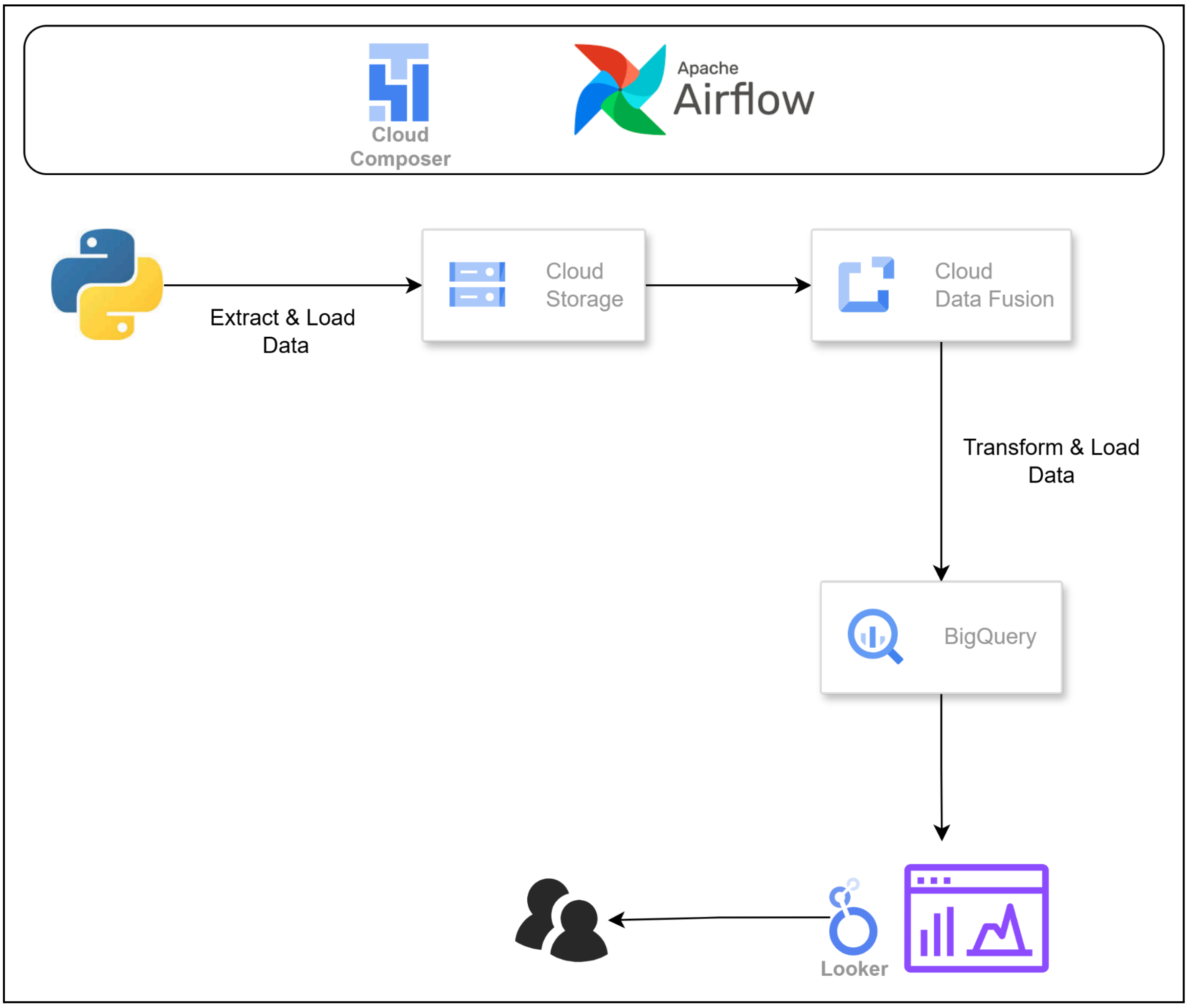
End-to-End ETL Pipeline + Automation + Analytics



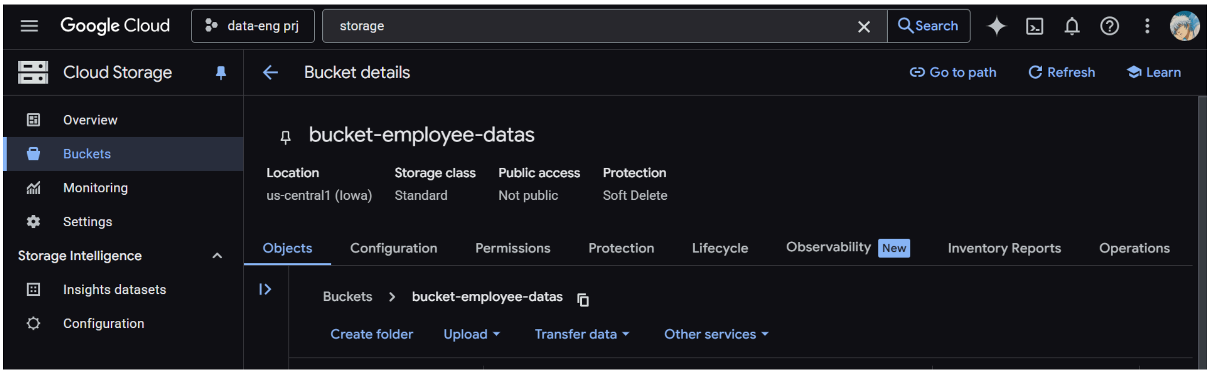
Stage 1 — Data Generation & Upload to GCS

7.1 Description

A Python script uses the `Faker` library to create synthetic employee records (e.g., id, name, email, phone, DOB, ssn, address, department, salary). Generated data is written as a CSV and uploaded to a Google Cloud Storage bucket named `bucket-employee-datas`.

7.2 Steps

1. Create a GCS bucket (Console or `gsutil mgs://bucket-employee-datas`).

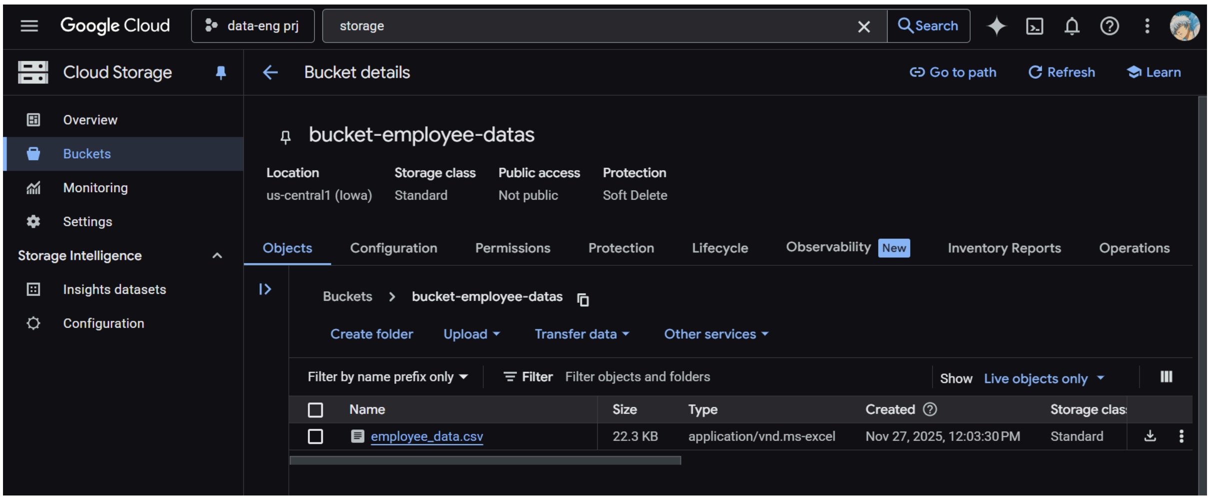


[Snapshot: Created bucket (bucket-employee-datas)]

2. Create Python script (see Appendix) that:

   - Generates `NUM_EMPLOYEES` records (e.g., 150).
   - Saves as `employee_data.csv`.
   - Uploads CSV to GCS using `google.cloud.storage`.

