,
          description='Runs an external Python script',
          schedule_interval='@daily',
          catchup=False)
```

- **schedule_interval:** The DAG is scheduled to run **daily** using the interval '@daily'. This means that the pipeline will execute once every day.

- **catchup:** This is set to False, which ensures that the DAG only runs from the defined start_date onwards and does not attempt to run for previous untriggered dates.

4. Task 1: Running the Data Extraction Script



```
with dag:
    run_script_task = BashOperator(
        task_id='extract_data',
        bash_command='python /home/airflow/gcs/dags/scripts/extract.py',
    )

    start_pipeline = CloudDataFusionStartPipelineOperator(
    location="us-central1",
    pipeline_name="sales-pipeline",
    instance_name="sales-datafusion",
    task_id="start_datafusion_pipeline",
    )

    run_script_task >> start_pipeline
```

- **BashOperator:** The first task, run_script_task, uses the BashOperator to run a Python script located at /home/airflow/gcs/dags/scripts/extract.py.
- The purpose of this task is to extract the sales data (likely through interaction with a database, API, or other data source) and prepare it for the next stages of the pipeline.

5. Task 2: Triggering the Cloud Data Fusion Pipeline

- **CloudDataFusionStartPipelineOperator:** After the data extraction script runs, the DAG triggers a Cloud Data Fusion pipeline using this operator.
  - The pipeline is located in the **us-central1** region.
  - The Cloud Data Fusion pipeline is named **sales-pipeline**, and it is executed on the Cloud Data Fusion instance **sales-datafusion**.

The operator sends a request to start the data pipeline in Cloud Data Fusion, where the data transformation and loading processes occur.

6. Task Dependencies

The workflow follows a linear sequence:

- **run_script_task → start_pipeline:** The DAG ensures that the Cloud Data Fusion pipeline will only start once the data extraction script has successfully executed.

7. DAG Usage Instructions

Steps to Use This DAG:

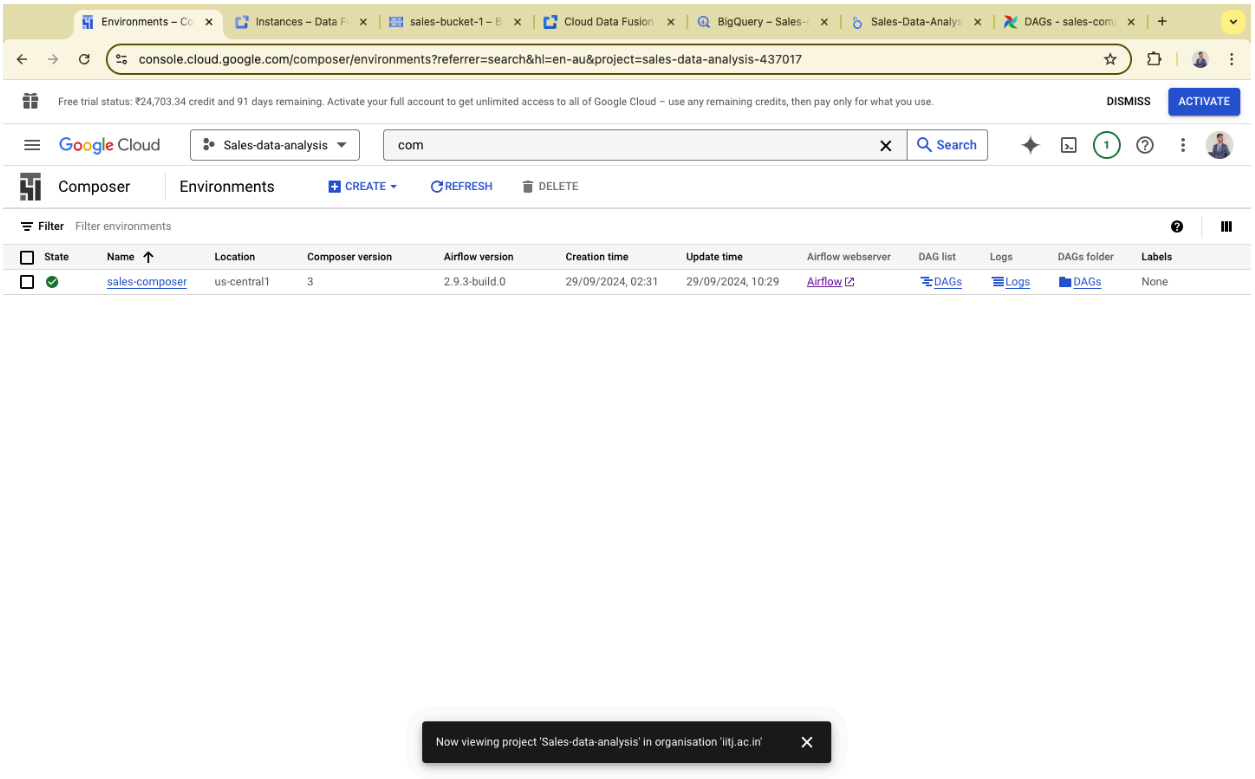
1. **Place Python Script in GCS:**
   - Ensure that the Python extraction script (extract.py) is placed in the appropriate directory on Google Cloud Storage (GCS). The path used in this DAG is /home/airflow/gcs/dags/scripts/extract.py.
