Zillow API ETL pipeline with Apache Airflow

Introduction

This *Zillow API ETL Pipeline* project involves developing an automated data pipeline to extract real estate data from the Zillow API, transform it for analytical use, and load it into a target database. Using **Apache Airflow** as the orchestration tool, the pipeline ensures reliable, scheduled data extraction and transformation processes. This setup provides a robust solution for integrating Zillow's rich data into data-driven decision-making systems, facilitating real-time analytics, forecasting, and trend analysis in real estate markets.

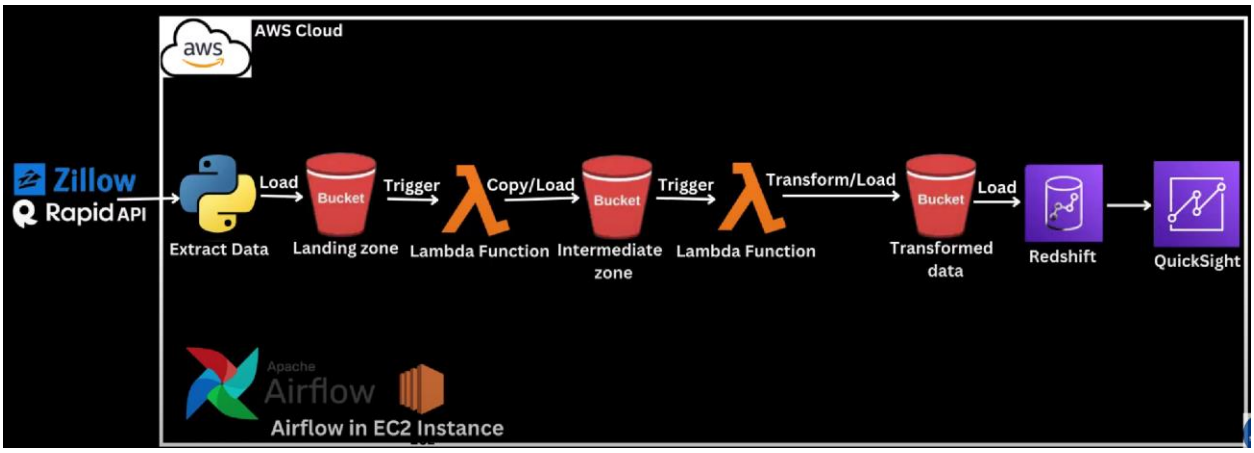
The key services utilized in this project include:

- **Amazon EC2** hosts Apache Airflow to manage and schedule the ETL workflows, providing reliable automation and orchestration.
- **Amazon S3** acts as intermediate storage for raw and transformed data, ensuring durability and easy access within the ETL flow.
- **Python** handles data extraction, enabling customized data processing from the Zillow Rapid API.
- **Lambda functions** execute specific, event-driven processing steps within the ETL pipeline, reducing the need for persistent compute resources.
- **Apache airflow** orchestrates the end-to-end ETL process, ensuring each stage from extraction to loading is timely and coordinated.
- **Redshift** stores the transformed data, enabling efficient querying and analysis of large real estate datasets.
- **QuickSight** provides visualization and dashboarding for insights from the processed Zillow data, supporting data-driven decisions.

Architecture Diagram

In the *Architecture Diagram* section, the Zillow API ETL Pipeline architecture shows a streamlined, automated data flow across several AWS services, coordinated through Apache Airflow. The process begins with **Python scripts** hosted on **EC2**, which make scheduled requests to the Zillow API to extract real estate data. **Apache Airflow** orchestrates these extractions at defined intervals, ensuring consistent data retrieval. **Lambda functions** are utilized for specific, event-driven transformations, optimizing the data structure for storage. **Amazon S3** serves as intermediate storage, holding both raw and processed data, making it easily accessible for subsequent processing steps. The transformed data is then loaded into **Amazon Redshift**, which provides structured, query-ready storage, enabling efficient analysis of large datasets. Finally, **Amazon QuickSight** accesses Redshift to generate dynamic visualizations and dashboards, offering clear insights into real estate trends and metrics for decision-making. This architecture highlights the coordinated use of AWS services to achieve a robust, scalable ETL pipeline for real estate data analytics.

