**Parallel Data Pipeline Ingestion with Apache Airflow, AWS, PostgreSQL, and OpenWeatherMap API**

### Project Overview

This project demonstrates how to build a simple end-to-end data pipeline using AWS, Python, and PostgreSQL. The goal is to extract store location data from a CSV in S3, enrich it with real-time weather data using the OpenWeatherMap API, store it in an RDS PostgreSQL database, and export the final result back to S3.

We are building a **parallel processing data pipeline** orchestrated with **Apache Airflow running on an AWS EC2 instance**.

The project involves:

1. Ingesting a **CSV file from an S3 bucket** and loading it into **PostgreSQL on AWS RDS**
2. Simultaneously fetching **weather data from OpenWeatherMap API**, transforming it, and loading it into the **same PostgreSQL instance**
3. Merging the two datasets and exporting the result back to **Amazon S3**
4. All tasks are orchestrated via **Apache Airflow DAGs**

### **Tools Involved:**

* **AWS S3**: Cloud storage for raw and processed files
* **Apache Airflow**: Orchestration tool to automate tasks
* **Python**: Used to write transformation and API scripts
* **AWS RDS (PostgreSQL)**: Stores transformed and joined data
* **OpenWeatherMap API**: Provides real-time weather data
* **Amazon EC2**: Runs the Airflow server
* **Libraries**: s3fs, fsspec, requests, pandas, sqlalchemy
* **Python (pandas):** Data transformation & integration
* **psycopg2 / SQLAlchemy:** PostgreSQL DB connection from Python

### **Step 1: Sign Up for OpenWeatherMap API Key**

* **Purpose**: Needed to fetch real-time weather data.
  + Go to openweathermap.org
  + Sign up and validate your email
  + Find your **API key** in your account’s profile > API Keys section
  + This key will be used in Python scripts to call the weather API

### **Step 2: Create an AWS Account**

* **Purpose**: Host cloud resources (EC2, RDS, S3, etc.)
  + Visit aws.amazon.com and sign up

### **Step 3: Provision PostgreSQL RDS Instance**

* **Purpose**: A managed database to store CSV and weather data.
  1. Navigate to RDS → Create Database
  2. Choose:
     + **Engine**: PostgreSQL
     + **Template**: Free Tier
     + **Instance Identifier**: Name it (e.g., weather-data-db)
     + **Master Username**: Leave as postgres
     + **Master Password**: Choose and note it
  3. Set other parameters:
     + **Port**: 5432 (default PostgreSQL)
     + **Public access**: Enabled
     + **VPC security group**: Create new or assign existing
     + Create a new security group (weather-db-sg)
     + Skip creating an initial database (PostgreSQL auto-creates postgres by default)
     + Enable automatic backups and encryption (optional)
  4. Click **Create database** and wait for it to finish provisioning

### **Step 4: Provision EC2 Instance (for Airflow)**

* **Purpose**: Host Apache Airflow to orchestrate the pipeline.
  1. Go to EC2 → Launch Instance
  2. Choose:
     + **Name**: airflow\_instance
     + **AMI**: Ubuntu
     + **Instance Type**: t2.medium (2 vCPU, 4GB RAM – suitable for Airflow)
     + **Key Pair**: Use existing or create new .pem file
     + **Security Group**:
       - Allow SSH (port 22)
       - Allow HTTP/HTTPS
  3. Launch the instance
  4. Update inbound rules, opening port 8080 in security groups
  5. Once it’s in **running** state, click **Connect → EC2 Instance Connect** to access the terminal or connect via Visual Studio Code

### **Step 5: Install Dependencies on EC2**

* **Purpose**: Set up the environment for Airflow and data processing.
  + **Update packages**:

sudo apt update

* + **Install pip**:

sudo apt install python3-pip

* + **Install virtual environment**:

sudo apt install python3.12-venv

* + **Create virtual environment**:

python3 -m venv airflow\_venv

* + **Activate virtual environment**:

source airflow\_venv/bin/activate

* + **Install required Python packages**:

pip install s3fs

pip install fsspec

pip install apache-airflow

pip install apache-airflow-providers-postgres

pip install apache-airflow-providers-http

pip install pandas

pandas: For data wrangling and cleaning

sqlalchemy + psycopg2-binary: To interact with PostgreSQL

s3fs, fsspec: For accessing AWS S3 files

### **Step 6: Allow EC2 to Access RDS**

* If you're connecting **from EC2 to RDS in the same VPC**, public accessibility isn’t needed — just security group rules must allow the traffic.
* **Purpose**: EC2 instance must connect to the PostgreSQL RDS.
  1. In AWS Console → EC2 → Security Groups
  2. Select the security group of the EC2 instance
  3. Click **Inbound Rules → Edit**
  4. Add a rule:
* **Type**: PostgreSQL
* **Protocol**: TCP
* **Port Range**: 5432
* **Source**:
  + Choose **Custom**
  + Enter the **Security Group ID of your EC2 instance SG** (not the IP address)

### **Step 7: Importing Data from Amazon S3 into RDS for PostgreSQL**

### Prerequisites:

* **PostgreSQL version 10.7 or higher** (RDS or Aurora PostgreSQL) [docs.aws.amazon.com+4docs.aws.amazon.com+4docs.aws.amazon.com+4](https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/USER_PostgreSQL.S3Import.html?utm_source=chatgpt.com)
* **Install PostgreSQL client:**