2. **Create Cloud Data Fusion Pipeline:**
   - Ensure that a Cloud Data Fusion instance (sales-datafusion) and pipeline (sales-pipeline) are correctly set up. This pipeline should handle the transformation and loading of sales data into Google BigQuery or another destination.
3. **Run the DAG:**
   - Once the script and Data Fusion pipeline are in place, the Airflow scheduler will automatically run the DAG based on the schedule (@daily), extracting the sales data and triggering the Cloud Data Fusion pipeline daily.

This Airflow DAG automates the orchestration of a sales data pipeline by running a Python data extraction script and triggering a Cloud Data Fusion pipeline. The workflow is designed to ensure that data flows smoothly and efficiently through extraction, transformation, and loading stages. By utilizing Airflow's scheduling and monitoring features, the pipeline can run reliably and be easily maintained over time.

Steps

Step-by-Step Guide for Sales Data ETL Pipeline Project

Step 1: Create a Cloud Composer Environment


console.cloud.google.com/composer/environments?referrer=search&hl=en-au&project=sales-data-analysis-437017

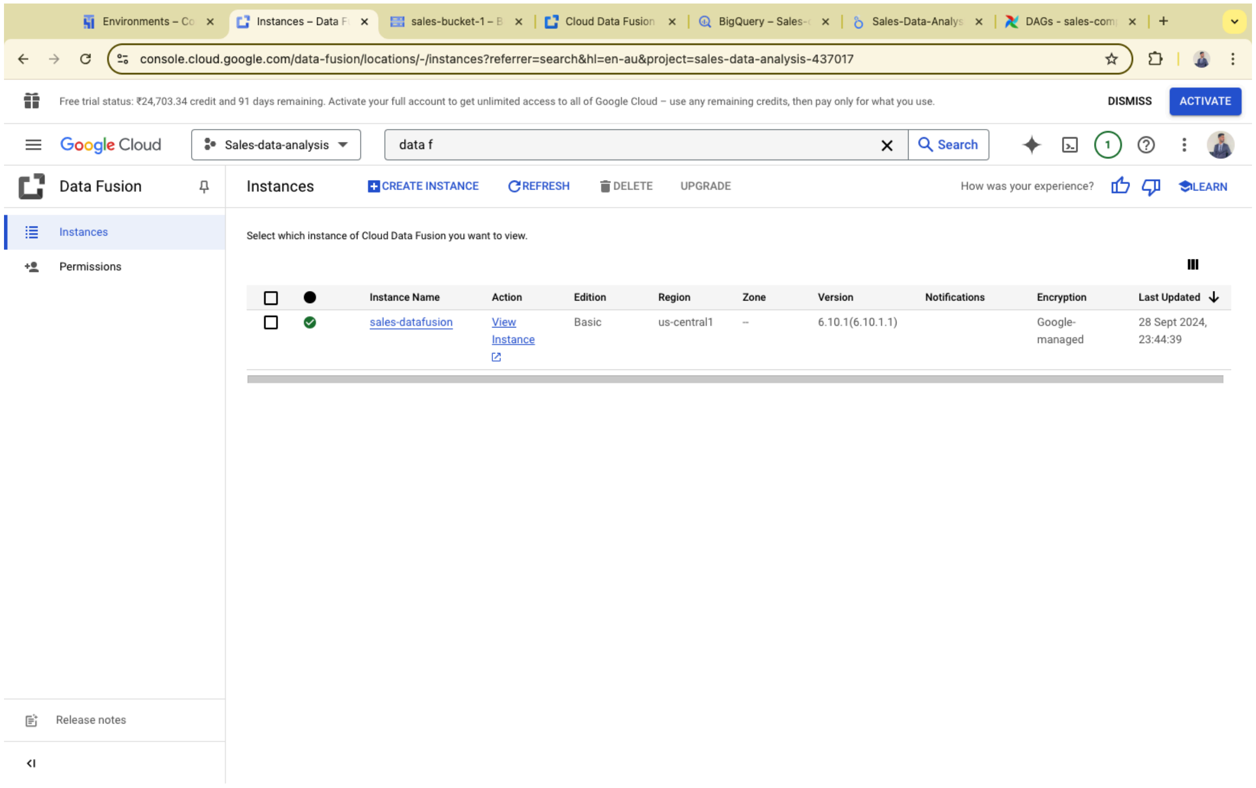
Free trial status: ₹24,703.34 credit and 91 days remaining. Activate your full account to get unlimited access to all of Google Cloud – use any remaining credits, then pay only for what you use.

State	Name	Location	Composer version	Airflow version	Creation time	Update time	Airflow webserver	DAG list	Logs	DAGs folder	Labels
✓	sales-composer	us-central1	3	2.9.3-build.0	29/09/2024, 02:31	29/09/2024, 10:29	Airflow	DAGs	Logs	DAGs	None

Now viewing project 'Sales-data-analysis' in organisation 'iitj.ac.in'

1. Open the **Google Cloud Console**.
2. Navigate to **Cloud Composer** from the navigation menu.
3. Click on **Create environment**.
4. Set your environment name and choose the **location** (e.g., us-central1).
5. Configure settings such as **Node count** and **Machine type** based on the scale of your DAG tasks.
6. Click **Create** and wait for the environment to be provisioned (this might take a few minutes).

Step 2: Create a Cloud Data Fusion Instance

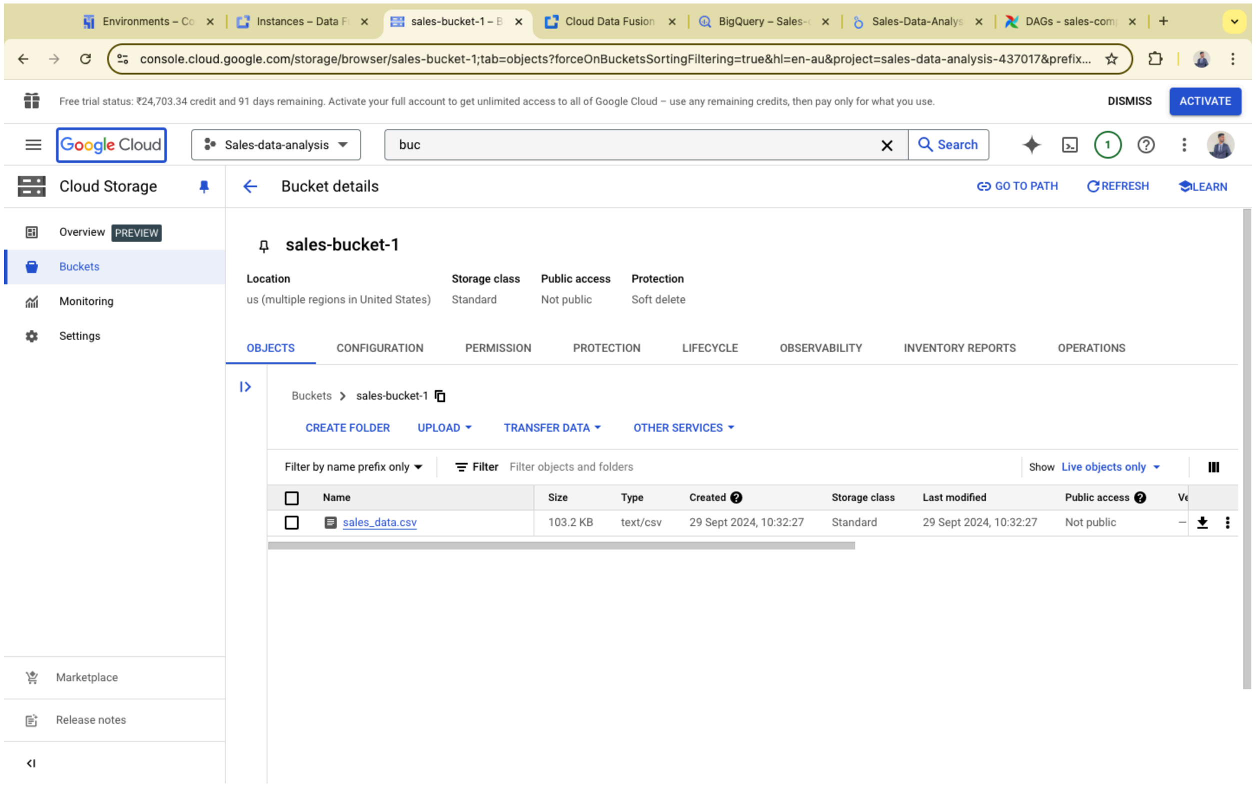

Free trial status: ₹24,703.34 credit and 91 days remaining. Activate your full account to get unlimited access to all of Google Cloud – use any remaining credits, then pay only for what you use.

