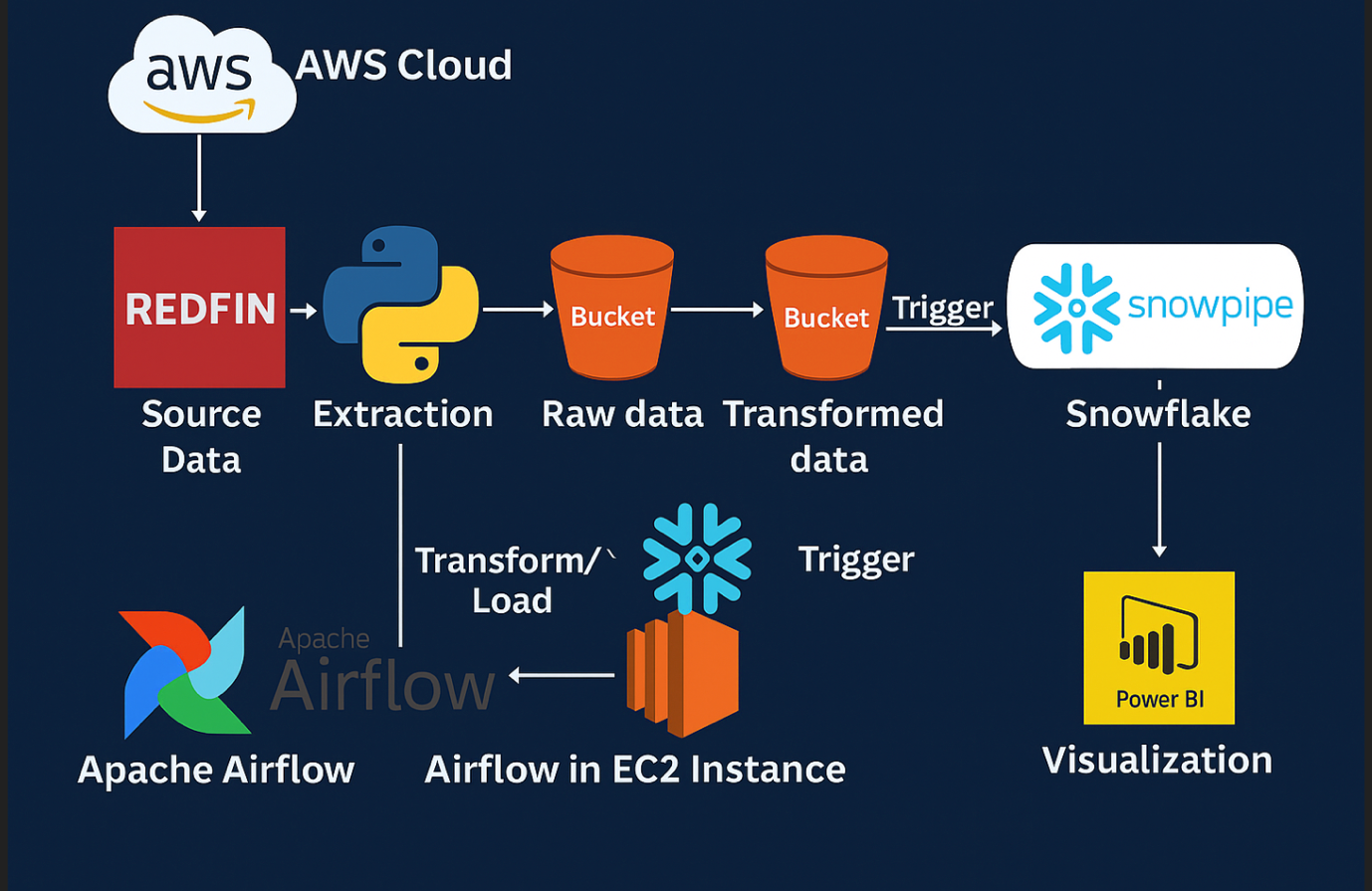
**CLOUD-NATIVE PYTHON-BASED ETL PIPELINE on AWS with AIRFLOW ORCHESTRATION and SNOWFLAKE**

In this project, the goal is to build an end-to-end data pipeline that extracts real estate data from the Redfin data source and processes it entirely on the AWS cloud. The pipeline begins by provisioning an EC2 instance using an Ubuntu AMI, where Apache Airflow is installed to orchestrate and automate the workflow. The raw data is first extracted from Redfin and stored in an Amazon S3 bucket. Using Python, the data is then cleaned and transformed before being saved to a separate S3 bucket. Once the transformed data lands in S3, it automatically triggers Snowpipe, a feature in Snowflake that loads the data into a Snowflake table in near real-time. Finally, Power BI is connected to the Snowflake database for data visualization.



1. **Extract** real estate data from Redfin's public datasets.
2. **Load** the raw data into an Amazon S3 bucket.
3. **Transform** the data using Python and Pandas.
4. **Store** the cleaned data back in S3 in Parquet format.
5. **Trigger Snowflake Snowpipe** automatically once the transformed data lands in S3.
6. **Load** the data into a Snowflake table.
7. **Visualize** it in Power BI connected to Snowflake
8. **Building the Apache Airflow DAG and Implement Extraction & Transformation Tasks**

We begin this step by defining the **DAG configuration parameters** and implementing the **first two PythonOperator tasks** in Airflow — one for **extracting data** from Redfin and another for **transforming** it.

Source:

<https://www.redfin.com/news/data-center/>

<https://redfin-public-data.s3.us-west-2.amazonaws.com/redfin_market_tracker/city_market_tracker.tsv000.gz>

Refer to scripts/dags/redfin\_analytics.ipynb for implementation

#### **DAG Configuration and Scheduling**

We'll specify the DAG's configuration, including:

* **Start date**: datetime(2025, 10, 2)
* **Email on failure/retry**: Set to False for now, but can be configured to send notifications
* **Schedule interval**: Since Redfin updates its data monthly (typically during the third full week), we recommend scheduling the DAG to run **weekly** or **monthly** depending on use-case.