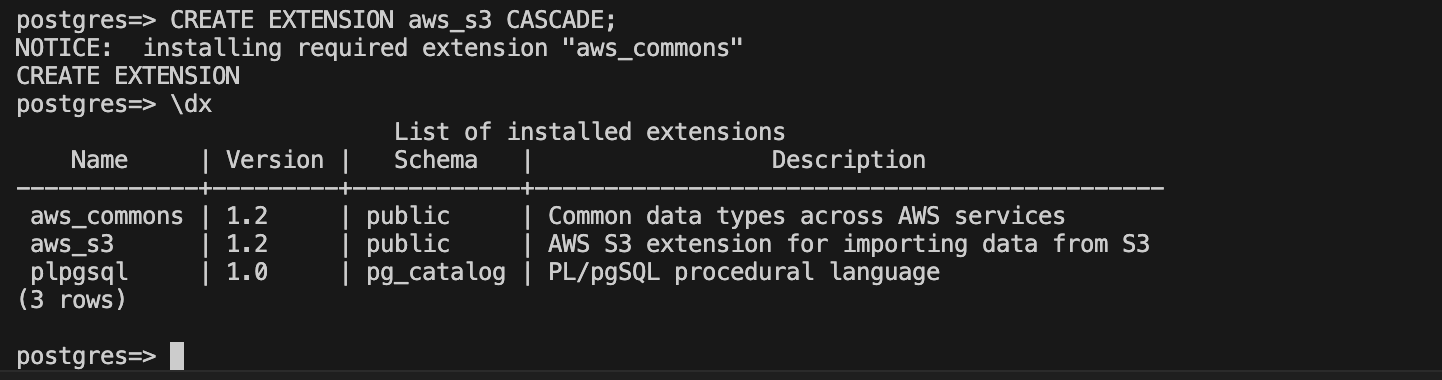
<https://www.postgresql.org/download/linux/ubuntu/>

* + - sudo apt install postgresql
* **CSV or GZIP-compatible file** stored in an S3 bucket in the same region as the RDS instance
* **Install** aws\_s3  **extension**

Before you can use Amazon S3 with your RDS for PostgreSQL DB instance, you need to install the aws\_s3 extension. This extension provides functions for importing data from an Amazon S3. It also provides functions for exporting data from an RDS for PostgreSQL DB instance to an Amazon S3 bucket

<https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/USER_PostgreSQL.S3Import.InstallExtension.html>

psql --host=weather-data-db1.cjwgyeeg8cxe.us-east-1.rds.amazonaws.com --port=5432 --username=postgres –password



* **IAM role to access an Amazon S3 bucket: Before you load data from an Amazon S3 file, give your RDS for PostgreSQL DB instance permission to access the Amazon S3 bucket the file is in. This way, you don't have to manage additional credential information**

<https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/USER_PostgreSQL.S3Import.AccessPermission.html>

* 1. Create an IAM policy

This policy provides the bucket and object permissions that allow your RDS for PostgreSQL DB instance to access Amazon S3.

Include in the policy the following required actions to allow the transfer of files from an Amazon S3 bucket to Amazon RDS:

s3:GetObject

s3:ListBucket

aws configure

aws iam create-policy \

--policy-name rds-s3-import-policy \

--policy-document '{"Version": "2012-10-17","Statement":[{"Sid": "s3import","Action": ["s3:GetObject", "s3:ListBucket"],"Effect": "Allow", "Resource": ["arn:aws:s3:::weather-raw-dt","arn:aws:s3:::weather-raw-dt/\*"]}]}'

* 1. Create an IAM role

aws iam create-role \

--role-name rds-s3-import-role \

--assume-role-policy-document '{"Version": "2012-10-17", "Statement": [{"Effect": "Allow", "Principal": {"Service": "rds.amazonaws.com"}, "Action": "sts:AssumeRole"}]}'

* 1. Attach the IAM policy that you created to the IAM role that you created.

aws iam attach-role-policy \

--policy-arn arn:aws:iam::626127091134:policy/rds-s3-import-policy \

--role-name rds-s3-import-role

* 1. Add the IAM role to the DB instance.

aws rds add-role-to-db-instance \

--db-instance-identifier weather-data-db1 \

--feature-name s3Import \

--role-arn arn:aws:iam::626127091134:role/rds-s3-import-role \

--region us-east-1

### **Step 8: Data Pipeline Design (Overview of DAG)**

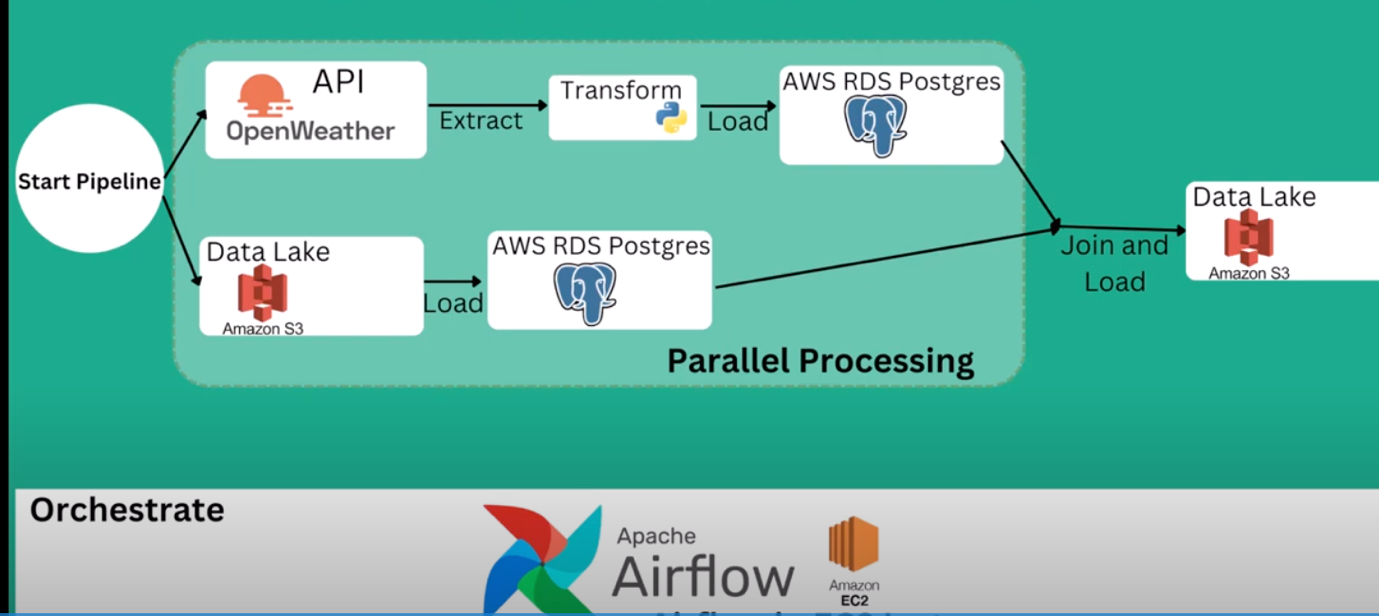
This Apache Airflow pipeline runs on an EC2 instance and automates the process of ingesting and transforming weather data from two sources — a CSV file in AWS S3 and real-time weather data from the **OpenWeatherMap API**. The final output is stored in an **AWS RDS PostgreSQL** database, and a merged export is saved back to **S3**.