3. Run script locally or from a VM. Confirm CSV uploaded to `gs://bucket-employee-datas/employee_data.csv`.



[Snapshot: CSV in GCS bucket]

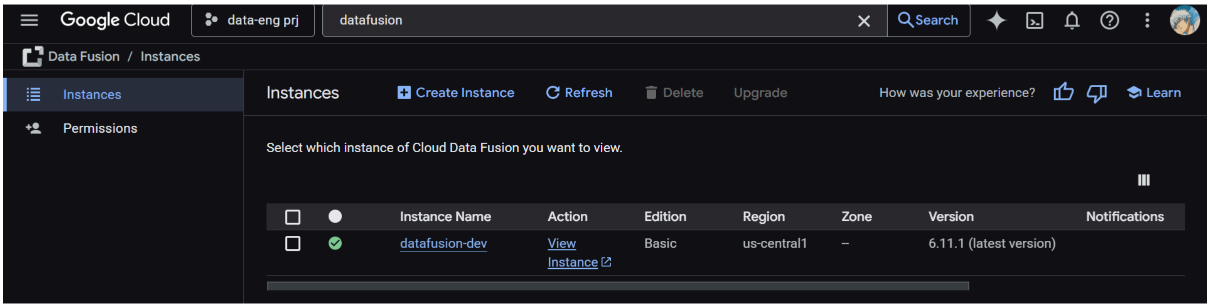
Stage 2 — Cloud Data Fusion: Wrangler & Transformations

8.1 Description

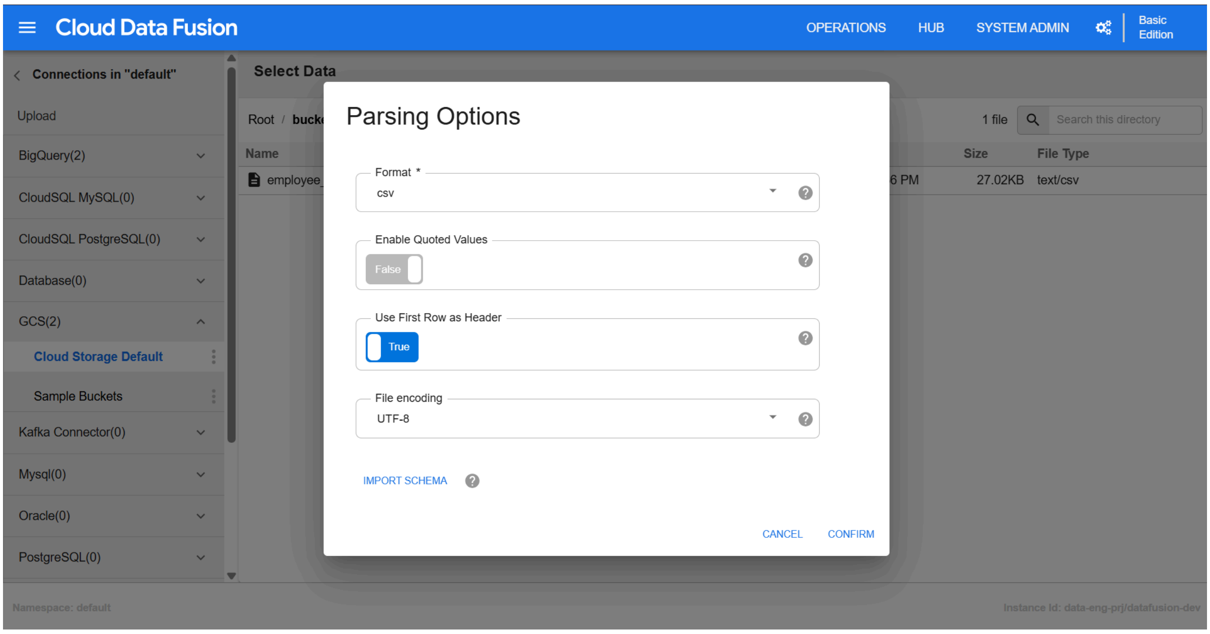
Use Cloud Data Fusion Wrangler to import the CSV, run data cleaning and transformations, then build a Batch pipeline to sync the transformed data to BigQuery.

8.2 Steps (Wrangler)

1. Open your Data Fusion instance → **View Instance** → **Wrangler**.



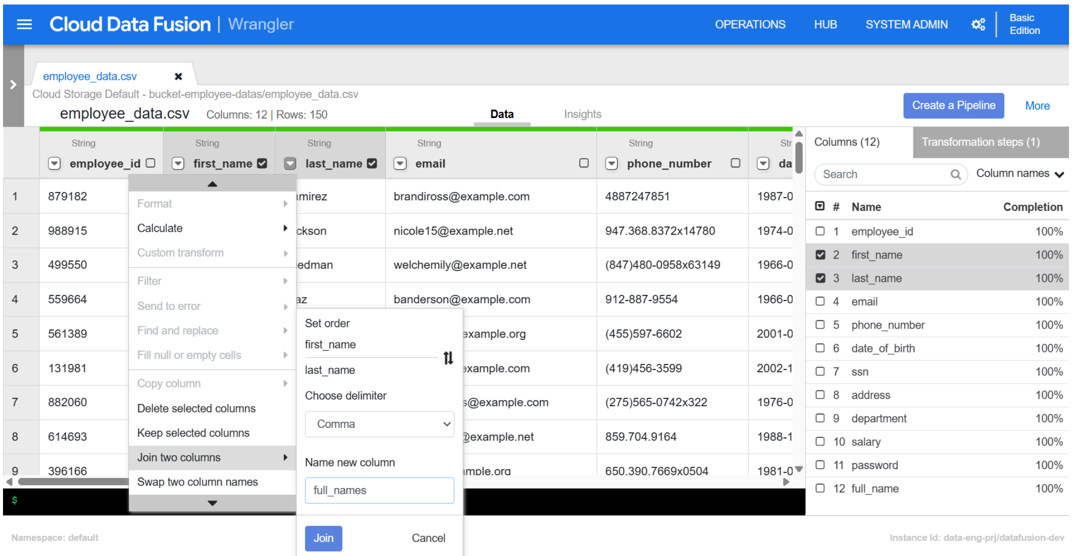
2. Choose the CSV file from Cloud Storage as source.

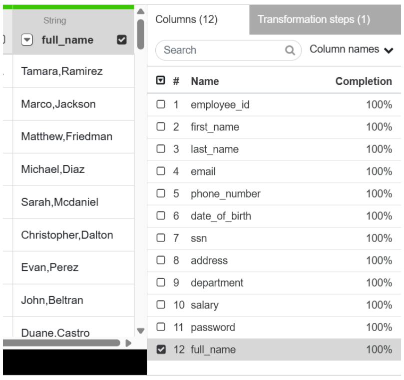


[Snapshot: Wrangler source selection]

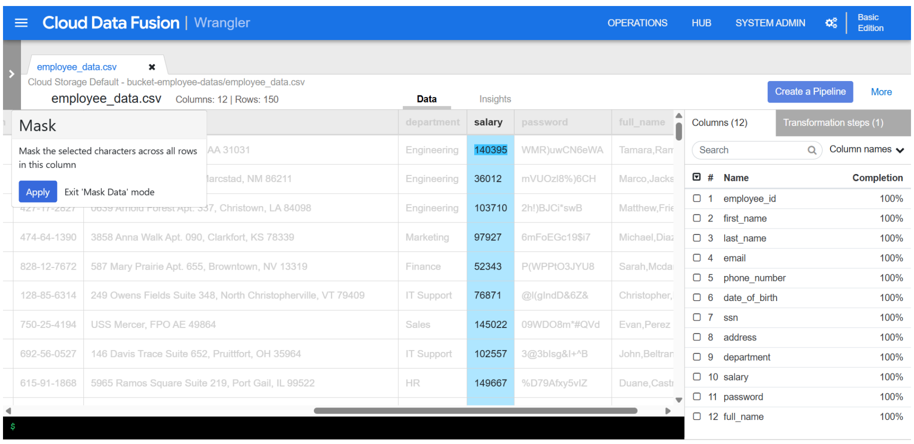
3. Transformation operations performed:

- **Join first and last name** to create `full_name` (select both columns → Join).



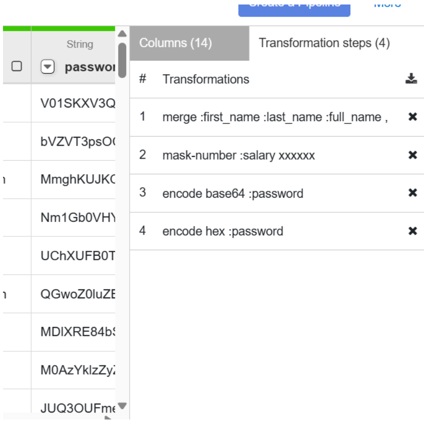


- **Mask salary**: select salary column → mask → choose custom mask → select characters to mask → Apply.