Select which instance of Cloud Data Fusion you want to view.

Instance Name	Action	Edition	Region	Zone	Version	Notifications	Encryption	Last Updated
sales-datafusion	View Instance	Basic	us-central1	–	6.10.1(6.10.1.1)		Google-managed	28 Sept 2024, 23:44:39

1. From the **Google Cloud Console**, search for **Cloud Data Fusion**.
2. Click **Create Instance**.
3. Select the appropriate **region** (e.g., us-central1), and provide a unique instance name (e.g., sales-datafusion).
4. Choose **Basic** or **Enterprise** edition based on your needs.
5. Click **Create**. Once the instance is ready, you'll be able to start creating pipelines.

Step 3: Create a Google Cloud Storage (GCS) Bucket


console.cloud.google.com/storage/browser/sales-bucket-1;tab=objects?forceOnBucketsSortingFiltering=true&hl=en-au&project=sales-data-analysis-437017&prefix...

Free trial status: ₹24,703.34 credit and 91 days remaining. Activate your full account to get unlimited access to all of Google Cloud – use any remaining credits, then pay only for what you use.

Cloud Storage — Bucket details

sales-bucket-1

Location	Storage class	Public access	Protection
us (multiple regions in United States)	Standard	Not public	Soft delete

OBJECTS | CONFIGURATION | PERMISSION | PROTECTION | LIFECYCLE | OBSERVABILITY | INVENTORY REPORTS | OPERATIONS

Buckets > sales-bucket-1

Name	Size	Type	Created	Storage class	Last modified	Public access
sales_data.csv	103.2 KB	text/csv	29 Sept 2024, 10:32:27	Standard	29 Sept 2024, 10:32:27	Not public

1. In the **Google Cloud Console**, go to **Cloud Storage** and click **Create bucket**.
2. Provide a globally unique name for your bucket (e.g., sales-bucket-1).
3. Choose a **region** matching your Data Fusion instance and Composer environment.
4. Select appropriate **storage class** (e.g., Standard).
5. Click **Create** to finish setting up your bucket.

Step 4: Run Python Code and Check the Bucket

1. Ensure your Python script for generating sales data (extract.py) is ready.
2. Use your local machine or an instance (e.g., Cloud Shell) to run the code and generate the CSV file.
3. Upload the generated sales_data.csv to the GCS bucket (sales-bucket-1).

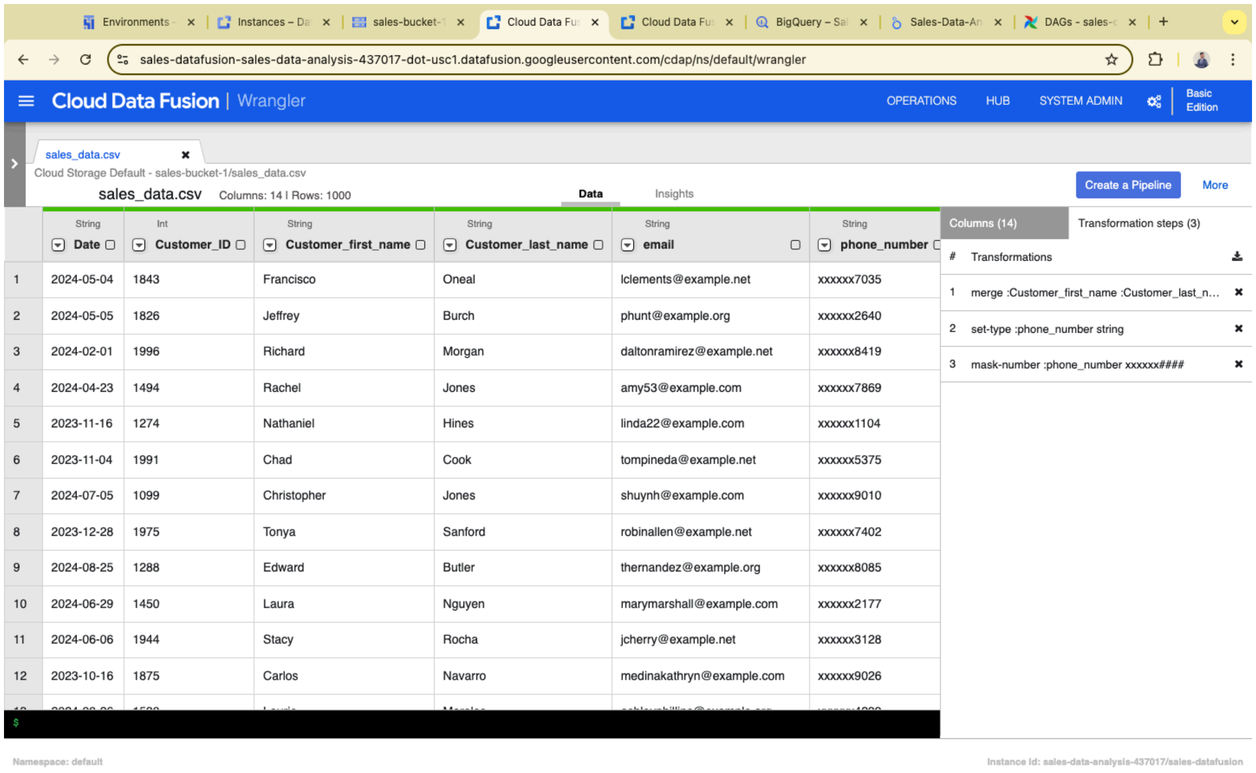
bash

Copy code