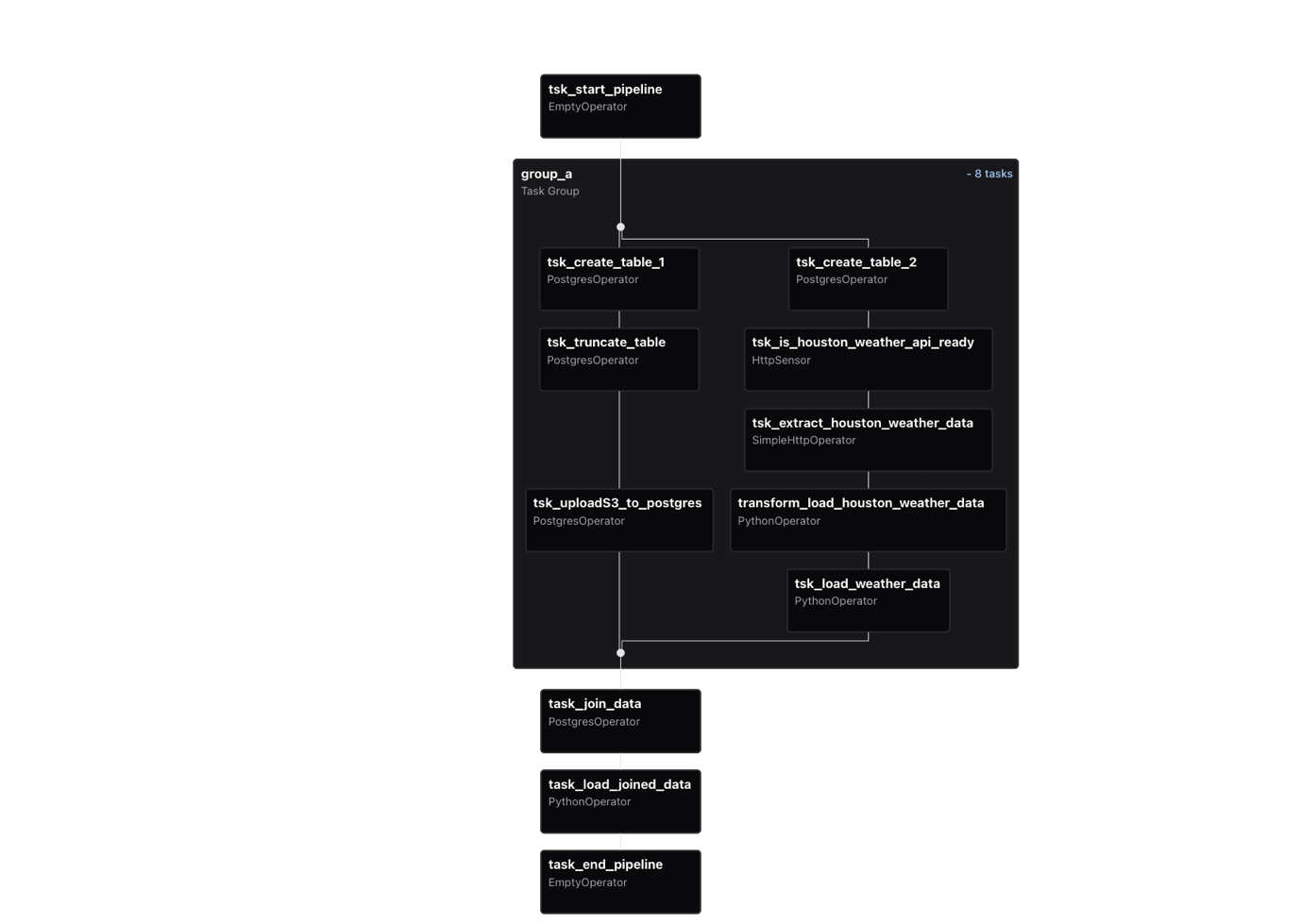
### **Overview**

This Airflow workflow orchestrates a **daily weather data pipeline** that performs several key tasks:

* Extracts weather data from an external API
* Loads supporting city data from an S3 bucket into a PostgreSQL database
* Transforms the raw weather data into a usable format
* Loads transformed weather data into PostgreSQL
* Joins the weather data with city metadata for enriched insights
* Saves the final joined data back into an S3 bucket as a CSV file

The pipeline is designed to run daily without catching up on past dates and includes retry policies and error handling.