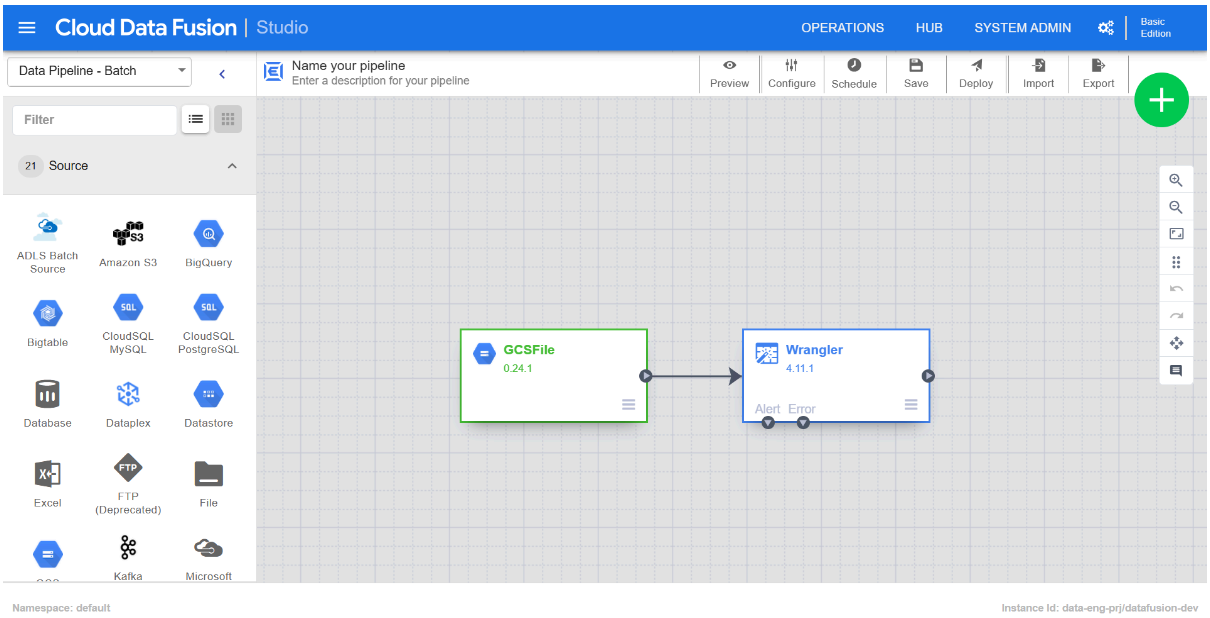
- **Password encoding** (applied similarly as masking/encoding).

- ○ Use the **transform step** side-click to inspect and undo transformations if necessary.

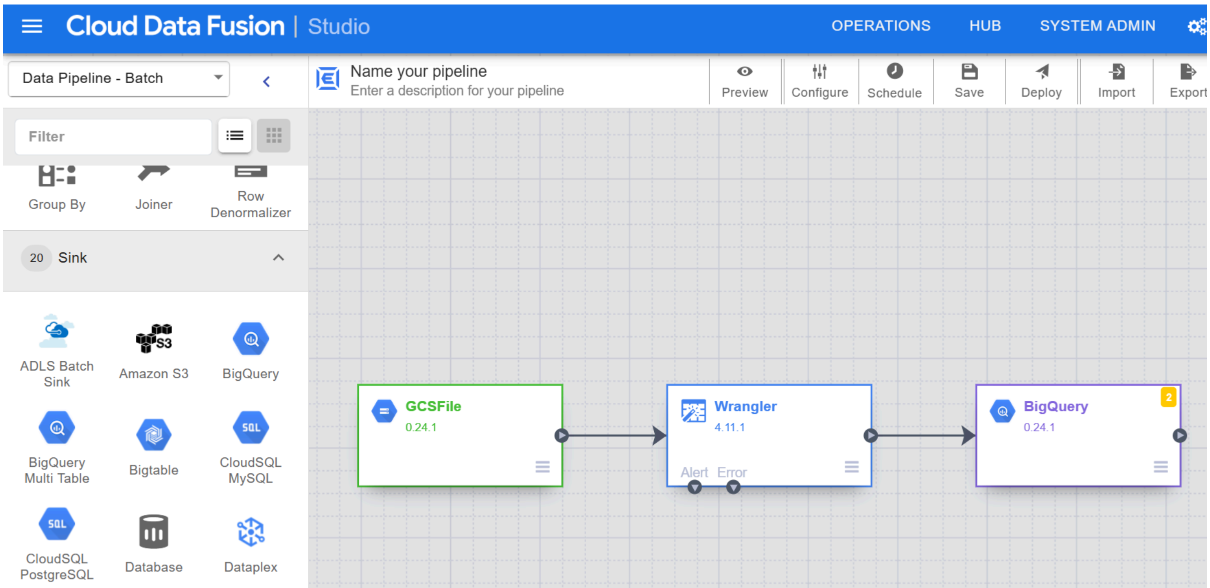


8.3 Build & Deploy Pipeline

1. Click **Create Pipeline** → *Batch Pipeline*.



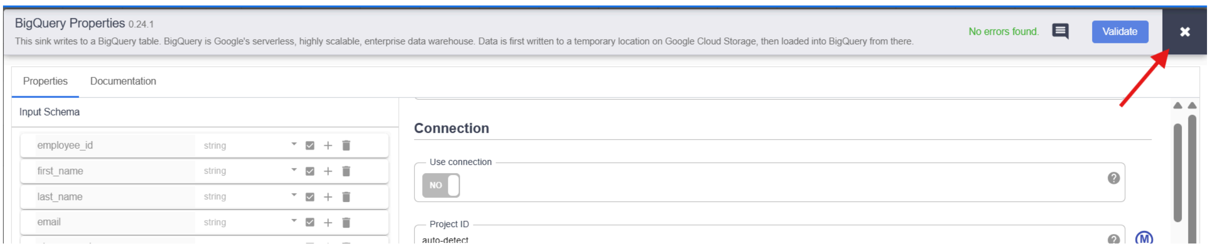
2. Add BigQuery sink (under Sync → BigQuery).



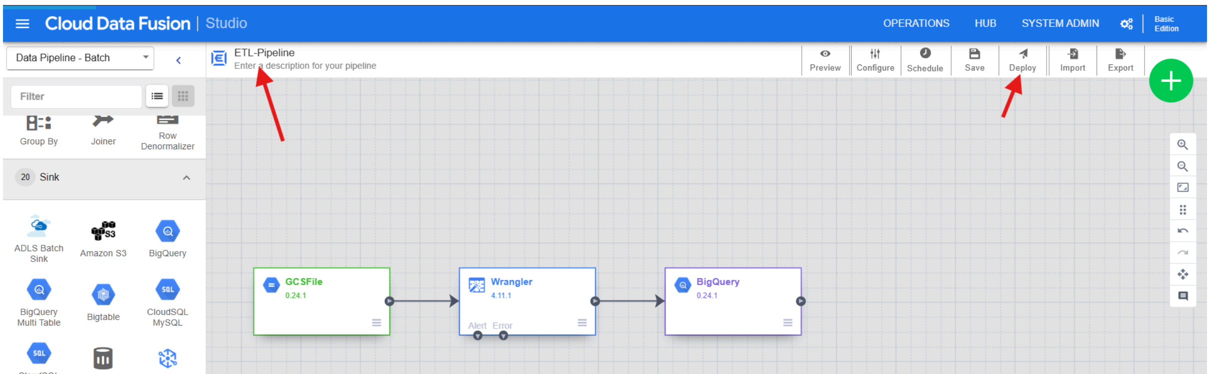
3. Configure BigQuery sink properties:

    - Dataset Project ID: `data-eng-prj`
    - Reference Name: `bq-load`
    - Dataset: `employee`
    - Table: `emp_data`