```
gsutil cp sales_data.csv gs://sales-bucket-1/sales_data.csv
```

4. Verify that the file is successfully uploaded by checking the **Cloud Storage** bucket.

Step 5: Perform Data Transformation (Masking Personal Information)



Cloud Storage Default - sales-bucket-1/sales_data.csv

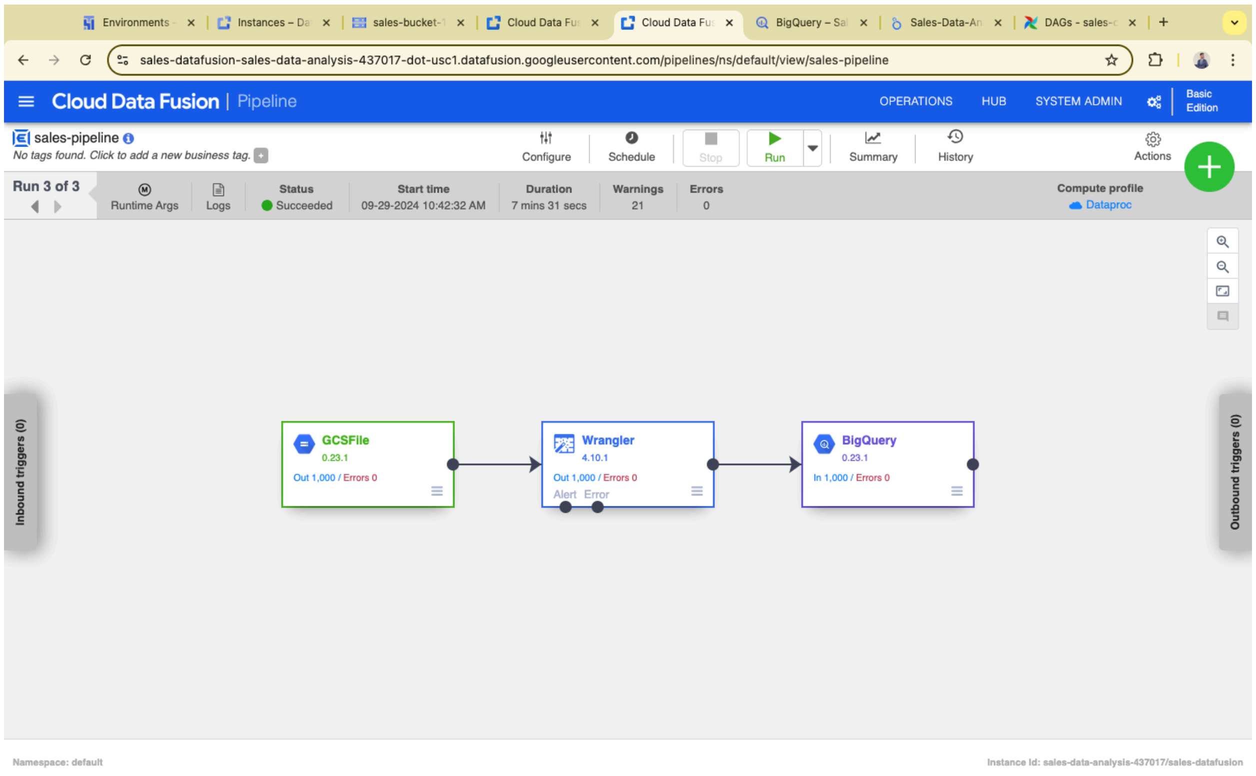
sales_data.csv Columns: 14 I Rows: 1000

	String Date	Int Customer_ID	String Customer_first_name	String Customer_last_name	String email	String phone_number
1	2024-05-04	1843	Francisco	Oneal	lclements@example.net	xxxxxx7035
2	2024-05-05	1826	Jeffrey	Burch	phunt@example.org	xxxxxx2640
3	2024-02-01	1996	Richard	Morgan	daltonramirez@example.net	xxxxxx8419
4	2024-04-23	1494	Rachel	Jones	amy53@example.com	xxxxxx7869
5	2023-11-16	1274	Nathaniel	Hines	linda22@example.com	xxxxxx1104
6	2023-11-04	1991	Chad	Cook	tompineda@example.net	xxxxxx5375
7	2024-07-05	1099	Christopher	Jones	shuynh@example.com	xxxxxx9010
8	2023-12-28	1975	Tonya	Sanford	robinallen@example.net	xxxxxx7402
9	2024-08-25	1288	Edward	Butler	thernandez@example.org	xxxxxx8085
10	2024-06-29	1450	Laura	Nguyen	marymarshall@example.com	xxxxxx2177
11	2024-06-06	1944	Stacy	Rocha	jcherry@example.net	xxxxxx3128
12	2023-10-16	1875	Carlos	Navarro	medinakathryn@example.com	xxxxxx9026

#	Transformations
1	merge :Customer_first_name :Customer_last_n…
2	set-type :phone_number string
3	mask-number :phone_number xxxxxx####

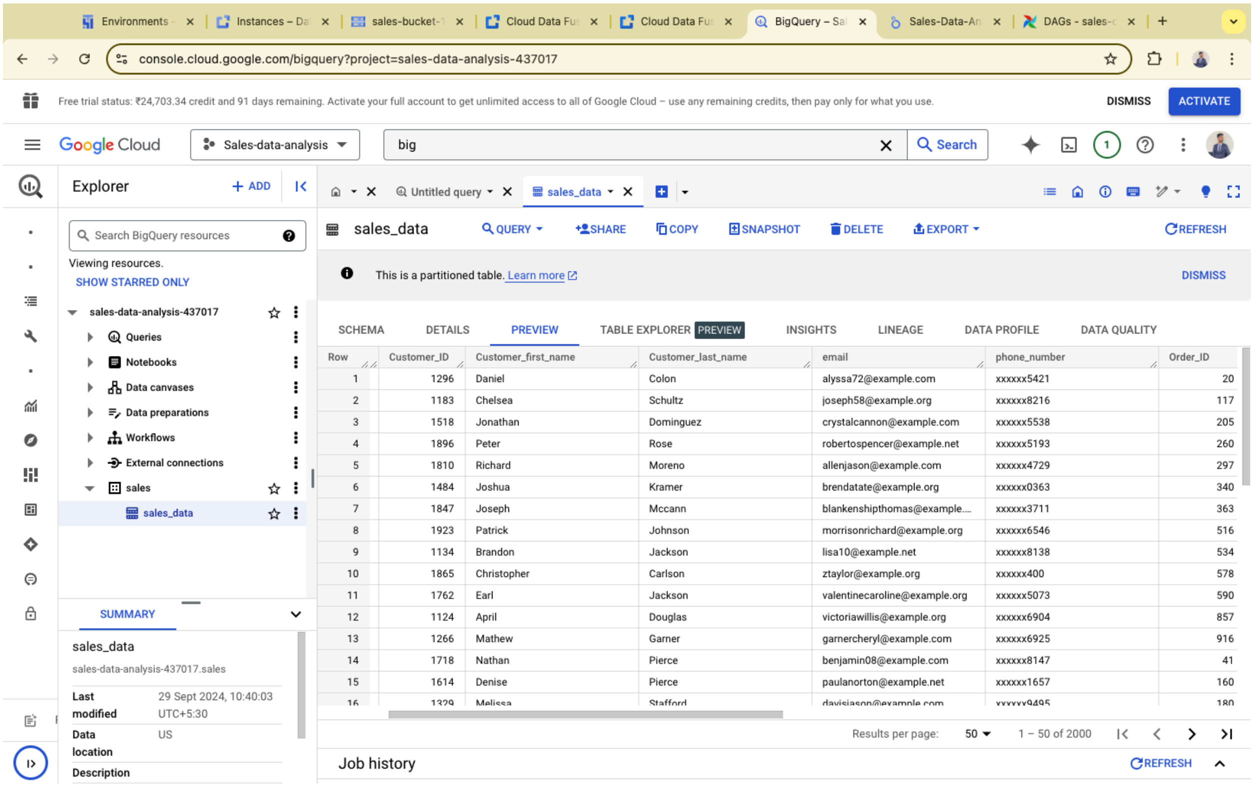
1. After adding the GCS source in **Cloud Data Fusion**, add a **Data Transformation** step.
2. Apply transformations such as:
    - **Masking the phone number** (e.g., replace with XXXX-XXXX).
    - **Hiding personal information** like email addresses or other sensitive data.
3. Use built-in **Data Preparation** transformations or write custom scripts to handle these tasks.

Step 6: Create a Data Fusion Pipeline


Cloud Data Fusion | Pipeline
sales-pipeline
Run 3 of 3
Status Succeeded
Start time 09-29-2024 10:42:32 AM
Duration 7 mins 31 secs
Warnings 21
Errors 0
Compute profile Dataproc
GCSFile 0.23.1 Out 1,000 / Errors 0
Wrangler 4.10.1 Out 1,000 / Errors 0
BigQuery 0.23.1 In 1,000 / Errors 0

1. Open **Cloud Data Fusion** and navigate to your instance.
2. Select **Studio** to create a new pipeline.
3. Choose **Batch Pipeline**.
4. Set up the **GCS Source**:
    - In the source step, point to the CSV file in the GCS bucket (gs://sales-bucket-1/sales_data.csv).
    - Ensure you configure the file format correctly (CSV with headers).

Step 7: Add a BigQuery Sink