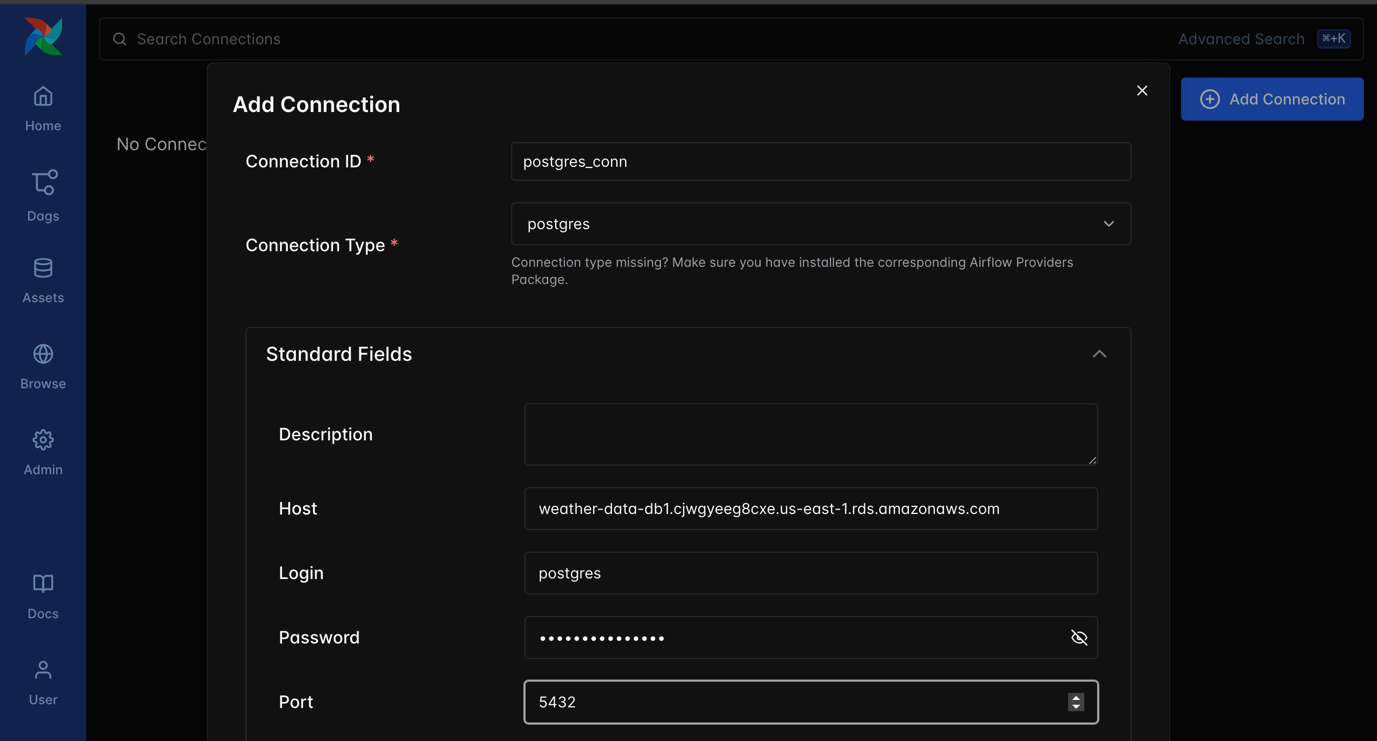




Airflow will run a **DAG (Directed Acyclic Graph)** with the following tasks:

1. load\_csv\_from\_s3: Read the CSV file from S3 into a DataFrame
2. fetch\_weather\_data: Pull weather info via API
3. transform\_and\_merge: Merge datasets on a common column (e.g., city)
4. load\_to\_postgres: Write the final table to RDS PostgreSQL
5. export\_to\_s3: Export the PostgreSQL result table back to S3
6. **Pipeline Initialization**
   * The pipeline begins with a placeholder **start task** (EmptyOperator) that signals the start of the workflow.
   * This helps visually separate the pipeline start in the Airflow UI and can be used for dependency management.
7. **Task Group group\_a – Setup and Extraction**

A **Task Group** in Apache Airflow is a way to **organize and group related tasks** visually and logically within a Directed Acyclic Graph (DAG). It helps manage complex DAGs by grouping tasks into collapsible sections in the Airflow UI and treating them as a single unit conceptually.



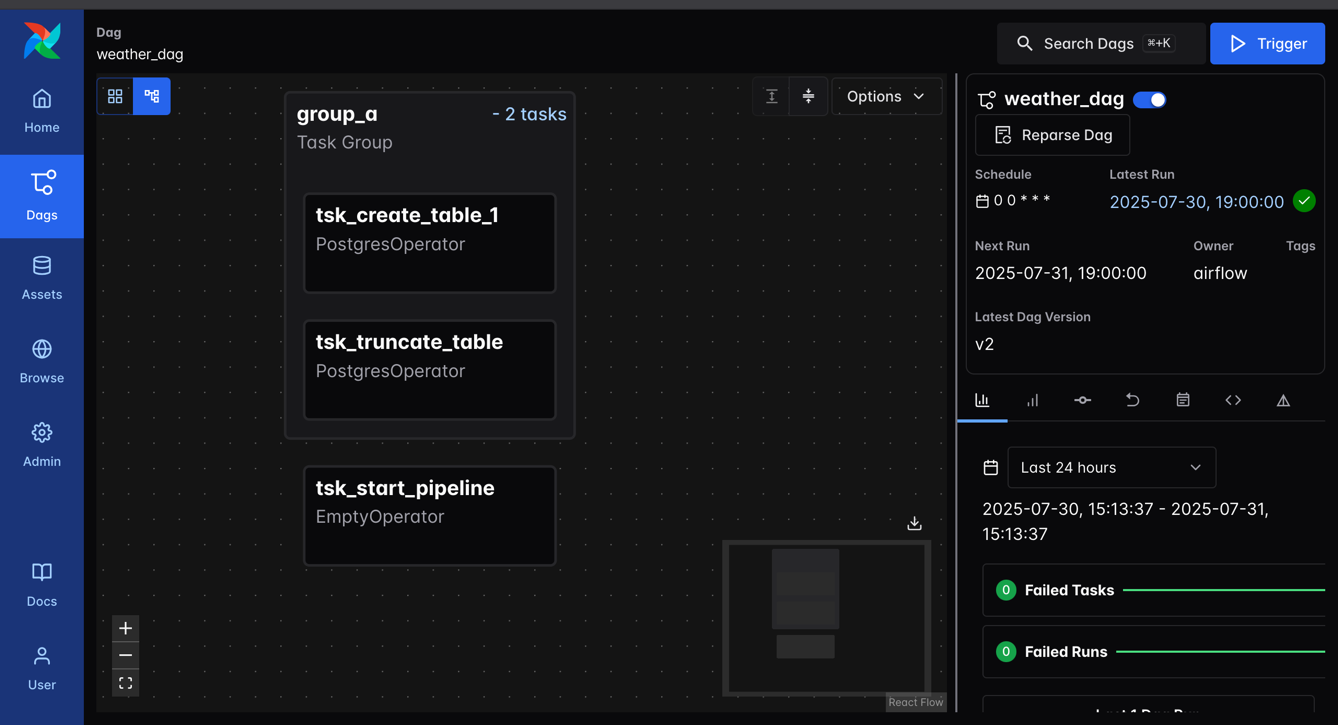
This group bundles multiple related tasks focused on setting up tables and extracting data:

a. **Create City Lookup Table**

* + Executes a SQL command to create a table named city\_look\_up if it doesn’t already exist.
  + This table will store city-level metadata like city name, state, population (census 2020), and land area.