4. Validate pipeline (top → Validate). Ensure green checks (no errors).



5. Deploy the pipeline (provide deployment name → Deploy).



6. Monitor logs (Advanced logs) until pipeline succeeds.

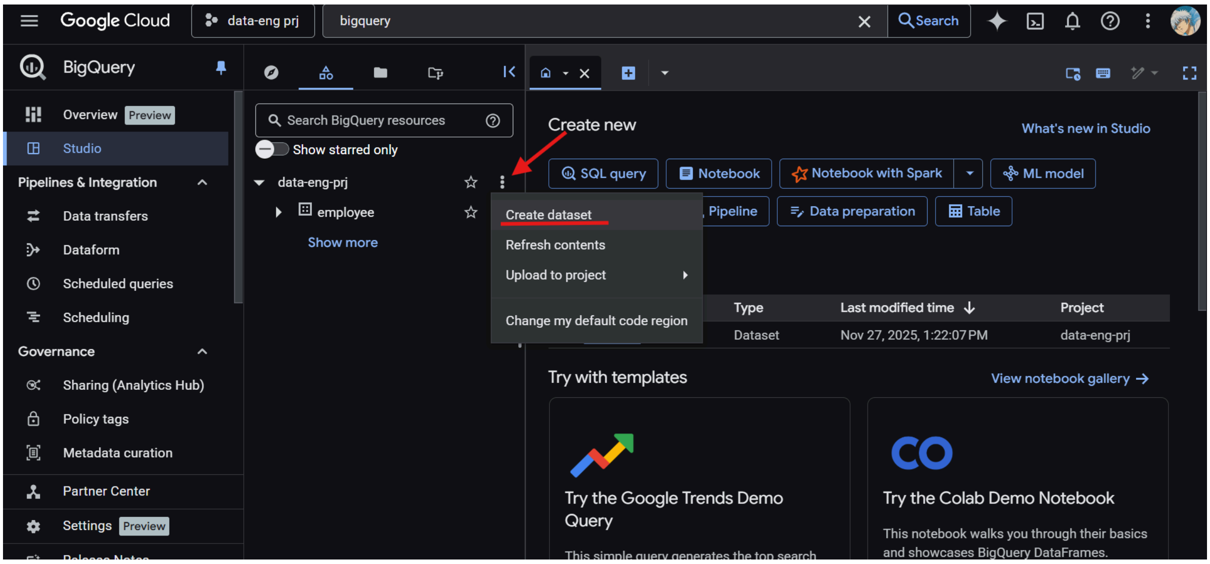
Stage 3 — BigQuery Setup & Data Load

9.1 Description

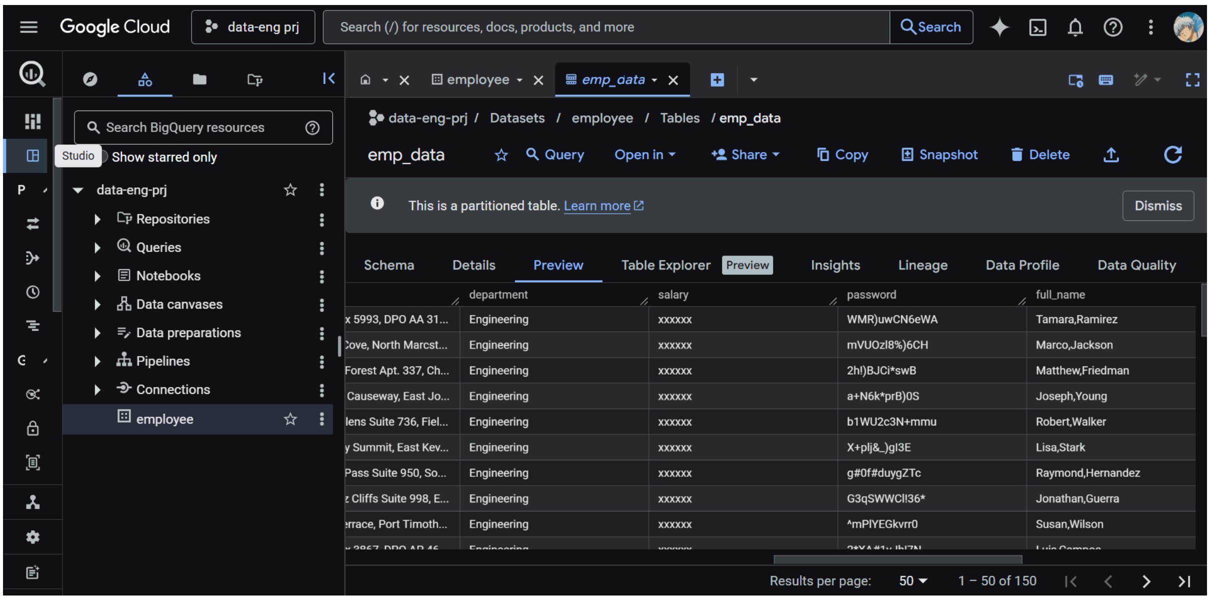
Configure dataset and table in BigQuery then verify the ETL load.

9.2 Steps

1. In BigQuery console, create dataset `employee` if not existing.



2. Confirm table `emp_data` is created by Data Fusion pipeline or create schema manually.
3. After pipeline run, verify `emp_data` contains the transformed and masked records.



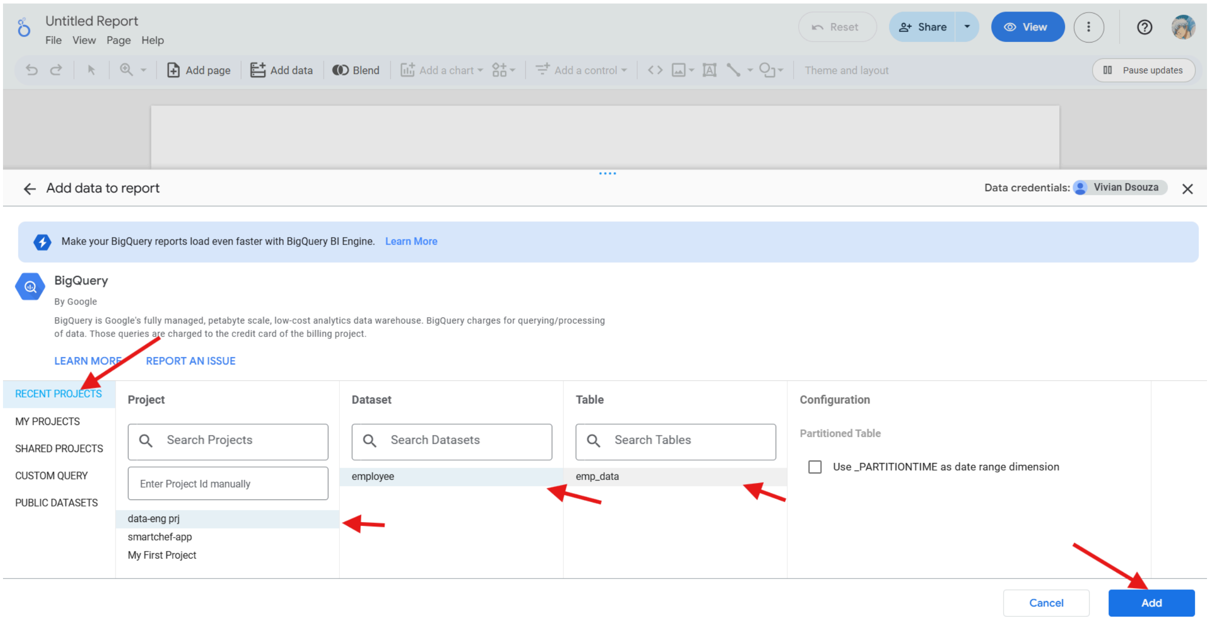
Stage 4 — Visualization (Looker Studio)

10.1 Description

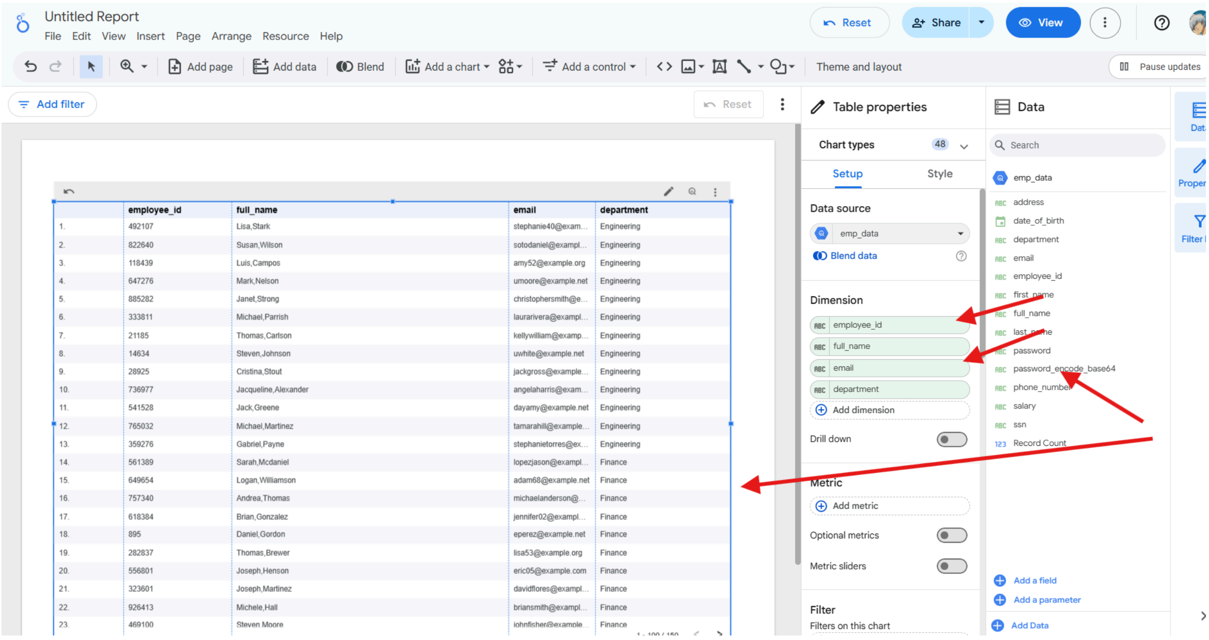
Connect BigQuery dataset to Looker Studio and create a dashboard to visualize employee data.