### **Extract Redfin Data Task-** extract\_redfin\_data

We define a PythonOperator task called extract\_redfin\_data task, which downloads city-level Redfin housing data and saves it to the EC2 instance with a timestamp-based filename.

### **Transform the Data Task-** transform\_redfin\_data

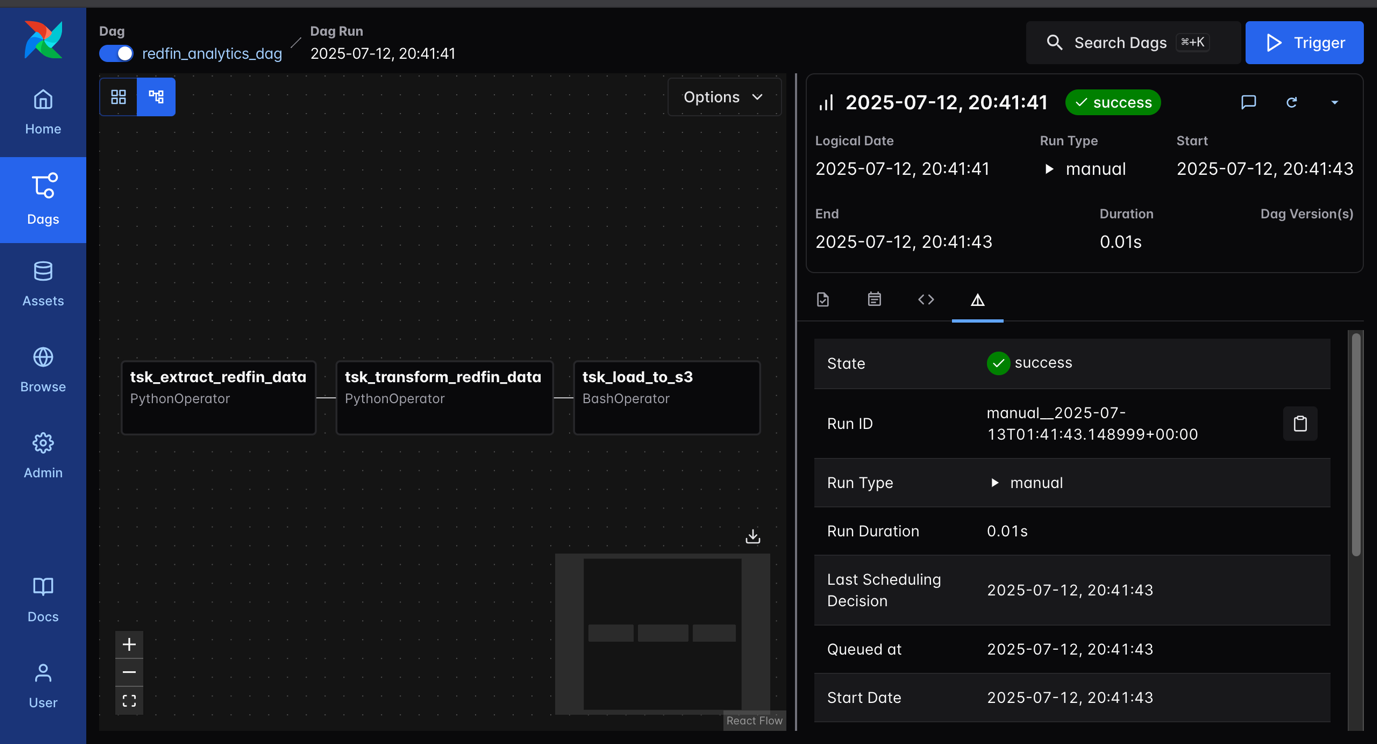
Once the raw data is extracted, we define a second PythonOperator transform\_redfin\_data task that reads the CSV, performs transformations using Pandas, and saves the cleaned data back into S3 in Parquet format.

Initial Data Exploration, Cleanup:

* + Count total rows to understand dataset size (approx. 60 million rows).
  + Detect and count null values in each column using PySpark functions.
  + Drop rows with any null values (dropna()), resulting in a cleaner dataset
  + Extract year and month from the period\_end date column as new separate columns.
  + Drop the original period\_end and last\_updated columns to reduce redundancy.
  + Map numeric month values (1-12) to their corresponding month names (January, February, etc.) for better interpretability.
  1. **Load Raw Extracted Data to S3 using BashOperator Task -** load\_to\_s3

### The raw CSV file was saved locally in EC2 during the extraction step. We want to preserve the original dataset in an S3 bucket. This way, you have: Transformed data in one S3 bucket for analytics. Raw data in another for traceability or reprocessing.

* Save the cleaned and transformed DataFrame as a Parquet file to a dedicated AWS S3 bucket.
* Use overwrite mode to ensure that the output replaces any existing data.
* Confirm via the AWS S3 Console that the Parquet file was successfully written.