Figure 1.0: Architecture diagram.



Execution

Launching an EC2 instance

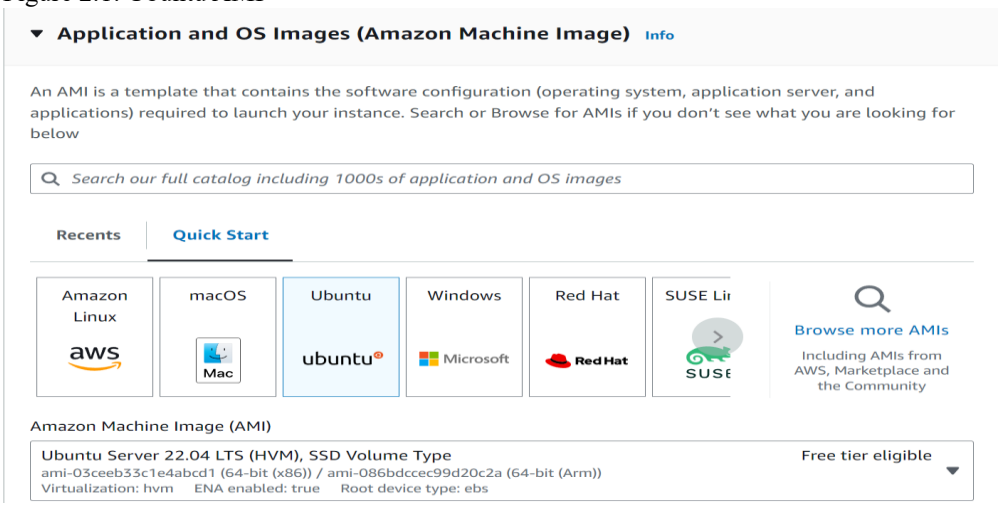
Launching an EC2 instance creates a virtual server that is accessible on the public internet. This server will host my airflow service and other dependencies needed for the project.

I also enabled SSH (secure shell) a protocol that secure remote access to computers or servers over an encrypted connection. I enabled SSH traffic so that I can connect to my EC2 server directly from my VSCode IDE.

Setting up a key pair - A key pair is a private security key that enables you to connect to your virtual machine through your physical computer. Once I setup my key pair, AWS automatically downloads a copy of the key pair into your local machine. AWS offers provision to either download the key pair as a .PEM file for “SSHing” via VSCode/CMD or .PPK, a PuTTY private key file. I used SSH in this project to connect to my instance, either approach works.

For this project I used the Ubuntu server 22.04 AMI which is free tier eligible. You can also choose to use the Ubuntu server 24.04 which is the latest Ubuntu AMI as at the time of this project.

Figure 2.1: Ubuntu AMI



For the instance type I went with t2.medium. This option allows for 2vCPUs and 4GiB of memory. However, it should be noted that this instance type is not on the free tier eligible band.

I then generated a key pair, this will be useful when SSHing into my server from my IDE.

Next step is to allow the creation of a security group, this option puts an extra layer of security on your EC2 by controlling the source of traffic to the instance. See image below.