10.2 Steps

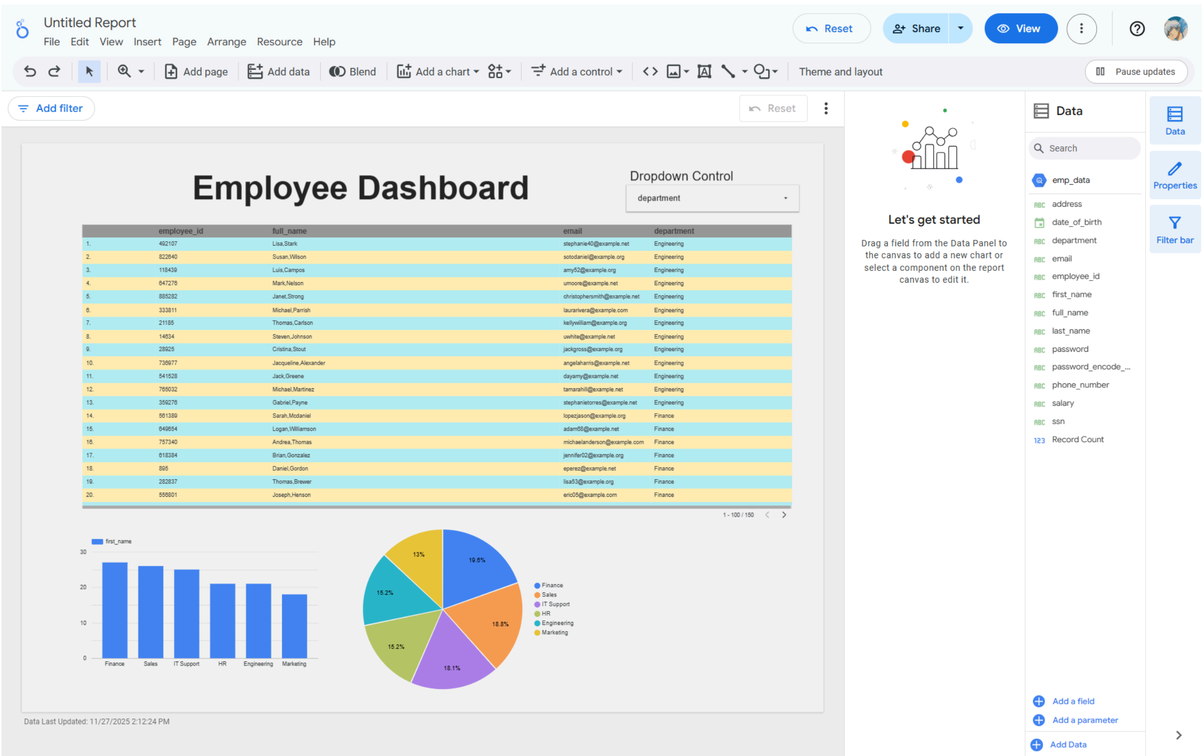
1. Open Looker Studio (Data Studio) → Create → Blank Report.
2. Add Data → BigQuery connector → Select recent project `data-eng-prj`.
3. Choose dataset `employee` → table `emp_data`.



4. Build visualization: tables, charts, and KPIs (e.g., headcount by department, salary distribution—note masked values).



5. Arrange dashboard visuals and add titles, filters, and date controls as needed.



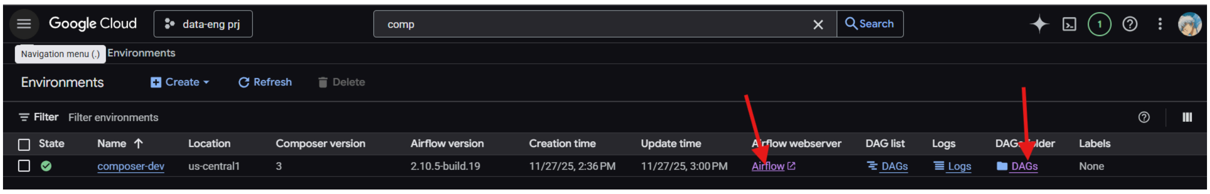
Stage 5 — Automation with Cloud Composer (Airflow)

11.1 Description

Automate the ETL pipeline by creating a Cloud Composer environment and an Airflow DAG that: runs the extraction script and triggers the Cloud Data Fusion pipeline.

11.2 Composer Setup & IAM Fix

1. Create Cloud Composer environment.



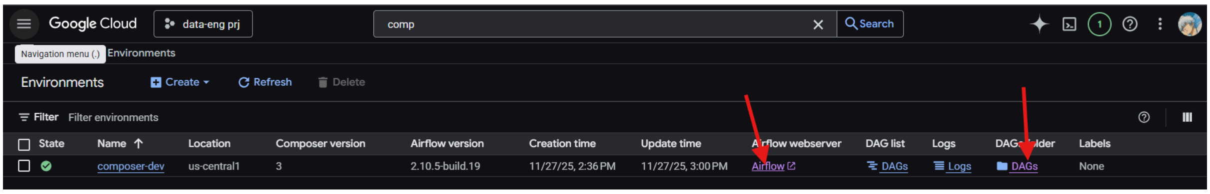
If Composer shows an error about missing IAM roles (example message):

```
The issue may be caused by missing IAM roles in the following Service Accounts:service-1047842664348@cloudcomposer-accounts.iam.gserviceaccount.com is missing role roles/composer.ServiceAgentV2Ext
```

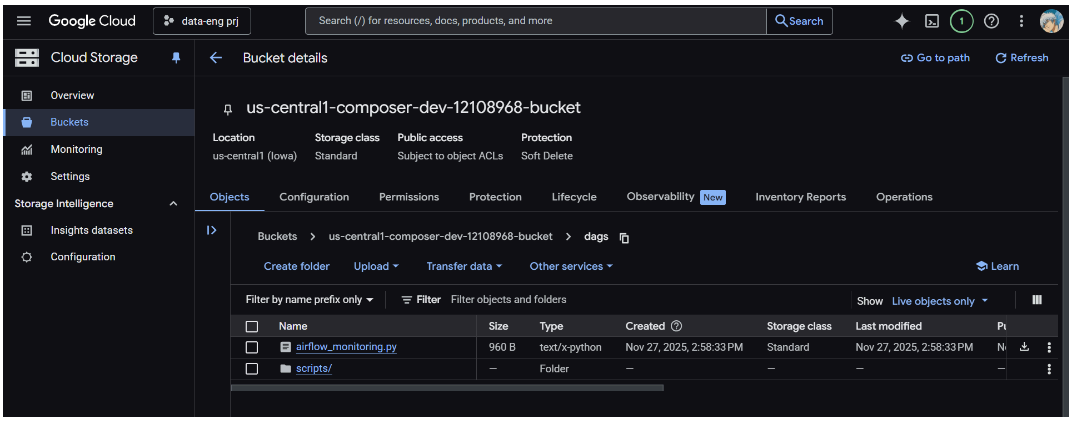
2. **Fix:** Go to IAM & Admin → locate the service account → Edit principal → Add role **Cloud Composer Service Agent v2 Extensions** → Save.

11.3 Prepare DAG & Scripts

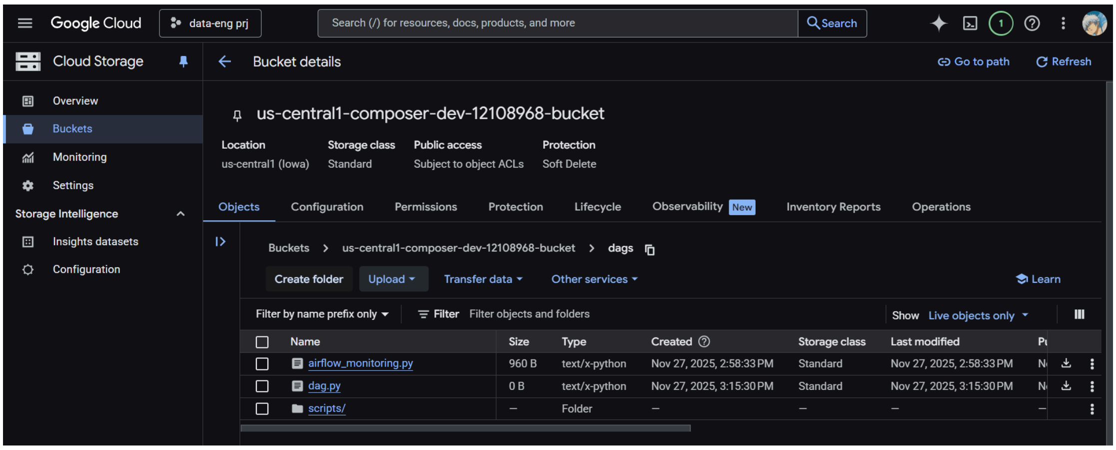
1. In Composer → navigate to the DAGs bucket.