1. **Load Transformed Redfin Data into Snowflake via Snowpipe (with Airflow Orchestration)**

Goal is to automate the process of loading transformed CSV data from your S3 bucket into Snowflake every time Airflow drops a new file into S3, Snowflake ingests it without manual intervention.

|  |  |
| --- | --- |
| Component | Role |
| S3 | Stores your cleaned/transformed CSV files (Airflow uploads them here) |
| Snowflake | Data warehouse where you’ll query, analyze, and visualize data |
| Snowpipe | A Snowflake service that **automatically detects and ingests new files** from S3 into a target table |
| Airflow (on EC2) | Still orchestrating the ETL process — uploads transformed data to S3 |

* + - * Creating Snowflake resources: database, schema, and table
      * Configuring Snowpipe to auto-load data from S3
      * Connecting S3 event notifications to trigger Snowpipe
      * Testing the pipeline end-to-end
      * Visualizing Snowflake data with BI tools

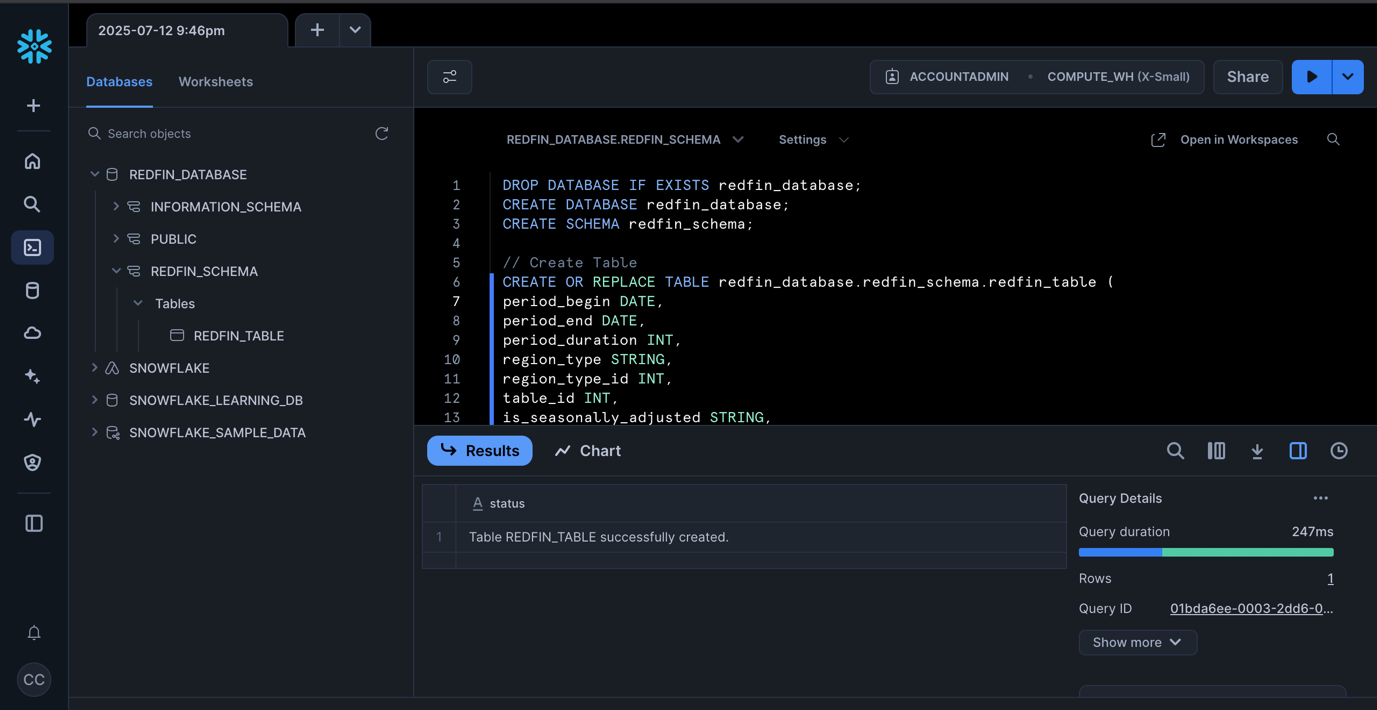
### **Create Database, Schema, and Table in Snowflake**

Snowflake needs to know where to put the data (database + schema + table). You can’t load CSVs without predefining the structure. Snowflake needs to understand the structure and rules of your files (CSV, JSON, etc.) to correctly parse and ingest them.

Snowflake needs **compute power** to execute queries. This is done using **virtual warehouses**.

* + Create your table matching the transformed CSV structure

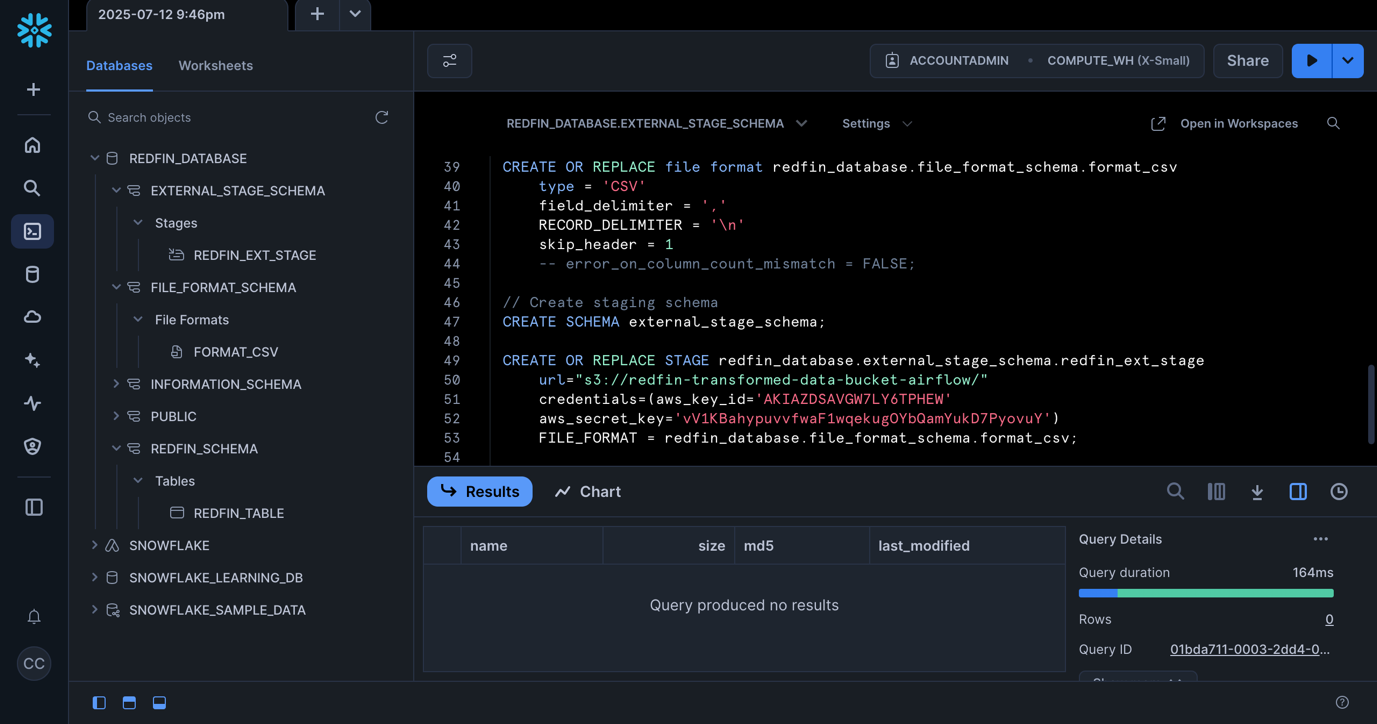
Refer to *scripts/snowflake/redfin\_snowflake.sql* for implementation