console.cloud.google.com/bigquery?project=sales-data-analysis-437017

Free trial status: ₹24,703.34 credit and 91 days remaining. Activate your full account to get unlimited access to all of Google Cloud – use any remaining credits, then pay only for what you use.

sales_data

This is a partitioned table. Learn more

Row	Customer_ID	Customer_first_name	Customer_last_name	email	phone_number	Order_ID
1	1296	Daniel	Colon	alyssa72@example.com	xxxxxx5421	20
2	1183	Chelsea	Schultz	joseph58@example.org	xxxxxx8216	117
3	1518	Jonathan	Dominguez	crystalcannon@example.com	xxxxxx5538	205
4	1896	Peter	Rose	robertospencer@example.net	xxxxxx5193	260
5	1810	Richard	Moreno	allenjason@example.com	xxxxxx4729	297
6	1484	Joshua	Kramer	brendatate@example.org	xxxxxx0363	340
7	1847	Joseph	Mccann	blankenshipthomas@example....	xxxxxx3711	363
8	1923	Patrick	Johnson	morrisonrichard@example.org	xxxxxx6546	516
9	1134	Brandon	Jackson	lisa10@example.net	xxxxxx8138	534
10	1865	Christopher	Carlson	ztaylor@example.org	xxxxxx400	578
11	1762	Earl	Jackson	valentinecaroline@example.org	xxxxxx5073	590
12	1124	April	Douglas	victoriawillis@example.org	xxxxxx6904	857
13	1266	Mathew	Garner	garnercheryl@example.com	xxxxxx6925	916
14	1718	Nathan	Pierce	benjamin08@example.com	xxxxxx8147	41
15	1614	Denise	Pierce	paulanorton@example.net	xxxxxx1657	160
16	1329	Melissa	Stafford	davisjason@example.com	xxxxxx9495	180

Results per page: 50 1 – 50 of 2000

sales_data
sales-data-analysis-437017.sales

Last modified	29 Sept 2024, 10:40:03 UTC+5:30
Data location	US
Description	

1. In your Data Fusion pipeline, add a **BigQuery Sink** as the destination.
2. Select the **BigQuery table** where the transformed data will be stored.
3. Configure the table schema, ensuring that sensitive fields (e.g., masked phone number) are correctly mapped.

Step 8: Set BigQuery Sink Properties

1. Set the **location** (e.g., us-central1) and **dataset** for your BigQuery instance.
2. Specify the table name where the data will be stored.
3. Ensure you set the appropriate **write disposition** (e.g., append or overwrite) based on your needs.

Step 9: Go to BigQuery and Create a Dataset

1. Navigate to **BigQuery** in the Google Cloud Console.
2. Click **Create Dataset**, and provide the necessary details:
   - Dataset ID (e.g., sales_data_analysis).
   - Set the **location** to match your GCS bucket and Data Fusion instance.
3. Create the dataset.

Step 10: Complete BigQuery Sink Property Configuration