2. Create `scripts/` folder and upload `extract.py` (the extractor script used earlier).

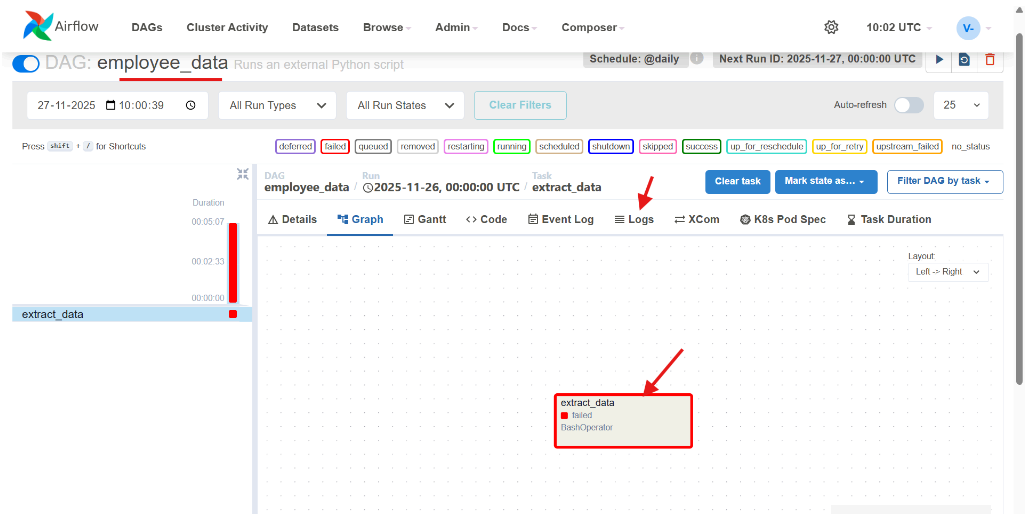


3. Upload `dag.py` (see Appendix for code). The DAG does:



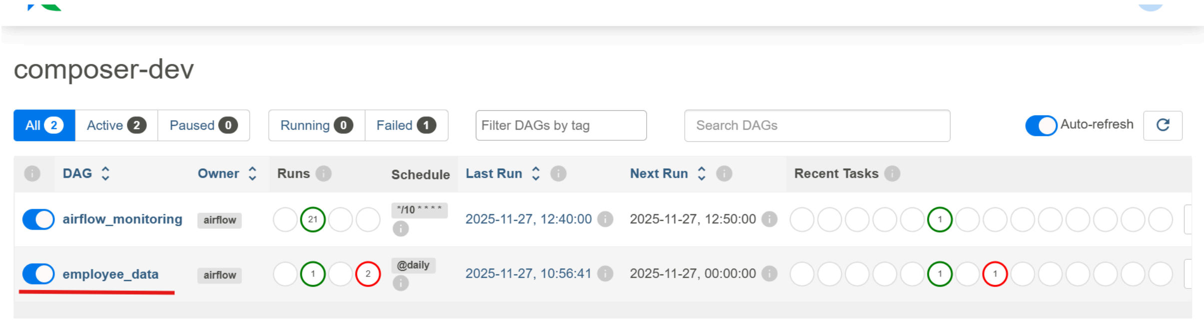
- Task 1: BashOperator → run `extract.py`.
- Task 2: CloudDataFusionStartPipelineOperator → start Data Fusion pipeline (e.g., `etl-pipeline`).
- Set DAG schedule to `@daily` and catchup = False.

4. After upload, the DAG should appear in the Airflow UI.

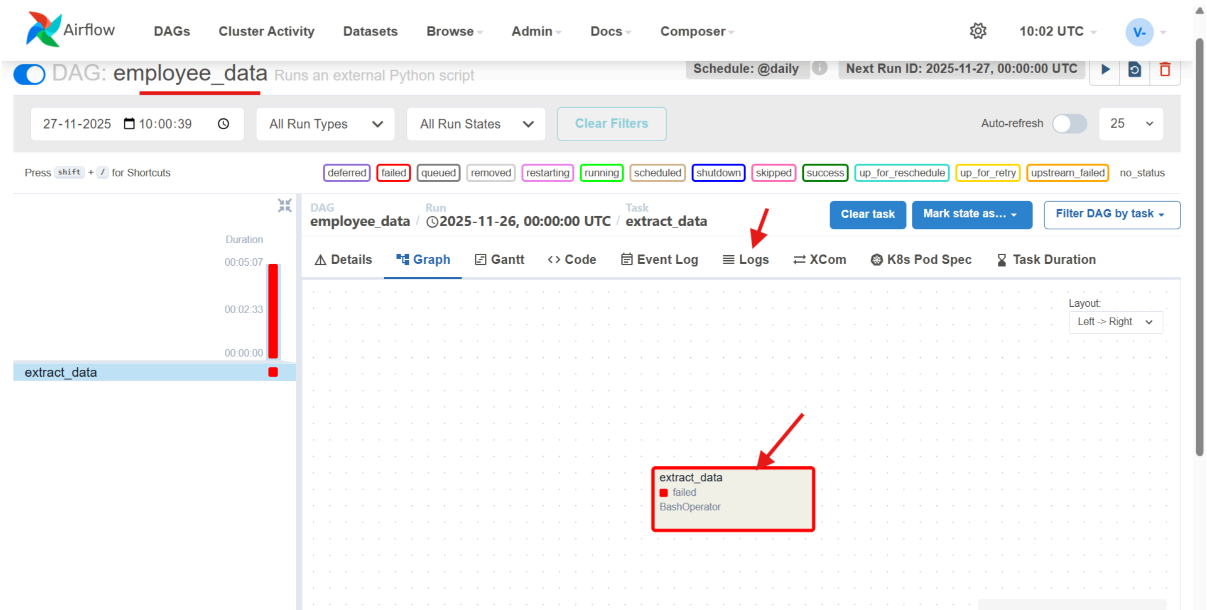


11.4 Running & Observing DAG

1. Trigger the DAG manually or wait for scheduled run.



2. If a task fails (yellow/red), click into the task → **View logs** to inspect stack trace and error.



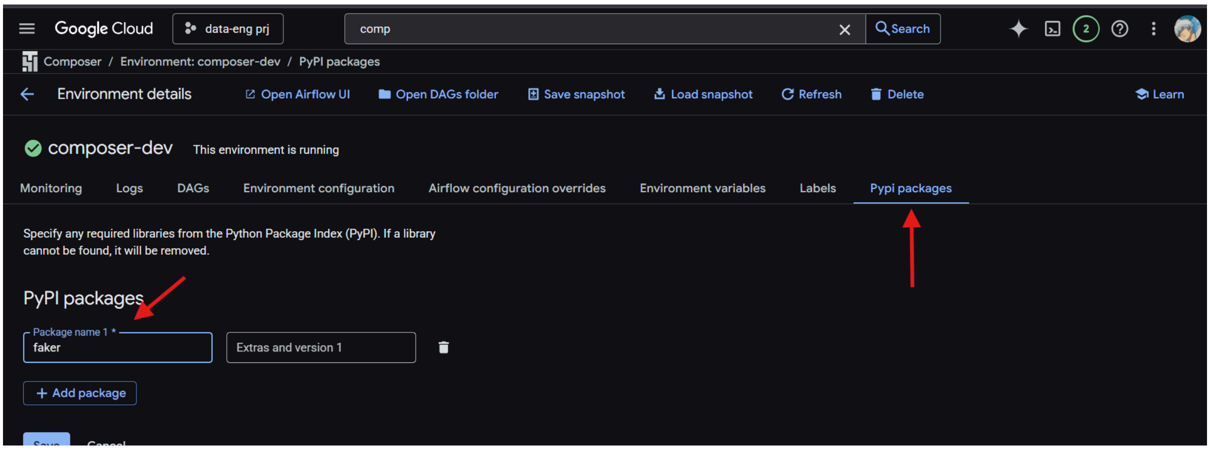
Composer Error Handling & Fixes

During the execution of the Airflow DAG inside Cloud Composer, one of the tasks failed. To identify the root cause of the failure, we first opened the Airflow UI and navigated to the task logs:

Based on the error message in the logs, the issue was related to missing Python dependencies required by the script. To fix this:

Fix 1 — Install Missing Python Packages in Cloud Composer

1. Go to **Cloud Composer** in the GCP Console.
2. Open your Composer **instance**.
3. Navigate to the **PyPI Packages** tab.
4. Click **Add Package** and install the missing package(s).



5. Save and wait for the environment to update.
   (Place snapshot here)

Fix 2 — Clear Incorrect Files from Composer DAG Bucket

While Composer updates, we also clean up an issue in the DAGs bucket:

1. Go to **Cloud Storage** → open the DAGs bucket for the Composer environment.
2. Navigate to:
   Bucket → bucket-employee-datas
3. Delete **only the incorrectly created table file** inside this folder.
4. Do *not* delete the entire `bucket-employee-datas` bucket—only the problematic file.

