Weather ETL Pipeline Project Report

Created by	sanagavaram badrinath
Created time	@May 27, 2025 10:09 PM
Category	Project Description
Last edited by	sanagavaram badrinath
Last updated time	@May 27, 2025 10:15 PM

Project Objective and Setup

This project's goal was to develop an automated ETL pipeline that extracts real-time weather data from the Open-Meteo API, transforms it into a structured format, and loads it into a PostgreSQL database. The project simulates a real-world data ingestion and transformation process using modern data engineering tools, with results validated through a visual database client.

I used Apache Airflow to automate and schedule the pipeline tasks. The entire environment runs in Docker containers to ensure isolated and consistent deployment. Using Docker Compose, I set up the PostgreSQL service with specific parameters for image version, port configuration, and database credentials. The database runs under the name "postgres" with the standard port 5432 exposed.

For the backend setup, I used Astro CLI to manage the Airflow environment. Astro streamlines the developer experience and comes with many built-in provider packages. I added the HTTP provider to the requirements file for API interactions within DAGs, then launched the Airflow instance through Astro CLI's development environment.

I also implemented an Airflow Settings YAML file to organize the management of connections, variables, and pools. For simplicity, I set up the HTTP and Postgres connections directly through the Airflow UI.

DAG Design and Workflow Explanation

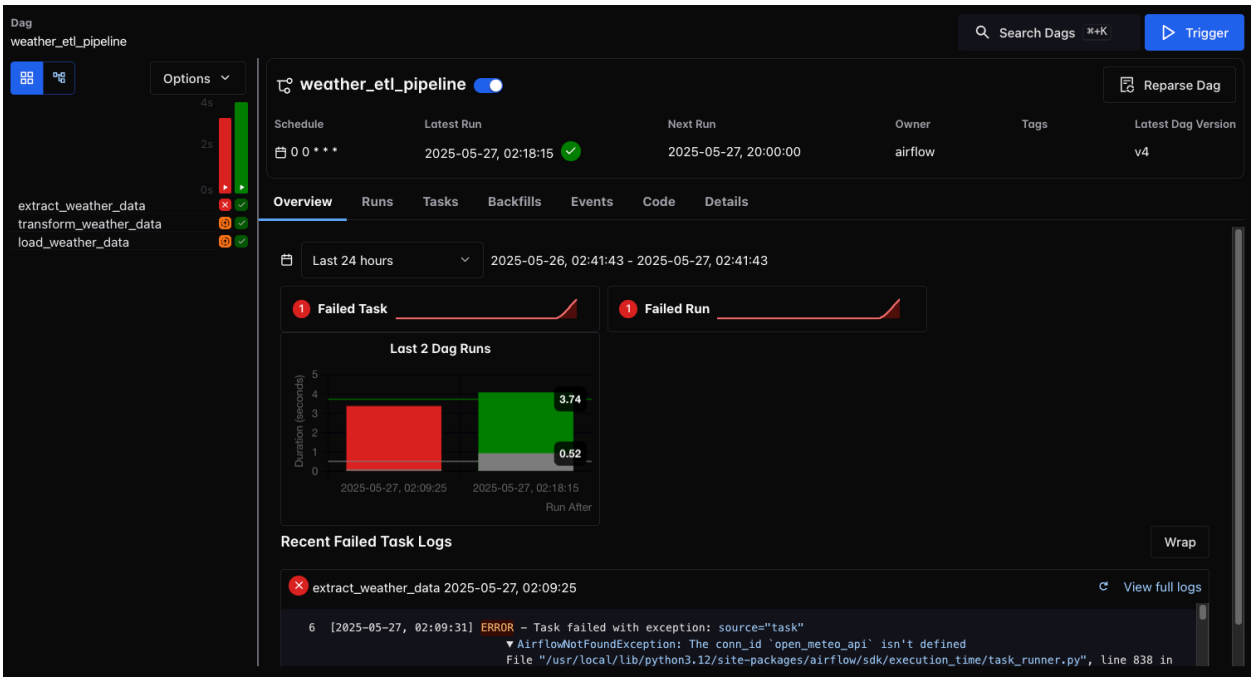
The DAG (Directed Acyclic Graph) defines our ETL workflow. I designed it to run daily under the name "weather_etl_pipeline" with three main tasks: extract, transform, and load. Each task is a decorated Python function within the DAG structure, implemented using Airflow's task decorator pattern for clean, modular design.