****

### **Create and Configure Snowpipe to Auto-Load from S3**

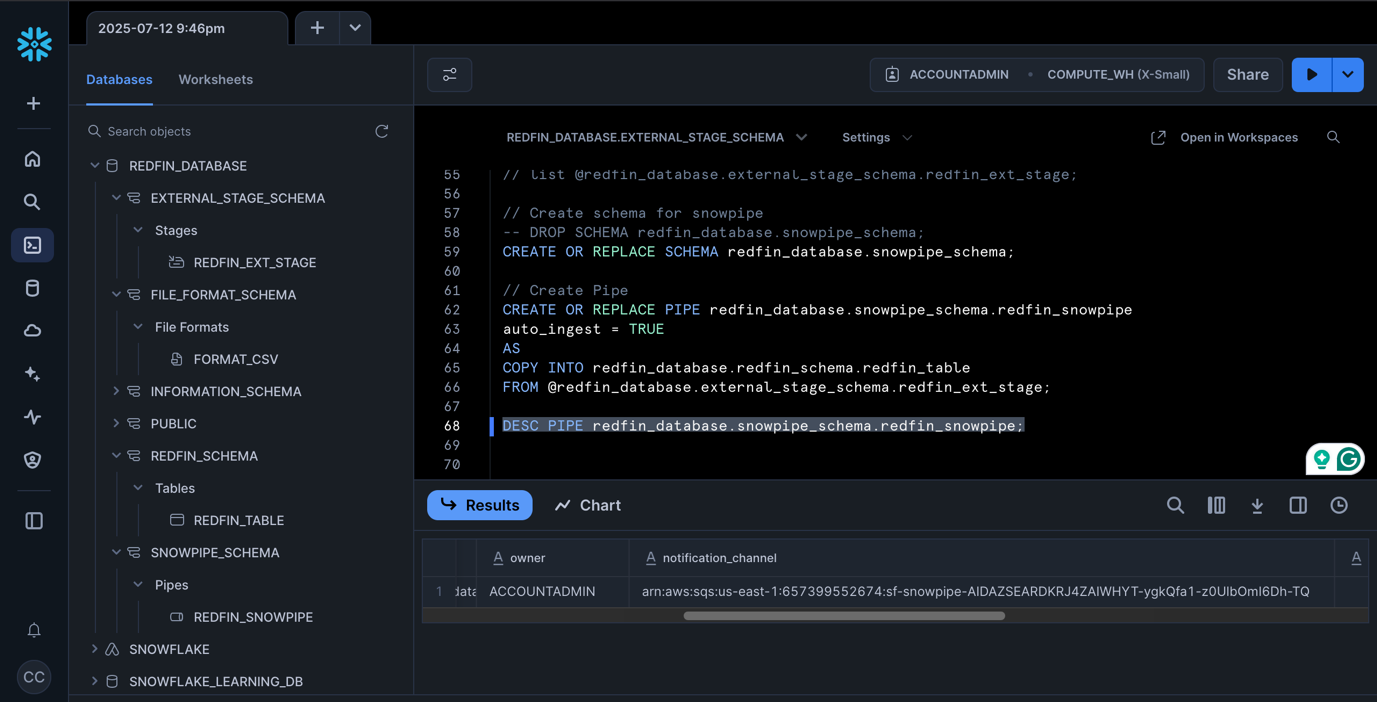
**Snowpipe** is a service that: Watches a specified S3 location. When a new file appears, it **automatically runs a COPY INTO command to insert** the data into a table. Without Snowpipe, you'd need to manually run SQL to load data every time Airflow uploads a file. With Snowpipe, this becomes real-time and event-driven.

* Create **STAGE** in Snowflake pointing to S3 bucket. A stage is a reference to an external data location. In this case, you want Snowflake to know:
  + Where to look (your S3 bucket)
  + How to authenticate (AWS keys or IAM role)
  + How the data is structured (via file format object)
  + Use/Create an IAM role in AWS that:
    - Allows Snowflake to assume the role
    - Grants read access to the specific S3 path (s3://redfin-transformed-data-bucket-airflow/)
* Create **PIPE** (Snowpipe) to ingest data. Just like with file formats and stages, you keep Snowpipe definitions in their own schema. This separation is essential in larger pipelines with dozens of pipes. Manually running COPY INTO every time a file lands in S3 is error-prone and slow. **Snowpipe automates this.** When a new file is detected in the stage:
* It reads the file
* Uses the file format definition
* Runs the COPY INTO command behind the scenes
* Inserts rows into your target table within seconds or minutes
* Snowpipe runs in **micro-batches**, not streaming
* It's **event-driven** if you connect S3 notifications (see next step)
* Can be triggered by **manual call, S3 event notification, or REST API**
* You configure Snowpipe with:
* The target table
* The source stage
* The file format



Snowpipe works with **Amazon SQS**, which listens for S3 events. Snowflake generates an **SQS-compatible ARN** when the pipe is created.