Figure 2.2: Network settings

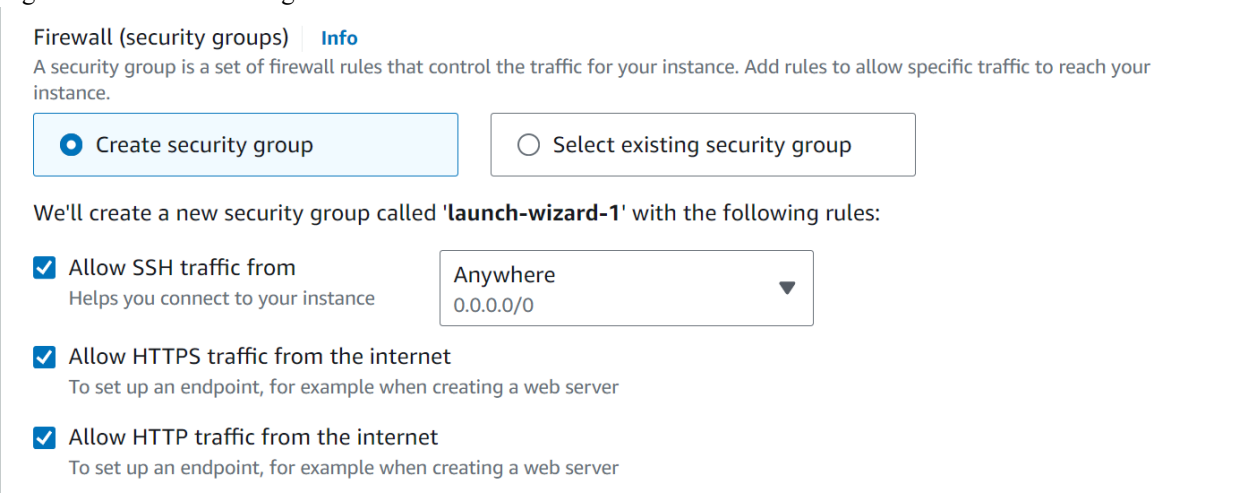
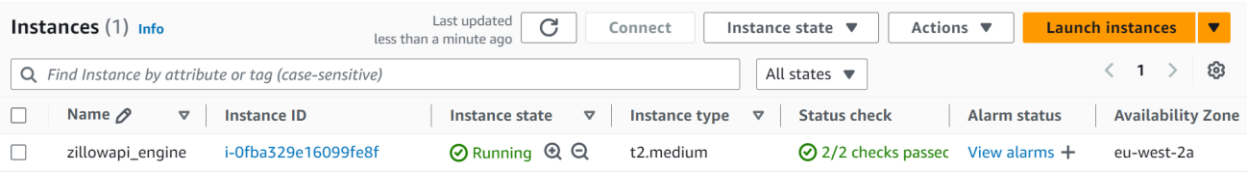


Figure 2.3: Successful EC2 instantiation



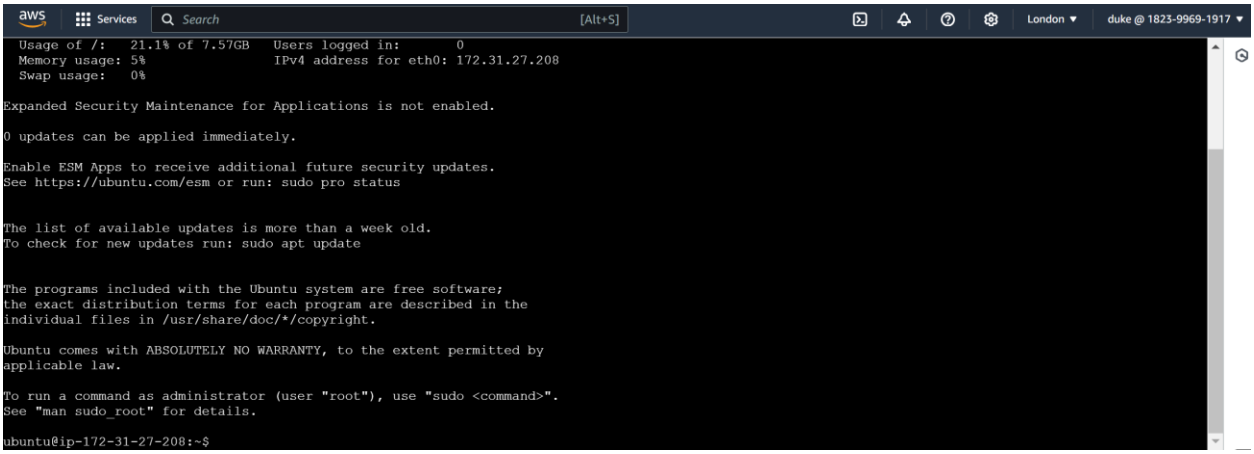
Installing project dependencies

Next step is to install the project dependencies needed for this project.

In the EC2 instance connect window type out the below commands:

1. sudo apt update – This tells the server to look for any and all updates for the already running programs in the server. It is essential for the overall health of the server just to ensure that everything is running on updated services.
2. sudo apt install python3-pip – Installs python 3 on the server.
3. sudo apt install python3.10.venv – Installs the python 3.10 virtual environment.
4. python 3 -m venv [name of environment] – Creates a custom name for the venv.
5. source [name of environment]/bin/activate – Activates the venv.
6. pip install --upgrade awscli – Installs the AWS CLI.
7. pip install apache-airflow – Installs apache airflow service.
8. Airflow standalone – Instantiates the airflow ready for DAGs.