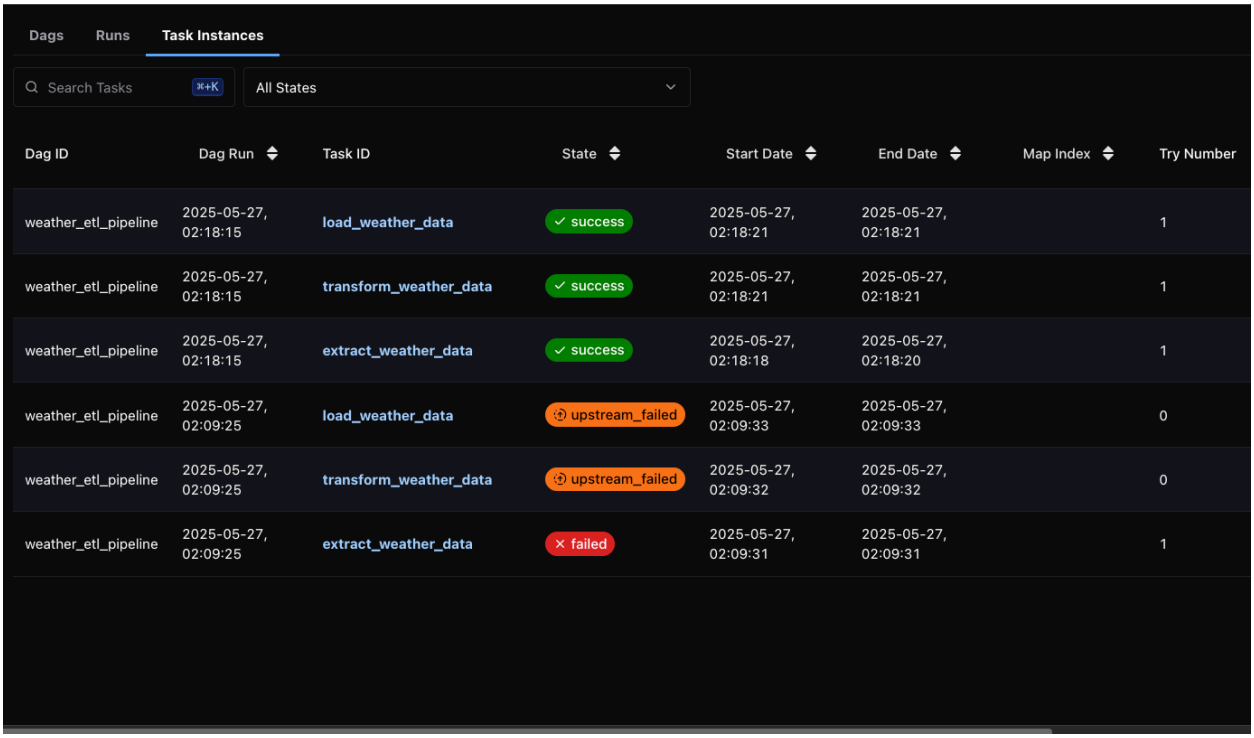
The extract task connects to the Open-Meteo API through Airflow's HttpHook. This hook uses a preconfigured connection ID in the Airflow UI for secure API endpoint handling. The task makes dynamic requests using London's latitude and longitude coordinates to fetch current weather data in JSON format.

In the transform task, I parse the JSON response to extract key weather components: temperature, wind speed, wind direction, and weather code. This step structures the data to match our PostgreSQL table format.

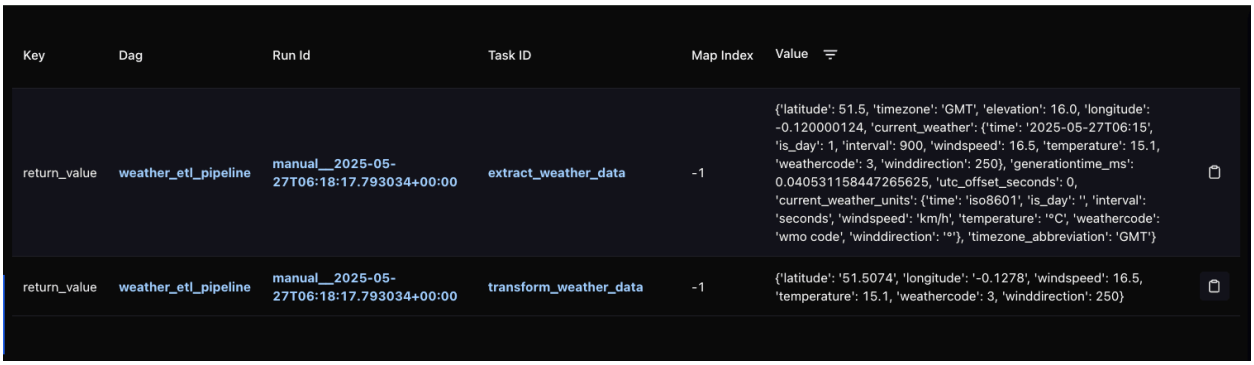
The load task, which is the final step, uses PostgresHook to connect to our Docker-based Postgres container. It creates a "weather_data" table if one doesn't exist, then inserts the transformed data. The table stores all weather metrics with timestamps for tracking. The task commits each transaction to ensure data durability.

I carefully structured the task sequence within the DAG block. After fixing an initial error where I had defined task dependencies outside the DAG block, Airflow successfully registered and executed the pipeline.