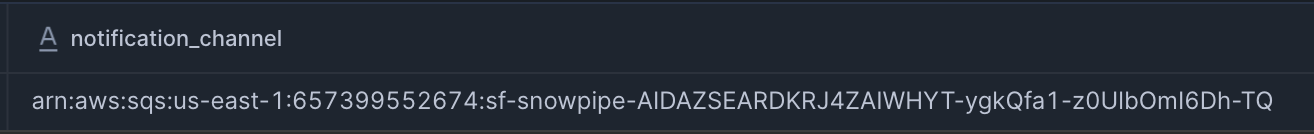
* Run DESCRIBE PIPE your\_pipe\_name
* Extract the NOTIFICATION\_CHANNEL value (an SQS ARN)
* You'll use this when setting up the event in the S3 bucket

****

### **Set Up S3 Event Notifications to Trigger Snowpipe**

You need to tell S3 to notify Snowflake (via SQS) whenever a **new object is added** to the bucket or folder where transformed data land.

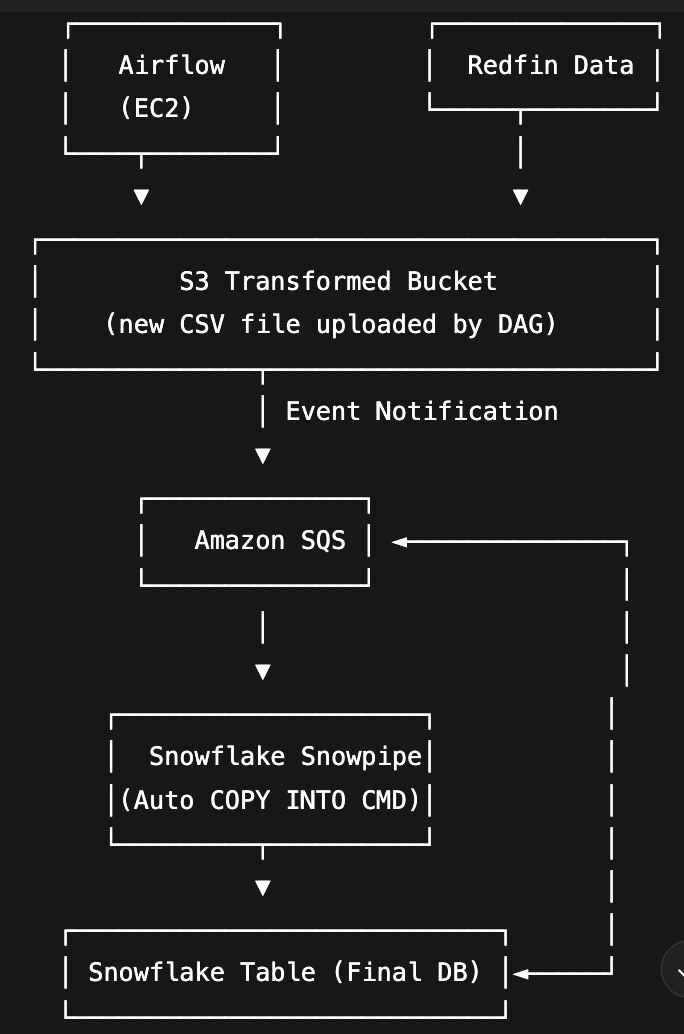
* Go to your S3 bucket’s **Properties > Event Notification**
* Create a new event:
  + **Name**: e.g., snowpipe\_trigger\_redfin
  + **Event Type**: All object create events (PUT, POST, etc.)
  + **Destination**: SQS (paste the SQS ARN from DESCRIBE PIPE)



* Snowpipe will then auto-ingest the file without delay.
  1. **Trigger Airflow DAG to Simulate Full Pipeline**

As soon as upload completes:

* S3 triggers the event
* Event hits the SQS queue
* Snowpipe is notified and runs
* COPY INTO is executed automatically
* Data appears in Snowflake table



### Next Steps: Replace EC2 with **AWS EMR** and use **PySpark** to handle big data scale

### **PART 3: Same Pipeline with PySpark on Amazon EMR: tomate EMR Jobs with Airflow**

1. **Provisioning EMR and Redfin Extraction**

3.1. Create an S3 bucket (e.g., redfin-emr) and folders:

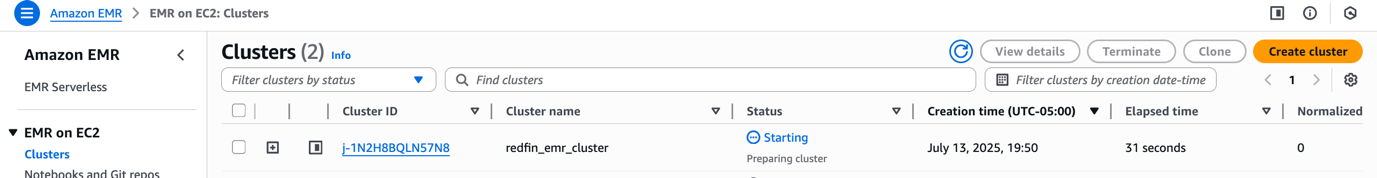
* raw\_data\_ – raw Redfin data
* transformed\_data – transformed output
* scripts – PySpark or shell scripts for EMR
* emr-logs – store EMR logs
  1. Start of the Pipeline

The workflow initiates with an EmptyOperator called start\_pipeline. This does not perform any operations but serves as a logical beginning to visually represent the entry point in the Airflow UI. Such empty operators enhance DAG readability and structure, especially in complex workflows.