Figure 2.4: EC2 instance connect



I needed to connect to my EC2 in two simultaneous sessions. One session is to monitor the performance of my airflow service and the other session to be able to pass commands to the instance if needed. This will clearly be understood when I install apache airflow on the server.

Figure 2.5: AWSCLI installation

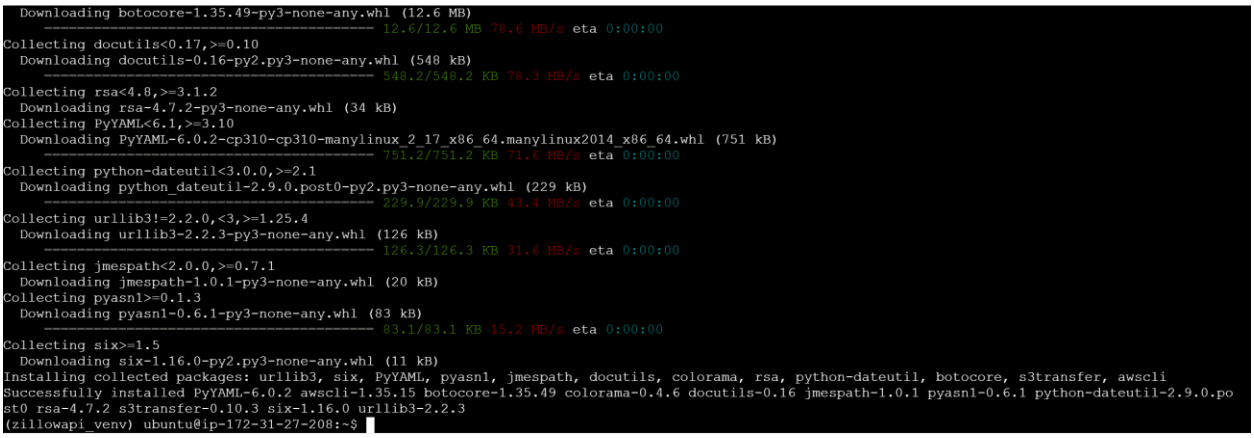


Figure 2.6: Apache airflow installation

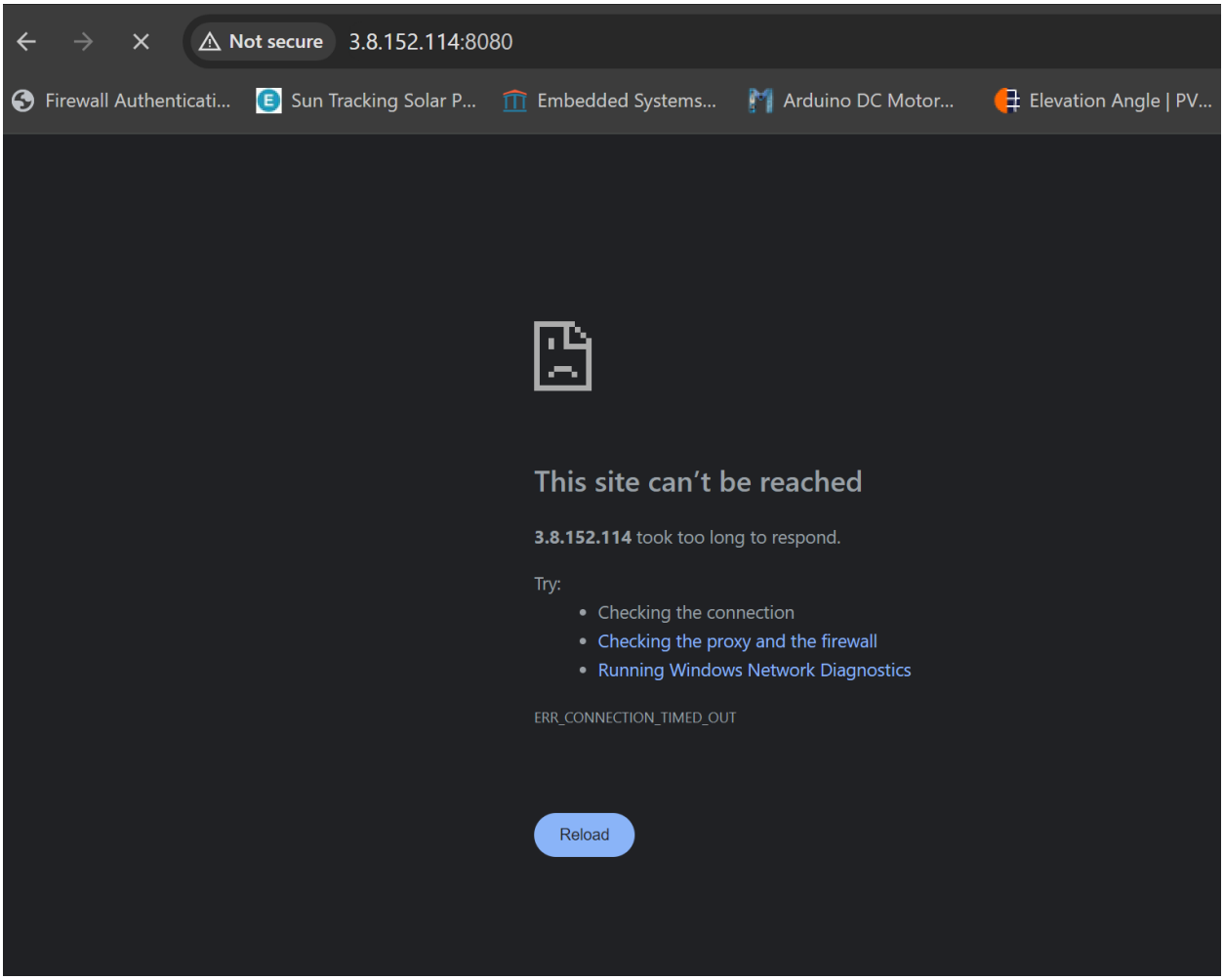
After running airflow standalone, the airflow server allocates you a username: “Admin” and a password: [“password”]. This is the password I used to login to my airflow UI.

Figure 2.7: Airflow credentials

```
standalone | Airflow is ready
standalone | Login with username: admin  password: F8mDGTX6aKy6rFqp