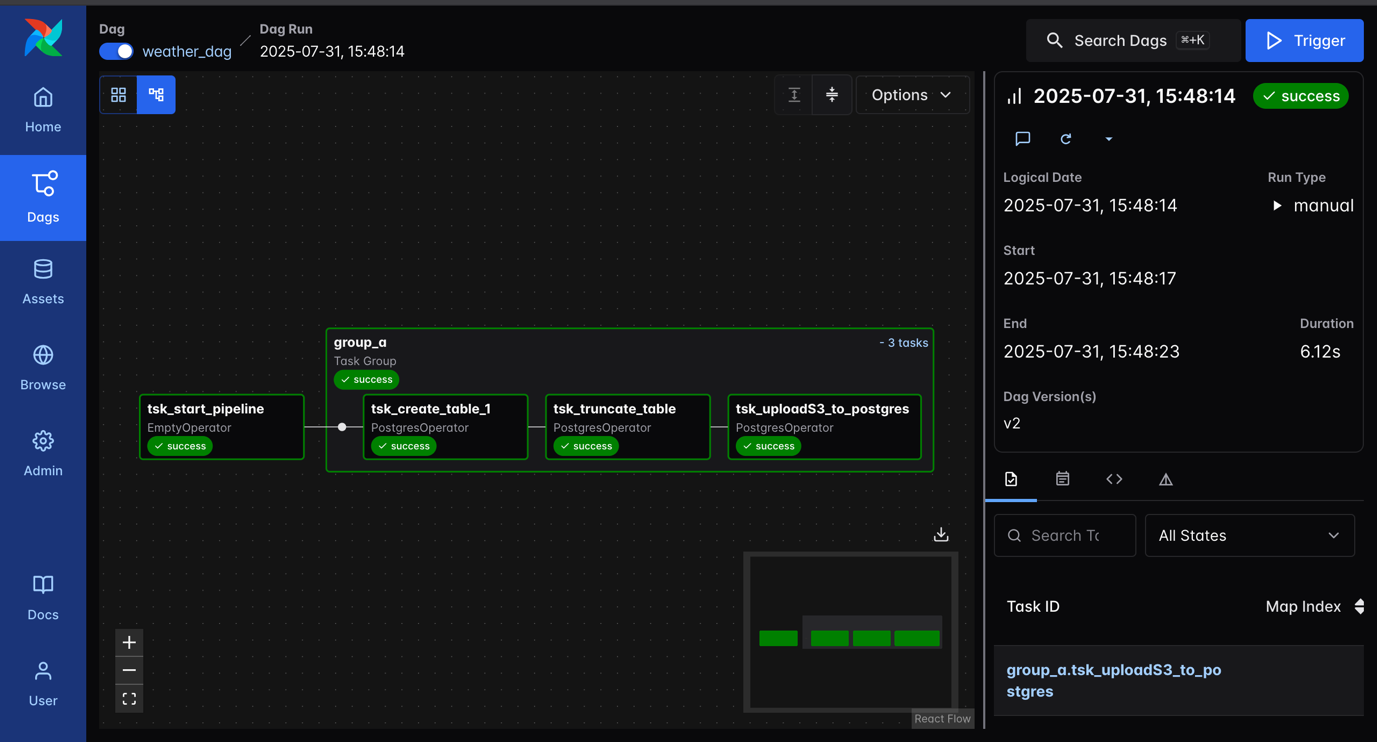
b. **Truncate City Lookup Table**

* + Empties the city\_look\_up table to ensure fresh data is loaded each run.
  + Prevents duplicates or stale data accumulation.



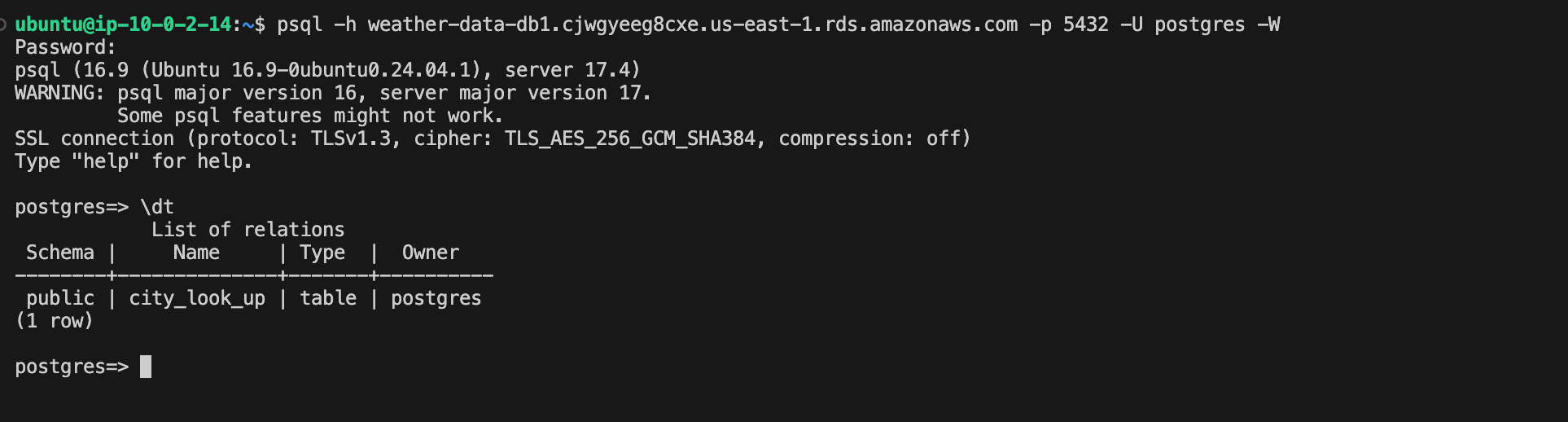
c. **Load City data from S3 into PostgreSQL**

* + Uses a PostgreSQL extension function aws\_s3.table\_import\_from\_s3 to import a CSV file (us\_city.csv) from the specified S3 bucket.
  + Loads the fresh city metadata into city\_look\_up table.