* 1. Creating an EMR Cluster

The next operation involves provisioning an EMR cluster using the EmrCreateJobFlowOperator. This operator launches a new cluster according to the configuration specified in job\_flow\_overrides. The next operation involves provisioning an EMR cluster using the EmrCreateJobFlowOperator. This operator launches a new cluster using the configurations defined in job\_flow\_overrides. These configurations include the following:

* Cluster Name: redfin\_emr\_cluster
* Release Label: emr-6.13.0
* Applications Installed: Apache Spark and JupyterEnterpriseGateway
* Logging: Enabled via LogUri pointing to an S3 bucket (s3://redfin-data-emr/emr-logs/)
* Visibility: Set to False, so it's not visible to all AWS users by default
* Cluster Composition: 1 master node and 2 core nodes, all of type m5.xlarge, running as on-demand instances
* Network Settings: Deployed in subnet subnet-087d3438cd6469ba0 and uses the key pair airflow-key for SSH access
* Lifecycle: KeepJobFlowAliveWhenNoSteps set to True allows us to add steps dynamically before auto-termination
* IAM Roles: Uses standard roles EMR\_EC2\_DefaultRole for EC2 and EMR\_DefaultRole for the cluster



This operation requires the apache-airflow-providers-amazon package. Without it, Airflow cannot recognize the EMR-related operators like EmrCreateJobFlowOperator, EmrAddStepsOperator, and EmrTerminateJobFlowOperator. This package provides official AWS integrations for Airflow, allowing orchestration of EMR, S3, Lambda, Redshift, and other AWS services. If this provider is not installed, a ModuleNotFoundError will be raised.

**No need to specify aws\_conn\_id** in the operator: AWS credentials are configured using IAM roles. This makes the DAG cleaner and avoids hardcoding credentials.

Upon execution, this operator returns a unique job\_flow\_id, which will be used to track and control the cluster through the rest of the DAG.

* 1. Checking Cluster Readiness

Once the EMR cluster is requested, its readiness is confirmed using the EmrJobFlowSensor. This sensor checks whether the EMR cluster has reached the WAITING state, meaning it's provisioned and idle, ready to execute Spark jobs. The sensor accesses the job\_flow\_id (key: return value, value:CLUSTER ID (same as AWS EMR)) from the output of the cluster creation task through Airflow's XCom mechanism, which supports cross-task data sharing. The cluster status is polled every 5 seconds (via poke\_interval) until it is marked as ready. This prevents the pipeline from proceeding prematurely and ensures that resources are fully provisioned before Spark job submission begins.

Again, there is **no need to set aws\_conn\_id**.

* 1. Adding Extraction Step

After confirming that the EMR cluster is in a ready state, the DAG proceeds to submit a Spark job using the EmrAddStepsOperator. This operator is responsible for pushing executable steps (or jobs) to the EMR cluster. But to define what the cluster actually **does**, we use a Python object called SPARK\_STEPS\_EXTRACTION.

* After ensuring the cluster is in a ready state, the DAG submits a Spark job using the EmrAddStepsOperator.
* This job adds an EMR step defined in the SPARK\_STEPS\_EXTRACTION object. The step leverages a built-in EMR script runner JAR located at:

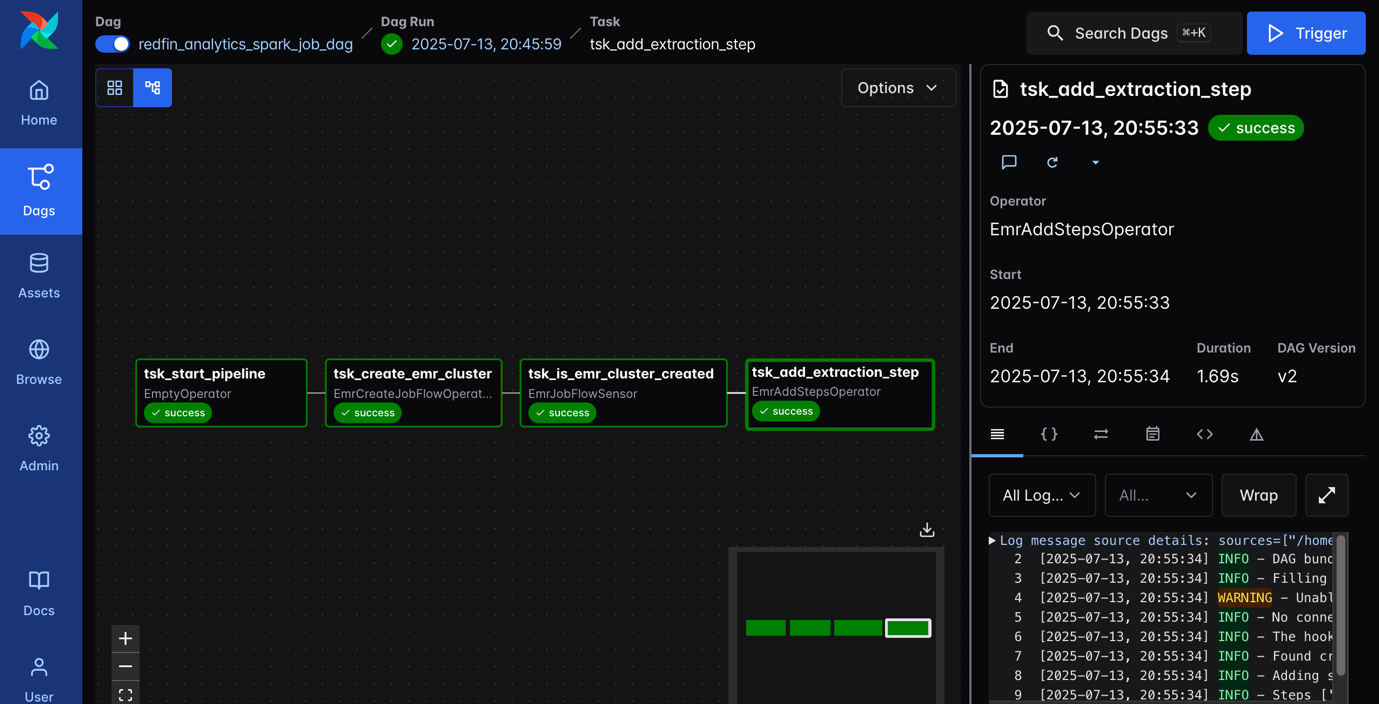
s3://<region>.elasticmapreduce/libs/script-runner/script-runner.jar