standalone | Airflow Standalone is for development purposes only. Do not use this in production!
triggerer  | [2024-10-26T02:56:44.286+0000] {triggerer_job_runner.py:510} INFO - 0 triggers currently running
```

In order to access the UI, I needed to copy the public URL of my EC2 server and open it on a new browser and append the port 8080 to the IP. This is because apache server runs on port 8080. See image demonstration below.

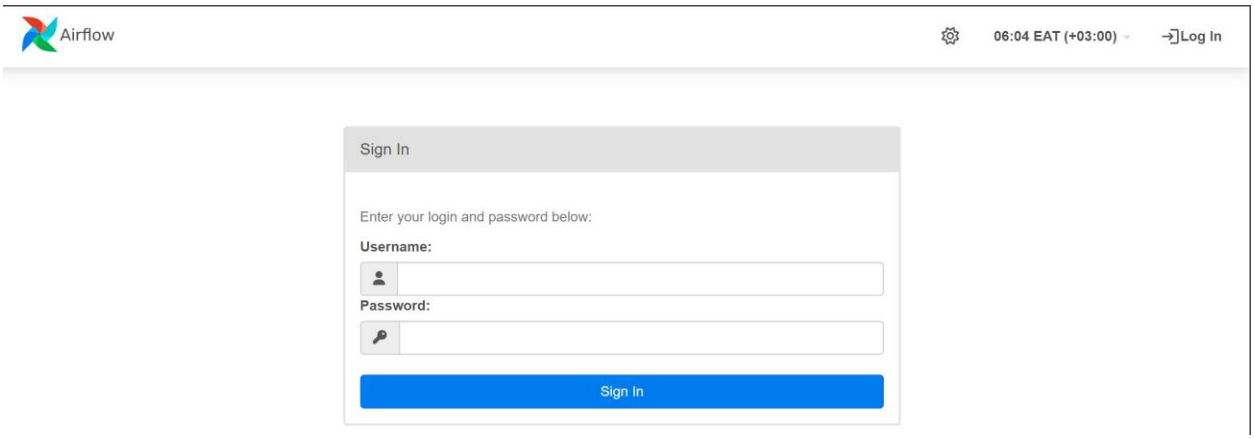
Fig 2.8: Airflow UI connection timeout



Uh-oh! The url times out. This issue happens cause of restrictions in the security group settings on my EC2. To resolve this error, I edited my inbound rules on the SG created when creating my EC2 instance to open the airflow port 8080.