1. Return to your Data Fusion pipeline and set the dataset you created (sales_data_analysis) as the sink for your BigQuery table.
2. Review the field mappings to ensure that all transformations, especially the masked data, are set correctly.

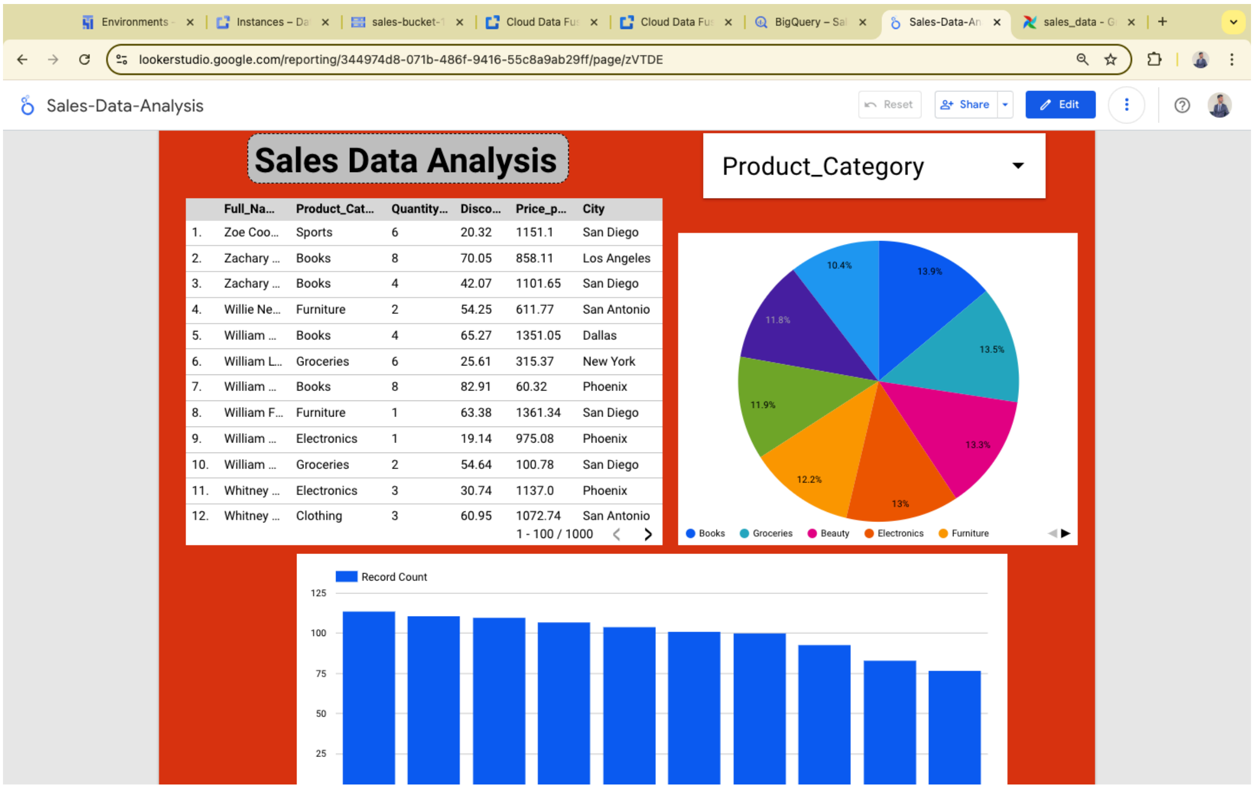
Step 11: Deploy the Data Fusion Pipeline

1. Once the pipeline is configured and all transformations are correctly set, click **Deploy** to finalize the pipeline.
2. Review the configuration for any errors or missing elements.

Step 12: Run the Data Fusion Pipeline

1. After deploying the pipeline, click **Run** to start the batch pipeline.
2. Monitor the job logs for any errors during execution.
3. Once complete, check BigQuery to verify that the transformed data has been loaded successfully.

Step 13: Visualize the Data in Looker Studio



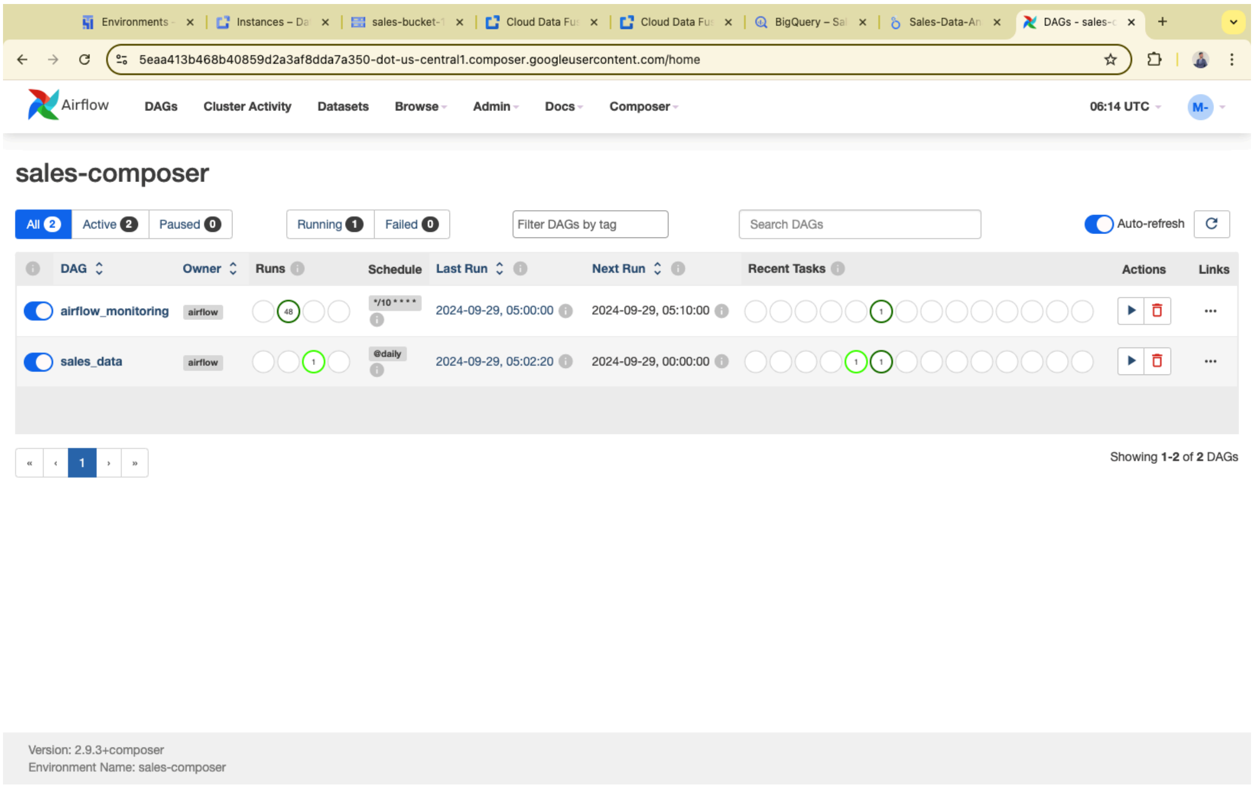
	Full_Na...	Product_Cat...	Quantity...	Disco...	Price_p...	City
1.	Zoe Coo...	Sports	6	20.32	1151.1	San Diego
2.	Zachary ...	Books	8	70.05	858.11	Los Angeles
3.	Zachary ...	Books	4	42.07	1101.65	San Diego
4.	Willie Ne...	Furniture	2	54.25	611.77	San Antonio
5.	William ...	Books	4	65.27	1351.05	Dallas
6.	William L...	Groceries	6	25.61	315.37	New York
7.	William ...	Books	8	82.91	60.32	Phoenix
8.	William F...	Furniture	1	63.38	1361.34	San Diego
9.	William ...	Electronics	1	19.14	975.08	Phoenix
10.	William ...	Groceries	2	54.64	100.78	San Diego
11.	Whitney ...	Electronics	3	30.74	1137.0	Phoenix
12.	Whitney ...	Clothing	3	60.95	1072.74	San Antonio

1 - 100 / 1000

1. Open **Looker Studio** (formerly Google Data Studio).
2. Create a new report and connect it to the BigQuery dataset (sales_data_analysis).
3. Design your dashboard by adding relevant charts and graphs to visualize the sales data.

4. Ensure sensitive data remains hidden or masked as per your transformations.

Step 14: Automate the Pipeline using Airflow (Composer)



5eaa413b468b40859d2a3af8dda7a350-dot-us-central1.composer.googleusercontent.com/home

Airflow DAGs Cluster Activity Datasets Browse Admin Docs Composer 06:14 UTC

sales-composer

All 2 | Active 2 | Paused 0 | Running 1 | Failed 0 | Filter DAGs by tag | Search DAGs | Auto-refresh

DAG	Owner	Runs	Schedule	Last Run	Next Run	Recent Tasks	Actions	Links
airflow_monitoring	airflow	48	*/10 * * * *	2024-09-29, 05:00:00	2024-09-29, 05:10:00	1		
sales_data	airflow	1	@daily	2024-09-29, 05:02:20	2024-09-29, 00:00:00	1 1		

Showing 1-2 of 2 DAGs

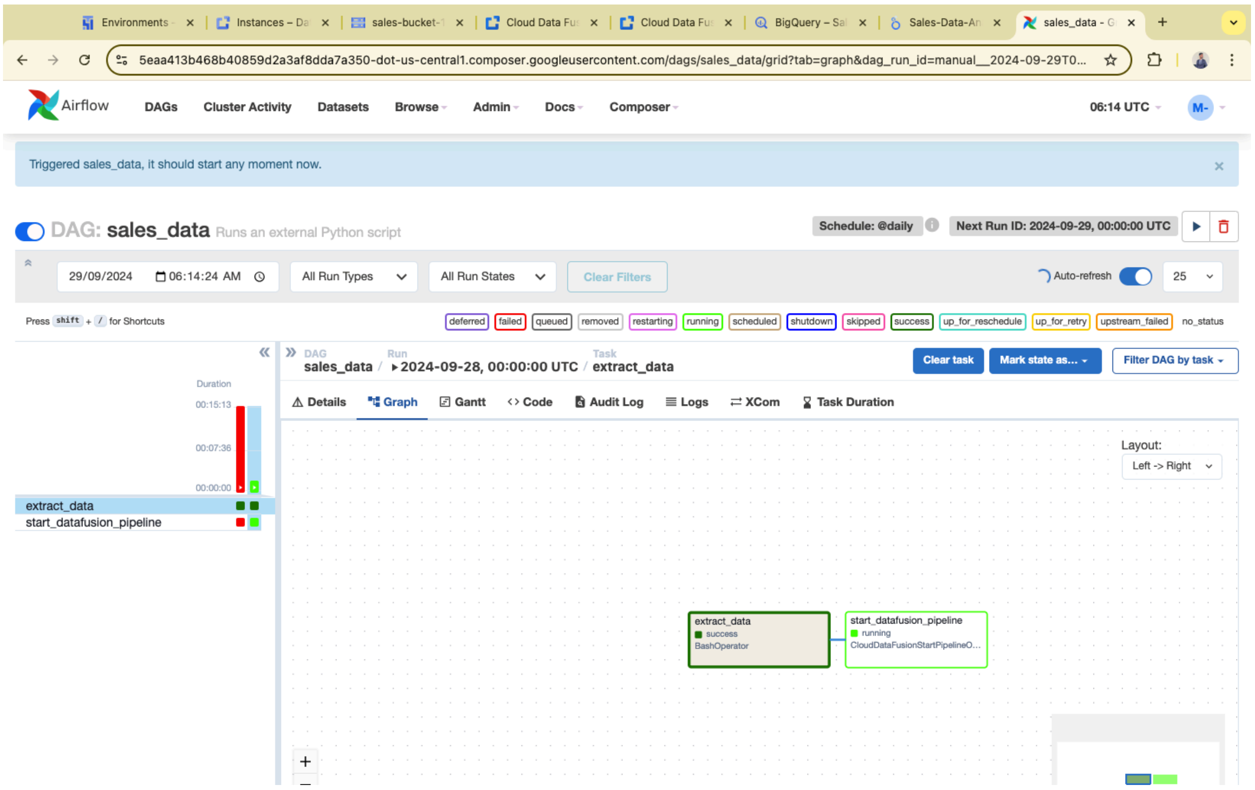
Version: 2.9.3+composer
Environment Name: sales-composer