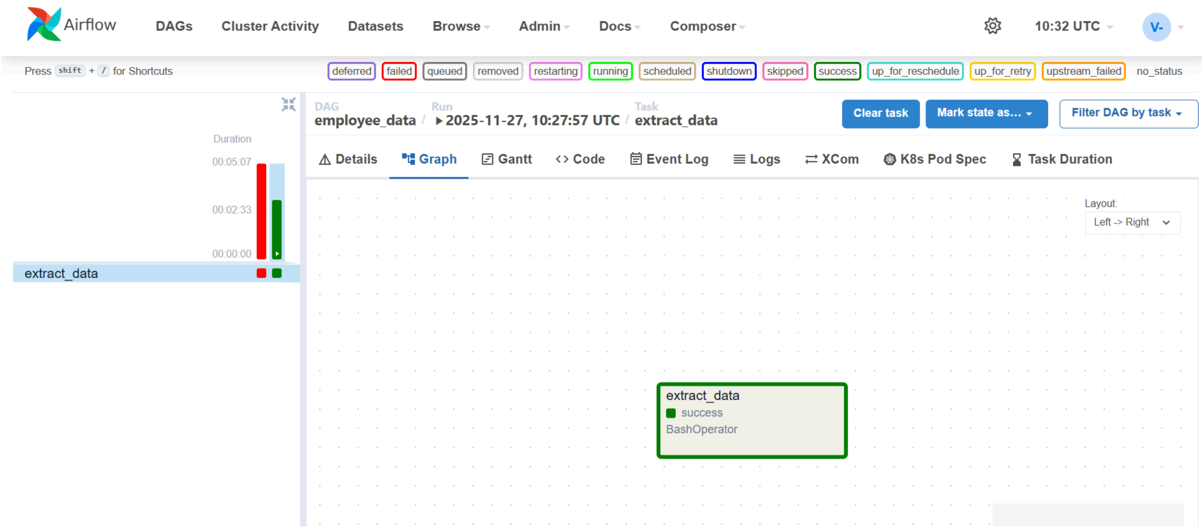
Re-running the DAG

- Go back to Airflow.

- Trigger the DAG again.

This time, the DAG runs successfully:

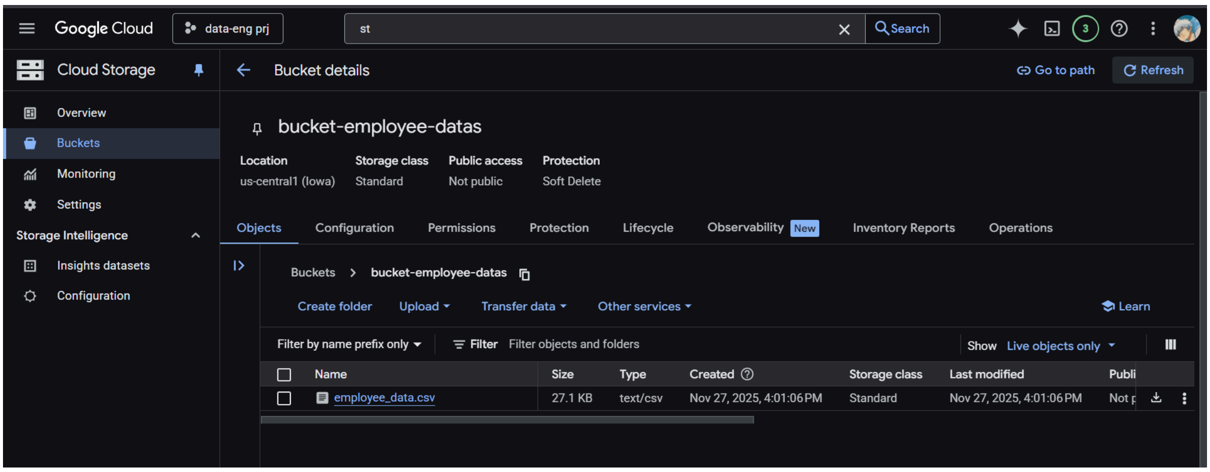
“When we rerun the Airflow DAG we get:”



Verification — Data Successfully Loaded

After the DAG executes, we confirm the new data has been successfully generated and uploaded:

“And now we see the data is loaded into the bucket.”



Starting the Data Fusion Pipeline from Airflow

To trigger the Cloud Data Fusion pipeline directly from Airflow, we reference the official Airflow documentation:

https://airflow.apache.org/docs/apache-airflow-providers-google/stable/operators/cloud/datafusion.html

We use the operator below in the DAG:

```
start_pipeline = CloudDataFusionStartPipelineOperator(
    location=LOCATION,
    pipeline_name=PIPELINE_NAME,
    instance_name=INSTANCE_NAME,
    pipeline_timeout=1000,
    task_id="start_pipeline",
)
```

This operator ensures that once the extract script completes, Airflow triggers the Data Fusion ETL pipeline automatically.

Final End-to-End Workflow Verification

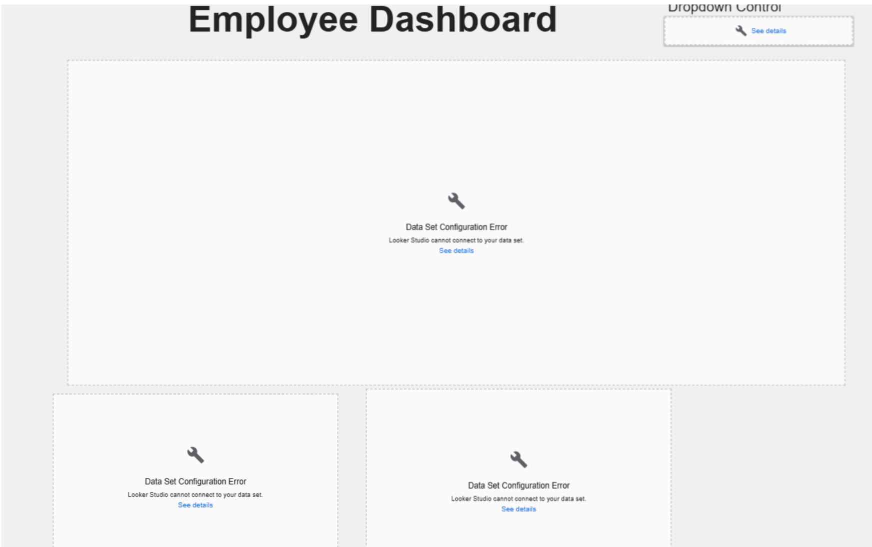
To fully validate the automated pipeline, we perform a complete end-to-end test by clearing the existing data, triggering the DAG, and verifying updates in each system: BigQuery → GCS → Data Fusion → Looker Studio.

1. Clearing the BigQuery Table

Before testing the automation flow, we delete the existing table contents in BigQuery:

- Delete the **emp_data** table (or truncate it).
- This ensures that Looker Studio will show **no data** when refreshed.

“We will then delete the emp_data from BigQuery, and when we check Looker Studio, it shows no data since BigQuery is empty.”



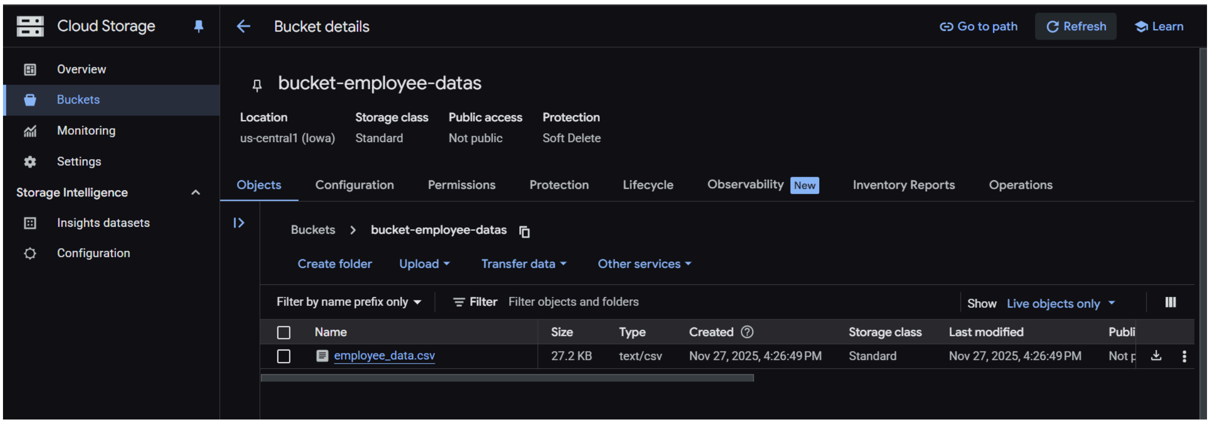
---

2. Running the Airflow DAG

Next, we trigger the Airflow DAG from Cloud Composer:

- Airflow runs the extract script.
- New synthetic employee data is generated.
- A fresh CSV file is uploaded into the GCS bucket.