Check the database

psql -h weather-data-db1.cjwgyeeg8cxe.us-east-1.rds.amazonaws.com -p 5432 -U postgres -W



**d. Create Weather Data Table**

* + Creates a weather\_data table if it doesn’t exist, which will hold weather metrics.
  + The columns are designed to store city name, weather description, temperature (in Fahrenheit), humidity, pressure, wind speed, and timestamps related to recording time, sunrise, and sunset.

e. **Check Houston Weather API Availability**

* + This is a sensor that continuously checks if the external Houston weather API endpoint is responsive and ready.
  + It waits until a successful HTTP response is received before continuing.

f. **Extract Houston Weather Data**

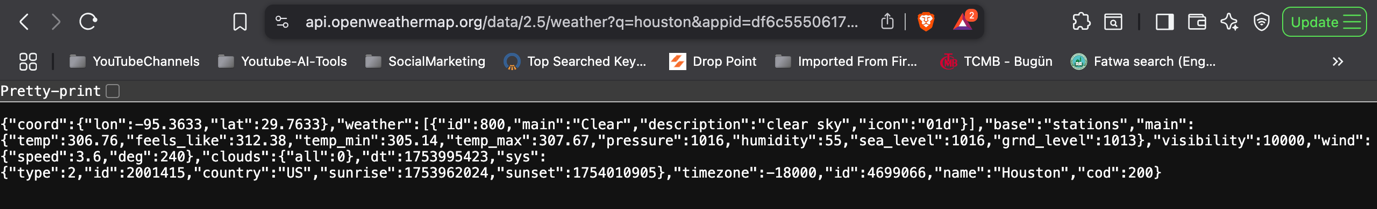
* + Makes a GET request to the weather API to retrieve the current weather data for Houston.
  + Parses the JSON response so it can be used in downstream tasks.

<https://openweathermap.org/current>

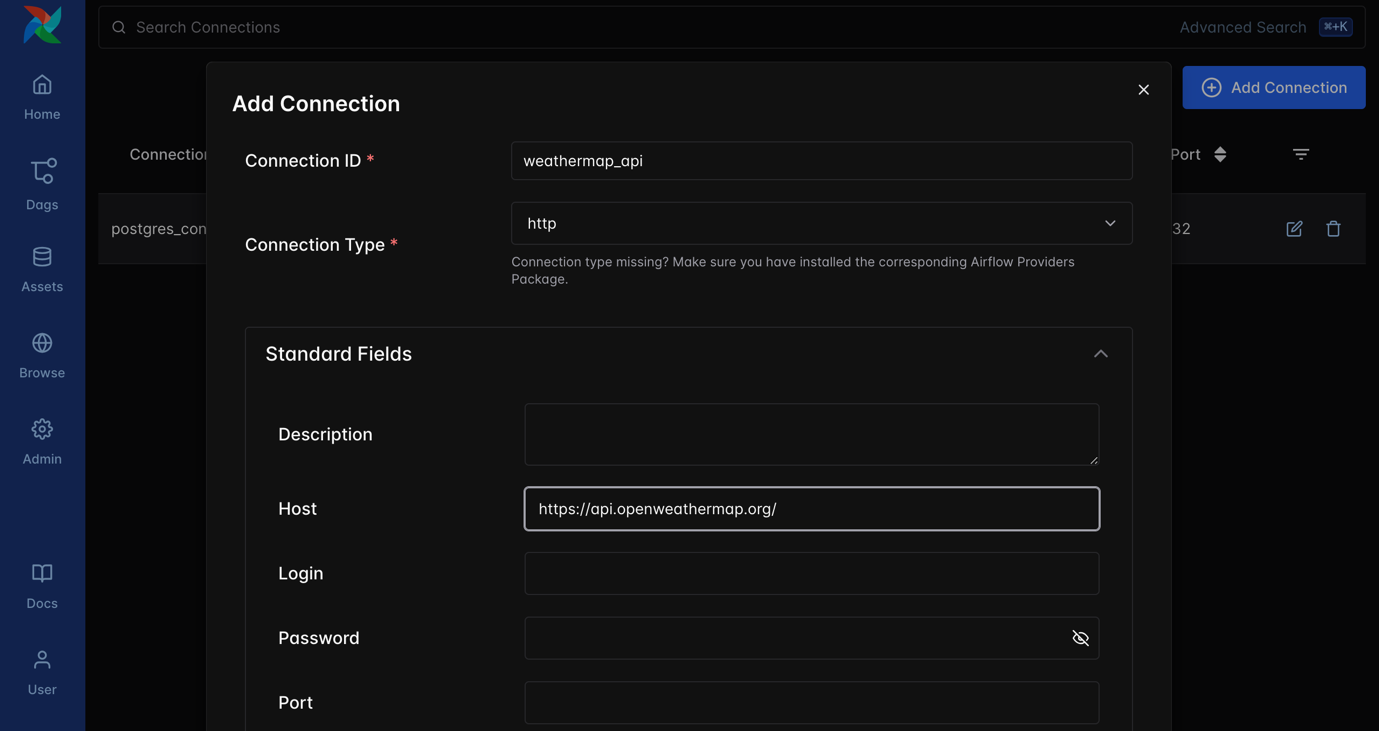
API Call:

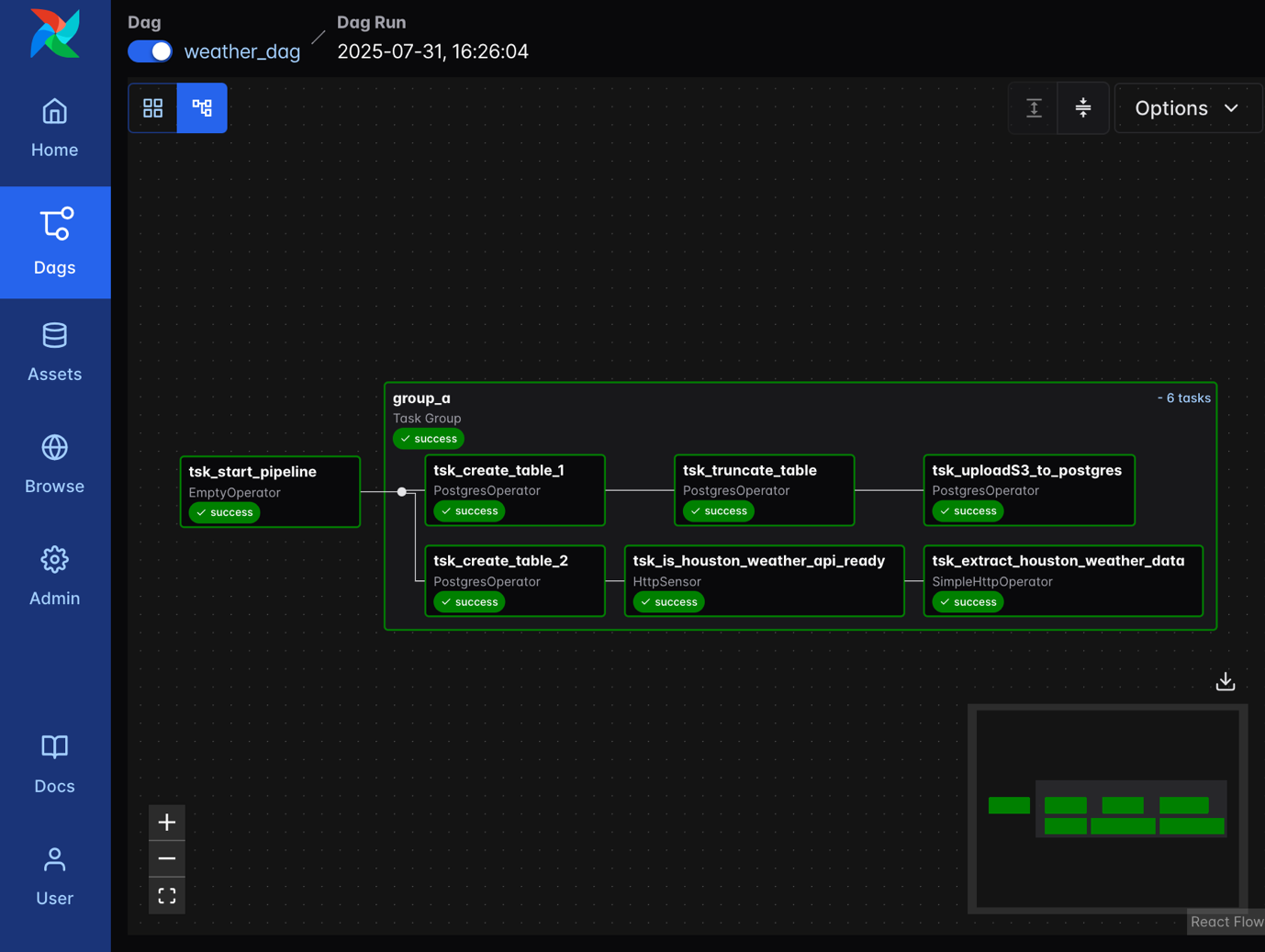
https://api.openweathermap.org/data/2.5/weather?q={city name}&appid={API key}

<https://api.openweathermap.org/data/2.5/weather?q=houston&appid=df6c5550617999ddd0e430be45366d98>



Create connection:





Check XCOM to see extracted data.

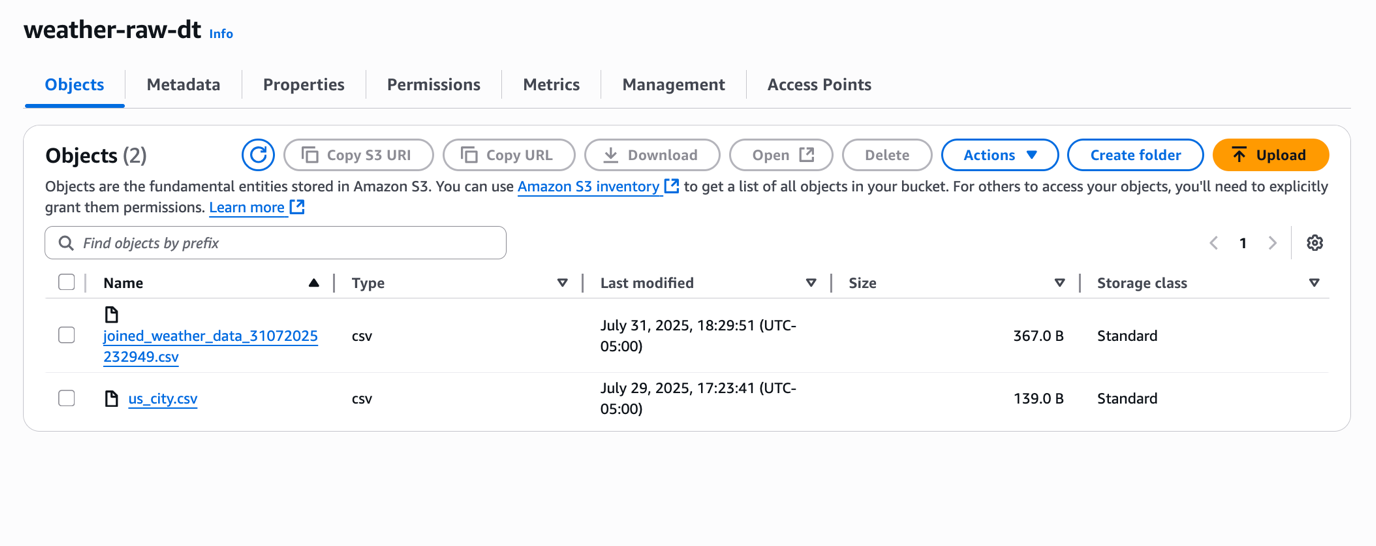
g. **Transform and Prepare Weather Data**

* + A Python function pulls the raw weather data from the API task.
  + Converts temperature values from Kelvin to Fahrenheit using a helper function.
  + Extracts other useful weather metrics (humidity, pressure, wind speed).
  + Converts UNIX timestamps into local datetime formats (time of record, sunrise, sunset).
  + Packages the transformed data into a Pandas DataFrame and writes it as a CSV file (current\_weather\_data.csv).