1. Open **Cloud Composer** and navigate to your environment.
2. Create a **scripts** folder within your Composer environment by uploading your extract.py file.
3. Upload this file into /home/airflow/gcs/dags/scripts/extract.py.

Step 15: Write the Airflow DAG

1. In the same **Cloud Composer** environment, write a DAG (dag.py) that orchestrates the pipeline:
   - Run the Python script (extract.py).
   - Trigger the Data Fusion pipeline (as described in the earlier Airflow code).
2. Save this DAG inside the **DAGs folder** in your Composer environment.

Step 16: Trigger the Airflow DAG



5eaa413b468b40859d2a3af8dda7a350-dot-us-central1.composer.googleusercontent.com/dags/sales_data/grid?tab=graph&dag_run_id=manual__2024-09-29T0...

Triggered sales_data, it should start any moment now.

DAG: sales_data Runs an external Python script

Schedule: @daily   Next Run ID: 2024-09-29, 00:00:00 UTC

29/09/2024   06:14:24 AM   All Run Types   All Run States   25

Press shift + / for Shortcuts

DAG sales_data / Run 2024-09-28, 00:00:00 UTC / Task extract_data

1. In **Cloud Composer**, navigate to the **DAGs** tab.
2. Find the DAG (sales_data) you uploaded and trigger it manually to verify that everything works as expected.
3. The DAG will execute daily according to the schedule you set.

For comparison we set up a local ETL Pipeline Using Apache Spark(Non-Cloud)

1. **Installation of Apache Spark**
   - We downloaded the latest stable release of Apache Spark from the official Apache Spark website(https://spark.apache.org/downloads.html).
   - Once downloaded, extract the tar file to a preferred directory. By using the code in the terminal:

     tar -xzf spark-3.5.3-bin-hadoop3.tgz\n
   - Set up Environment Variables:

     export SPARK_HOME=/usr/local/spark
     export PATH=$SPARK_HOME/bin:$PATH

- Spark requires Java to run. Since I didn’t have Java installed on my system, i downloaded it using:

  brew install openjdk@11
  nano ~/.zshrc
  export JAVA_HOME=$(/usr/libexec/java_home -v 11)
  source ~/.zshrc

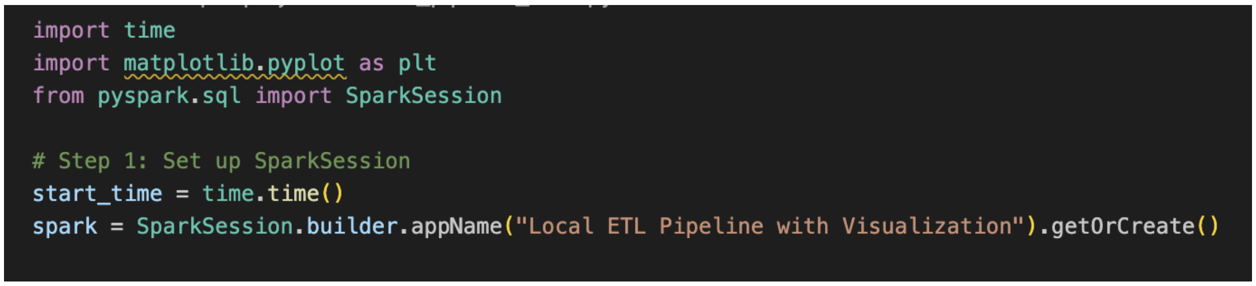
- Install PySpark:

  pip install pyspark

2. **Setting Up a Local ETL Pipeline Using Apache Spark**
After setting up Spark, we create an ETL pipeline to process the same CSV file generated for the cloud setup. The following steps outline the pipeline development process.

Explanation of the Code

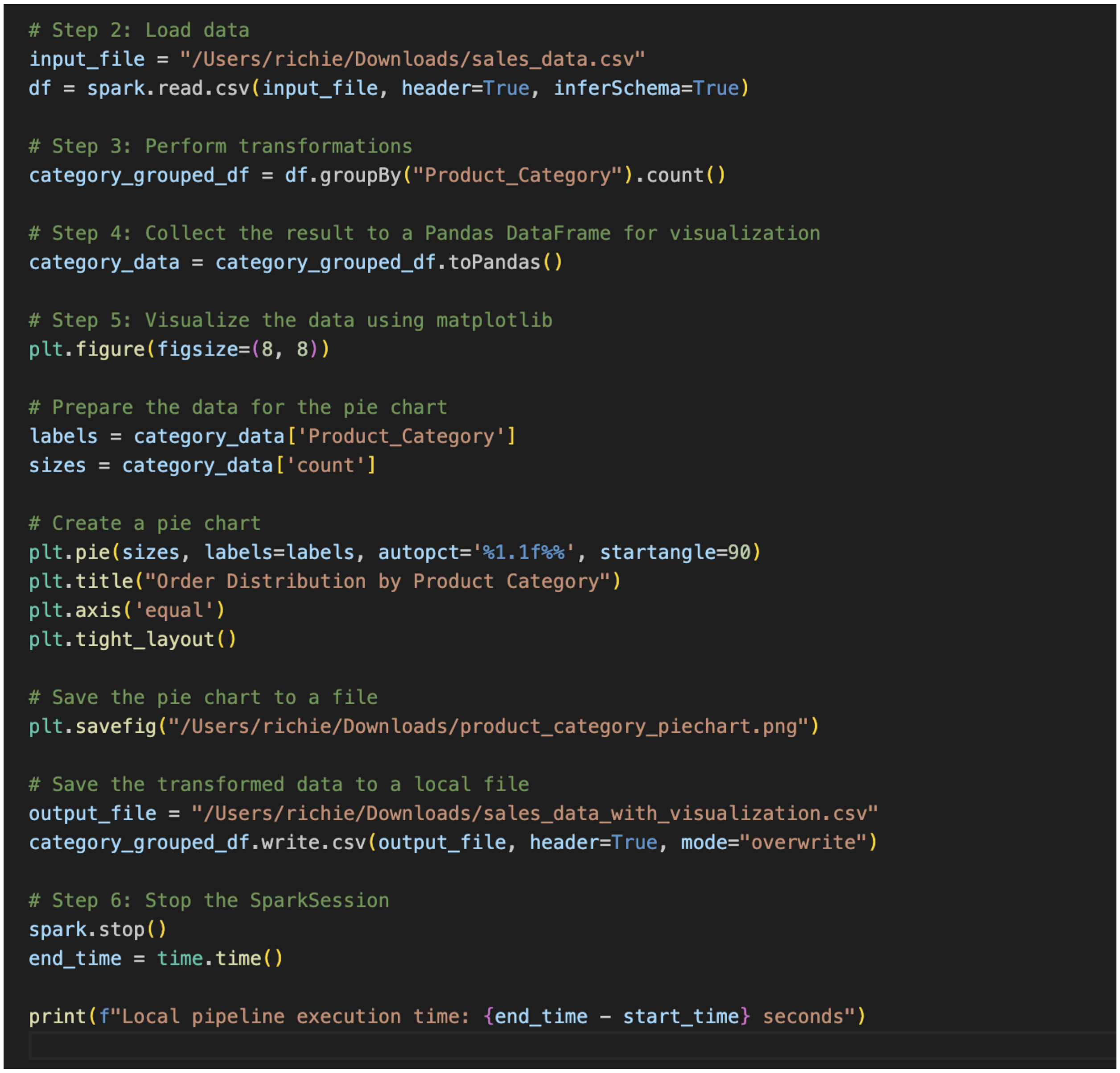
- **Spark Session Setup**: The `SparkSession.builder` initializes a Spark session required for data processing.