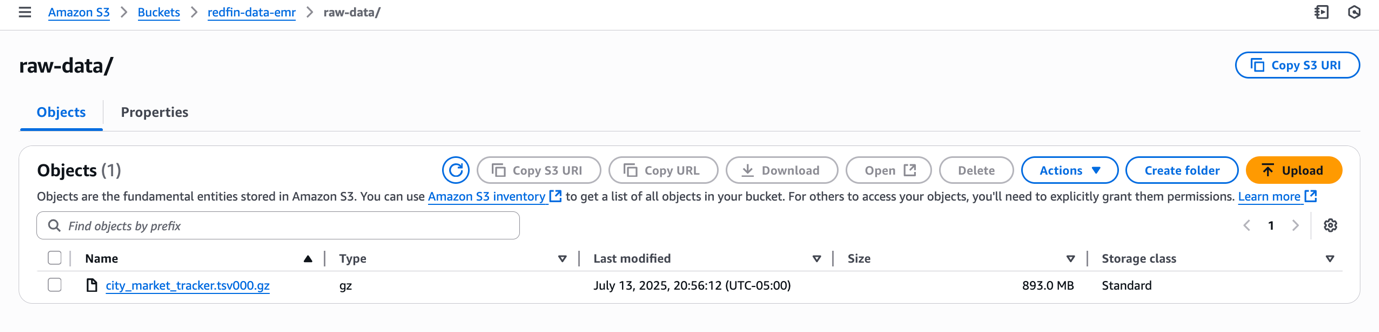
* This ingest.sh script is responsible for downloading real estate data from Redfin using wget, unzipping the file, and storing it in the raw data zone of your S3 bucket.

wget -O - https://redfin-public-data.s3.us-west-2.amazonaws.com/redfin\_market\_tracker/city\_market\_tracker.tsv000.gz | aws s3 cp - s3://redfin-data-emr/raw-data/city\_market\_tracker.tsv000.gz

* SPARK\_STEPS\_EXTRACTION tells EMR *what* to do (run a shell script): It is a **list of dictionaries** that describe one or more EMR steps in a format compatible with AWS EMR. In our case, it includes a single step called **"Extract Redfin data"**.
* EmrAddStepsOperator sends that job to the active EMR cluster

The job\_flow\_id is dynamically retrieved using Airflow’s XCom system, pulling from the cluster creation task’s return\_value. This allows the operator to submit the Spark step to the correct EMR cluster instance. The SPARK\_STEPS\_EXTRACTION itself is a list of step configurations where the action is defined as CANCEL\_AND\_WAIT on failure, providing a safe stop in the event of step failure. Notably, AWS connection settings like aws\_conn\_id are not required if Airflow is already configured with IAM roles or environment-based credentials.





* 1. Monitoring Extraction Completion

The pipeline then uses an EmrStepSensor to monitor whether the data extraction job has completed. It fetches the specific step ID from XCom and continues to check until the Spark job reports the status COMPLETED. This continuous monitoring ensures the DAG does not proceed until extraction is verified to be successful.

* 1. Adding Transformation Step

Upon successful extraction, a transformation step is submitted to the cluster using another EmrAddStepsOperator. This Spark job invokes a PySpark script (transform\_redfin\_data.py) using spark-submit and command-runner.jar. The script reads raw data from the raw zone S3 path, performs cleaning (e.g., handling nulls, selecting specific columns), and writes the processed data to a separate S3 path, the transformed zone, in Parquet format.

This script reads raw data from the raw zone S3 path, performs cleaning (e.g., handling nulls, selecting specific columns), and writes the processed data to a separate S3 path, transformed\_data in **Parquet** format. This shift from CSV to Parquet ensures optimized storage and performance for downstream data consumption.

* 1. Monitoring Transformation Completion

As with extraction, the transformation step’s progress is tracked by another EmrStepSensor. It listens for the COMPLETED status for the transformation task. As the Spark job runs, the sensor reports the intermediate states like PENDING and RUNNING, until it changes to COMPLETED, ensuring accurate progress tracking.