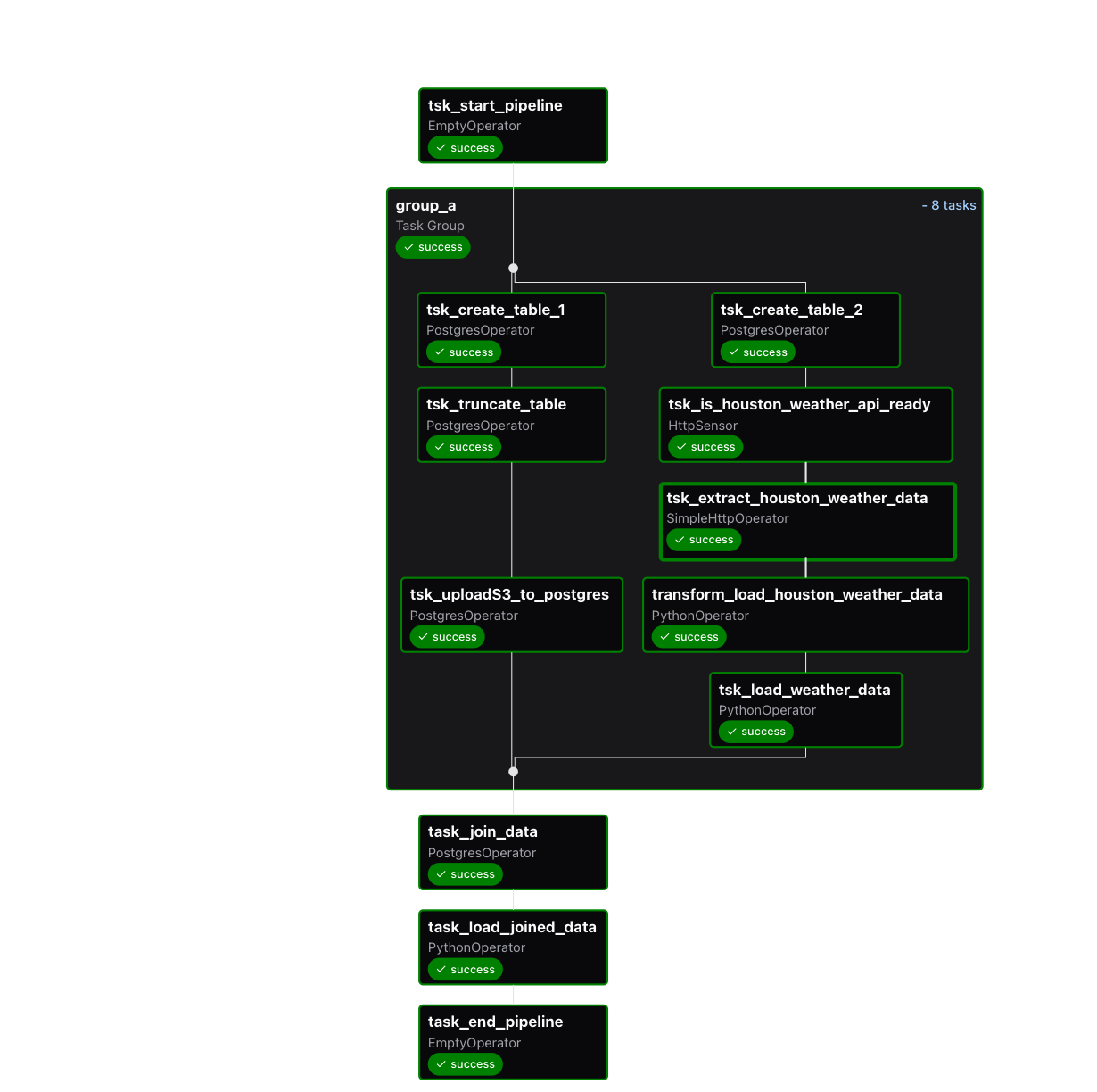
h. **Load Weather Data into PostgreSQL**

* + Loads the transformed CSV data into the weather\_data PostgreSQL table.
  + Uses PostgreSQL's efficient bulk COPY command to ingest the data.

1. **Joining Weather and City Data**
   * After the group\_a tasks are complete, a **PostgresOperator** runs a SQL query that joins the weather\_data with city\_look\_up on the city name.
   * This enriches the weather data with city-level details such as state, census population, and land area.
   * The joined result is pulled for further processing.
2. **Saving Joined Data to S3**
   * A Python task converts the joined query result into a Pandas DataFrame.
   * It dynamically creates a timestamped filename for uniqueness.
   * Then it saves the DataFrame as a CSV file directly into an S3 bucket ([weather-raw-dt](https://us-east-1.console.aws.amazon.com/s3/buckets/weather-raw-dt?region=us-east-1&bucketType=general)).
   * This step ensures the enriched weather data is persisted in cloud storage for downstream consumption or analytics.



1. **Pipeline Completion**
   * The workflow ends with another EmptyOperator as a visual and logical end marker.
   * This cleanly indicates pipeline completion in Airflow’s UI.



### **Additional Details**

* **Retries and Failures:**

The pipeline is configured to retry failed tasks twice with a 2-minute delay between retries, but it does not send emails on failure or retry.

* **Task Grouping:**

Using a TaskGroup groups logically related tasks together in the UI for better readability and management.

* **Data Flow and Dependencies:**

The tasks are connected to enforce order, e.g., tables are created before loading data, and the API sensor ensures data is only fetched when the API is ready.

* **XCom Usage:**

XCom (cross-communication) is used to pass data between tasks, such as passing the extracted API data to the Python transformation task.

### **What This Pipeline Achieves**

* **Automated daily ingestion** of weather data for Houston with supporting city metadata.
* **Data transformation** to convert raw API data into a standardized format suitable for storage and analysis.
* **Efficient loading** of data into a relational database.
* **Data enrichment** by combining weather and city information.
* **Exporting processed data** into cloud storage (S3) for further use, like reporting or machine learning.