```
import time
import matplotlib.pyplot as plt
from pyspark.sql import SparkSession

# Step 1: Set up SparkSession
start_time = time.time()
spark = SparkSession.builder.appName("Local ETL Pipeline with Visualization").getOrCreate()
```

- **Data Loading**: The CSV data is loaded into a Spark DataFrame using `spark.read.csv`, with headers inferred and schema automatically detected.



```python
# Step 2: Load data
input_file = "/Users/richie/Downloads/sales_data.csv"
df = spark.read.csv(input_file, header=True, inferSchema=True)

# Step 3: Perform transformations
category_grouped_df = df.groupBy("Product_Category").count()

# Step 4: Collect the result to a Pandas DataFrame for visualization
category_data = category_grouped_df.toPandas()

# Step 5: Visualize the data using matplotlib
plt.figure(figsize=(8, 8))

# Prepare the data for the pie chart
labels = category_data['Product_Category']
sizes = category_data['count']

# Create a pie chart
plt.pie(sizes, labels=labels, autopct='%1.1f%%', startangle=90)
plt.title("Order Distribution by Product Category")
plt.axis('equal')
plt.tight_layout()

# Save the pie chart to a file
plt.savefig("/Users/richie/Downloads/product_category_piechart.png")

# Save the transformed data to a local file
output_file = "/Users/richie/Downloads/sales_data_with_visualization.csv"
category_grouped_df.write.csv(output_file, header=True, mode="overwrite")

# Step 6: Stop the SparkSession
spark.stop()
end_time = time.time()

print(f"Local pipeline execution time: {end_time - start_time} seconds")
```

- **Data Transformation**: We group the data by the "Product_Category" column using `groupBy` and count the number of orders in each category.
- **Data Collection for Visualization**: Spark DataFrames cannot be directly used with libraries like Matplotlib. Therefore, we convert the Spark DataFrame to a Pandas DataFrame using `toPandas()`.
- **Visualization**: A pie chart is generated using `matplotlib.pyplot.pie` to visualize the distribution of orders across product categories. The chart is saved as a PNG file.
- **Saving Transformed Data**: The transformed data is written to a new CSV file using the `write.csv` method.

Running the ETL Pipeline

To run this Spark ETL pipeline:

1. Save the script in a `.py` file.
2. Open the terminal and navigate to the directory where the file is saved.
3. Run the script using the following command:

```
spark-submit etl_pipeline.py
```

This executes the ETL pipeline, performs the transformations, and generates the visualization.