* 1. Terminating the EMR Cluster

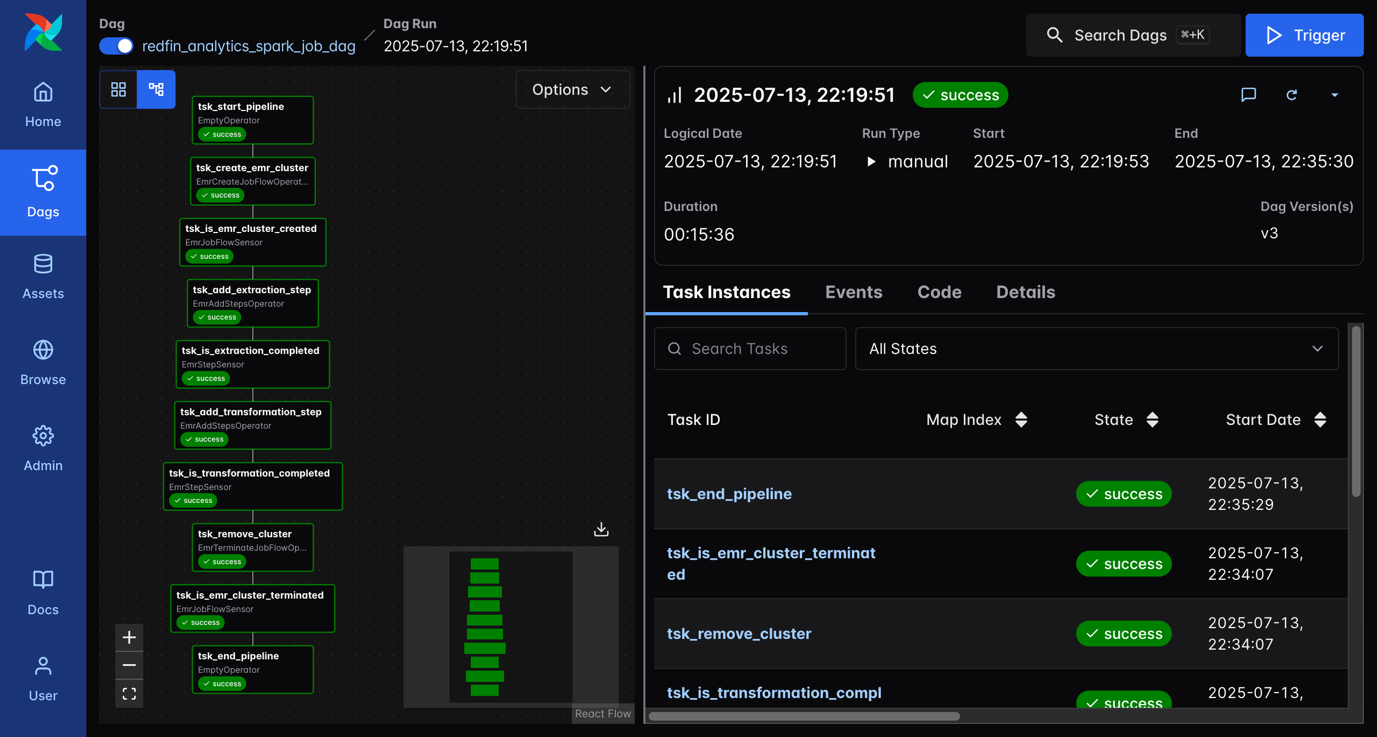
With all processing done, the cluster is programmatically terminated using EmrTerminateJobFlowOperator. This step is crucial to control AWS costs by stopping resource consumption. The operator uses the original job\_flow\_id to identify which cluster instance to shut down.

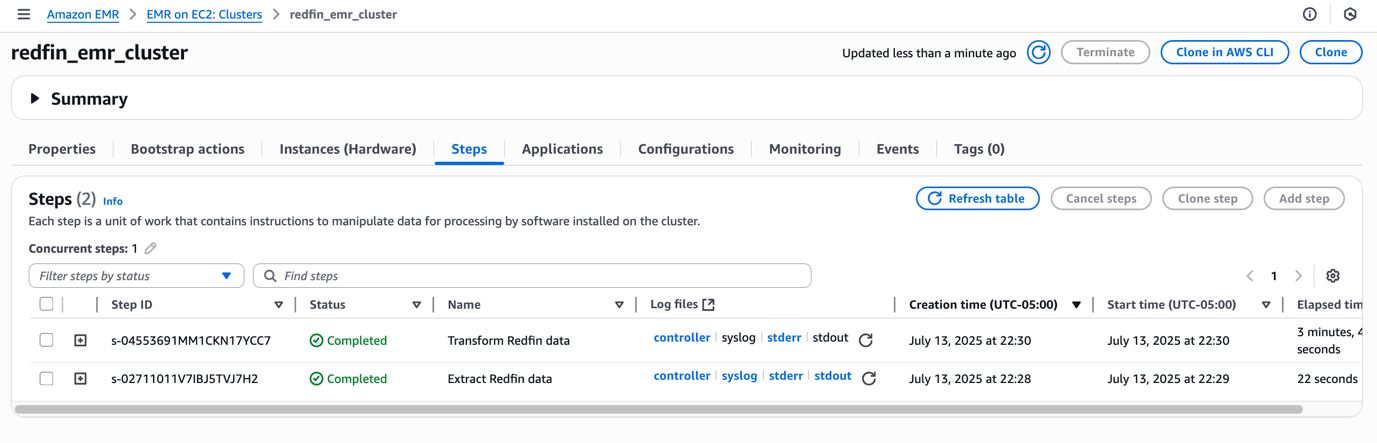
* 1. Confirming Termination

A second EmrJobFlowSensor is used to validate that the EMR cluster has been successfully terminated. It polls the cluster status at regular intervals until it confirms a TERMINATED state, ensuring the workflow does not exit prematurely.

* 1. End of the Pipeline

The DAG concludes with a DummyOperator named end\_pipeline. This provides a clear visual marker indicating successful completion of the pipeline. In large or branched DAGs, such markers are essential for easy navigation and logical grouping.





* 1. **Validation and Triggering Snowflake Pipeline**

After the transformation step is complete, the processed Parquet files are verified to be written to the transformed S3 path. This event sets the stage for Snowpipe in Snowflake to be triggered. Snowpipe, which listens to S3 bucket events, initiates a COPY INTO operation to load Parquet data into a Snowflake table.

Refer to step 2.3 and 2.4 above

Notification\_channel to create S3 event to trigger snowpipe:

arn:aws:sqs:us-east-1:657399552674:sf-snowpipe-AIDAZSEARDKRJ4ZAIWHYT-ygkQfa1-z0UlbOmI6Dh-TQ