“We then run the Airflow DAG, which creates the data in the bucket.”



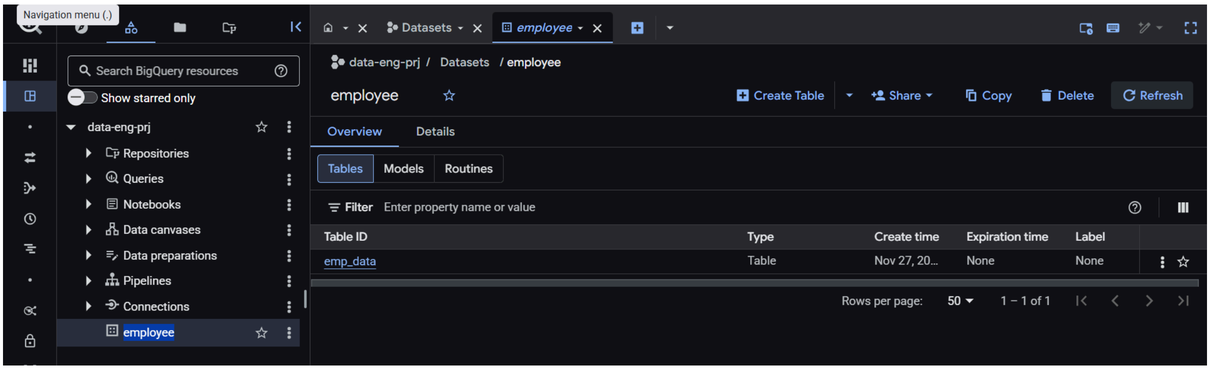
---

3. Cloud Data Fusion Pipeline Execution

After the extract script completes, Airflow triggers the Data Fusion pipeline:

- Data Fusion retrieves the CSV from the GCS bucket.
- Transformations (join, masking, encoding) are applied.
- The processed data is pushed into BigQuery.

“Then we check Data Fusion where the pipeline is triggered, and the transformed data is loaded into BigQuery.”
(Data Fusion pipeline run status = succeeded)



---

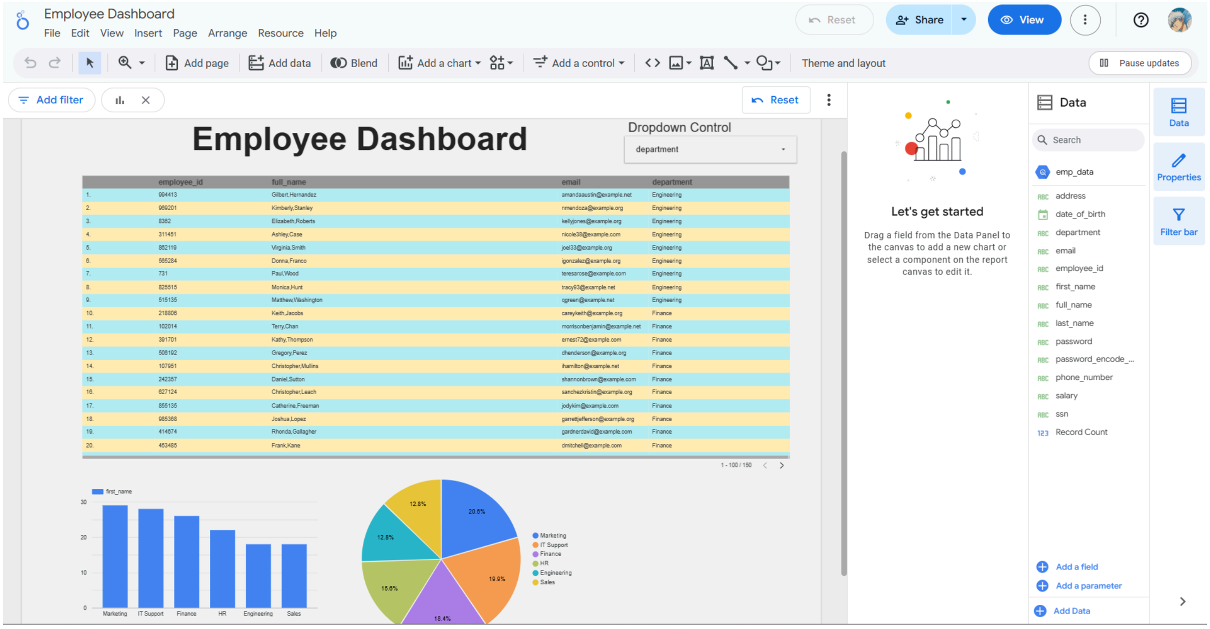
4. Refreshing Looker Studio Dashboard

Finally, we move to Looker Studio and refresh the data source:

- Looker Studio re-pulls data from BigQuery.

- Since the pipeline successfully loaded the new records, the dashboard instantly updates.

"Then we go to Looker Studio and refresh, and now we get the updated dashboard with the newly loaded data."



Code & Commands

16.1 Synthetic Data Generator & GCS Upload (`employee_generator.py`)

```
from faker import Faker
import csv
import random
from google.cloud import storage
import os
import hashlib  # Only needed if using hashed passwords option

# -----------------------------
# CONFIGURATION
# -----------------------------

NUM_EMPLOYEES = 150
OUTPUT_FILE = "employee_data.csv"
GCS_BUCKET_NAME = "bucket-employee-datas"
GCS_DESTINATION_BLOB = "employee_data.csv"

fake = Faker()

DEPARTMENTS = ["HR", "Finance", "Engineering", "Sales", "Marketing", "IT Support"]

def generate_employee():
    # Generate a strong random password
    password = fake.password(
        length=12,
        special_chars=True,
        digits=True,
        upper_case=True,
        lower_case=True
    )

    # Clean address to avoid CSV / BigQuery issues
    address = fake.address().replace('\n', ', ').replace('"', "'")

    return {
        "employee_id": fake.unique.random_number(digits=6),
        "first_name": fake.first_name(),
        "last_name": fake.last_name(),
        "email": fake.email(),
```

```
        "phone_number": fake.phone_number(),
        "date_of_birth": fake.date_of_birth(minimum_age=20,
maximum_age=65).isoformat(),
        "ssn": fake.ssn(),
        "address": address,
        "department": random.choice(DEPARTMENTS),
        "salary": random.randint(35000, 150000),
        "password": password
    }

def generate_employee_data(num_records):
    return [generate_employee() for _ in range(num_records)]

def save_to_csv(data, filename):
    with open(filename, mode="w", newline="", encoding="utf-8") as file:
        writer = csv.DictWriter(
            file,
            fieldnames=data[0].keys(),
            quoting=csv.QUOTE_ALL   # Critical fix for cleaner parsing
        )
        writer.writeheader()
        writer.writerows(data)

# -----------------------------
# UPLOAD TO GCS
# -----------------------------

def upload_to_gcs(bucket_name, source_file, destination_blob):
    client = storage.Client()
    bucket = client.get_bucket(bucket_name)
    blob = bucket.blob(destination_blob)
    blob.upload_from_filename(source_file)
    print(f"File uploaded to gs://{bucket_name}/{destination_blob}")

# -----------------------------
# MAIN EXECUTION
# -----------------------------

if __name__ == "__main__":
    employees = generate_employee_data(NUM_EMPLOYEES)
    save_to_csv(employees, OUTPUT_FILE)
    print(f"Generated {NUM_EMPLOYEES} employee records")
    print(f"CSV saved locally as: {OUTPUT_FILE}")

    upload_to_gcs(GCS_BUCKET_NAME, OUTPUT_FILE, GCS_DESTINATION_BLOB)
```

16.2 Airflow DAG (`dag.py`)

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

default_args = {
    'owner': 'airflow',
    'start_date': datetime(2023, 12, 18),
    'depends_on_past': False,
    'email': ['vishal.bulbule@techtrapture.com'],
    'email_on_failure': False,
    'email_on_retry': False,
    'retries': 1,
    'retry_delay': timedelta(minutes=5),
}

dag = DAG(
    'employee_data',
    default_args=default_args,
    description='Runs an external Python script and triggers Data Fusion
ETL',
    schedule_interval='@daily',
    catchup=False
)

with dag:
    # Task 1: Run extraction script to generate & upload CSV
    run_script_task = BashOperator(
        task_id='extract_data',
        bash_command='python /home/airflow/gcs/dags/scripts/extract.py',
    )

    # Task 2: Trigger Cloud Data Fusion pipeline
    start_pipeline = CloudDataFusionStartPipelineOperator(
        location="us-central1",
        pipeline_name="ETL-Pipeline",
        instance_name="datafusion-dev",
        task_id="start_datafusion_pipeline",
    )

    # DAG Task Order
    run_script_task >> start_pipeline
```