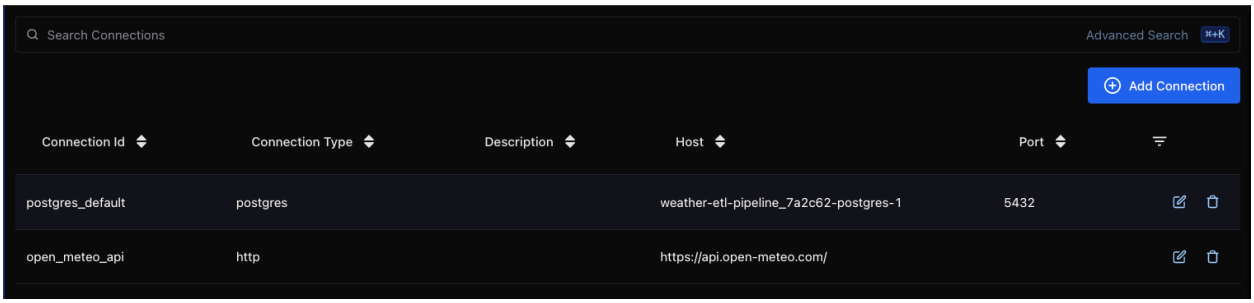




Dag ID	Dag Run	Task ID	State	Start Date	End Date	Map Index	Try Number
weather_etl_pipeline	2025-05-27, 02:18:15	load_weather_data	success	2025-05-27, 02:18:21	2025-05-27, 02:18:21		1
weather_etl_pipeline	2025-05-27, 02:18:15	transform_weather_data	success	2025-05-27, 02:18:21	2025-05-27, 02:18:21		1
weather_etl_pipeline	2025-05-27, 02:18:15	extract_weather_data	success	2025-05-27, 02:18:18	2025-05-27, 02:18:20		1
weather_etl_pipeline	2025-05-27, 02:09:25	load_weather_data	upstream_failed	2025-05-27, 02:09:33	2025-05-27, 02:09:33		0
weather_etl_pipeline	2025-05-27, 02:09:25	transform_weather_data	upstream_failed	2025-05-27, 02:09:32	2025-05-27, 02:09:32		0
weather_etl_pipeline	2025-05-27, 02:09:25	extract_weather_data	failed	2025-05-27, 02:09:31	2025-05-27, 02:09:31		1



Key	Dag	Run Id	Task ID	Map Index	Value
return_value	weather_etl_pipeline	manual__2025-05-27T06:18:17.793034+00:00	extract_weather_data	-1	{'latitude': 51.5, 'timezone': 'GMT', 'elevation': 16.0, 'longitude': -0.120000124, 'current_weather': {'time': '2025-05-27T06:15', 'is_day': 1, 'interval': 900, 'windspeed': 16.5, 'temperature': 15.1, 'weathercode': 3, 'winddirection': 250}, 'generationtime_ms': 0.040531158447265625, 'utc_offset_seconds': 0, 'current_weather_units': {'time': 'iso8601', 'is_day': '', 'interval': 'seconds', 'windspeed': 'km/h', 'temperature': '°C', 'weathercode': 'wmo code', 'winddirection': '°'}, 'timezone_abbreviation': 'GMT'}
return_value	weather_etl_pipeline	manual__2025-05-27T06:18:17.793034+00:00	transform_weather_data	-1	{'latitude': '51.5074', 'longitude': '-0.1278', 'windspeed': 16.5, 'temperature': 15.1, 'weathercode': 3, 'winddirection': 250}



Connection Id	Connection Type	Description	Host	Port
postgres_default	postgres		weather-etl-pipeline_7a2c62-postgres-1	5432
open_meteo_api	http		https://api.open-meteo.com/	

Validation and Testing with DBeaver

To validate the data loading, I connected to the Postgres container using DBeaver, a graphical SQL client. I created a new connection by entering the container hostname ("localhost"), port 5432, and the credentials from the Docker Compose file.

In DBeaver, I navigated to the weather_data table's schema and ran a SELECT query to retrieve all rows. The query returned the data ingested through the Airflow DAG, confirming that the fields matched the API response and the timestamp aligned with the scheduled interval.

This validation confirmed that our pipeline—from API call to database storage—worked correctly. Through SQL queries and DBeaver's visual interface, I verified the table structure, data types, and records. Seeing real-time data transform from an external API into a structured, queryable format was particularly rewarding.

Throughout development, I maintained modular and scalable practices. I leveraged Python libraries like JSON and requests alongside Airflow's hooks to handle connections and responses efficiently. With incremental testing of each component, the final DAG proved both reliable and maintainable.

In conclusion, this project deepened my understanding of workflow orchestration, API integration, containerized infrastructure, and relational data modeling. It provided valuable hands-on experience building a complete data pipeline using industry-standard tools and best practices.