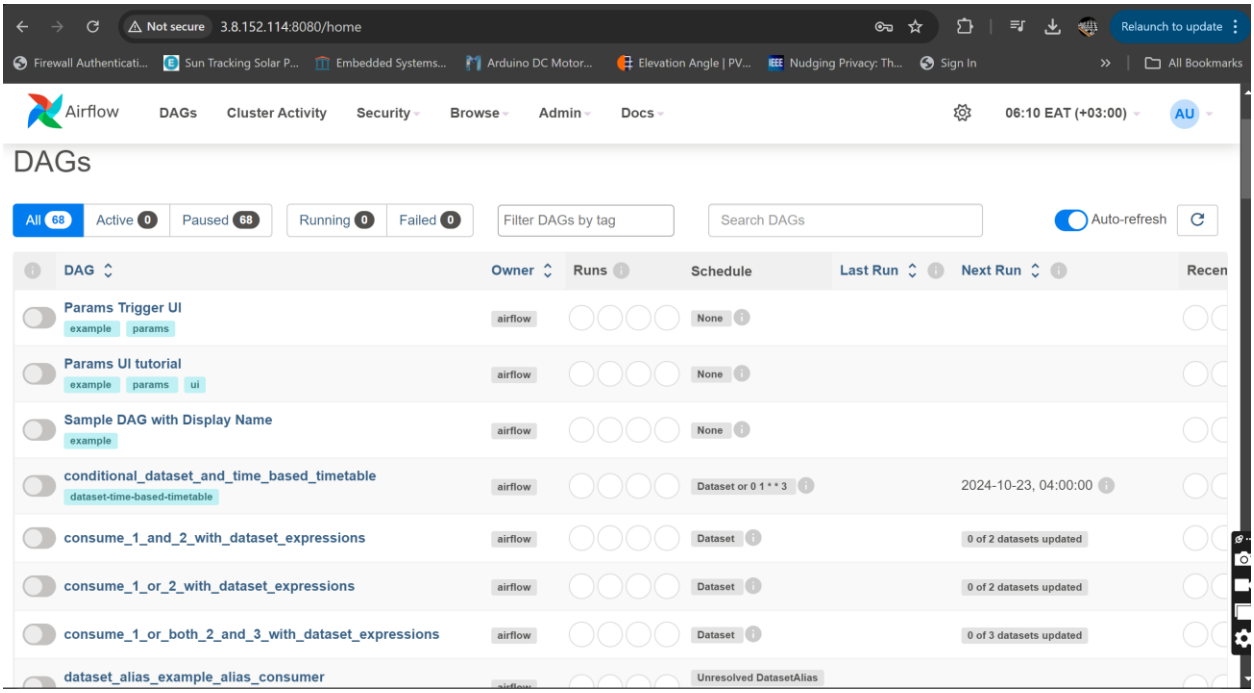
Figure 2.9: Successful landing page on the airflow UI

Resolving the SG issues fixes the problem



Using the credentials provided by the airflow server I logged in to the UI.

Figure 2.10: Successful UI login



Apache airflow comes with default DAG (direct acyclic graph) activities. Read more about DAGs in apache airflow under https://airflow.apache.org/

EC2 instance connect window now shows all the activities of the apache server. I will use this window to monitor the health of the airflow. I needed to still connect to my EC2 instance incase I need to pass commands to the server. Based on this understanding, I SSHed to my EC2 using my local IDE (VSCode).

SSH connection to EC2 instance

To connect to EC2 using VSCode and maintain the IDE properties on the server, I installed the remote SSH on VSCode and modified the config file to reflect the properties of my server.

Host [Public IPv4 DNS name]

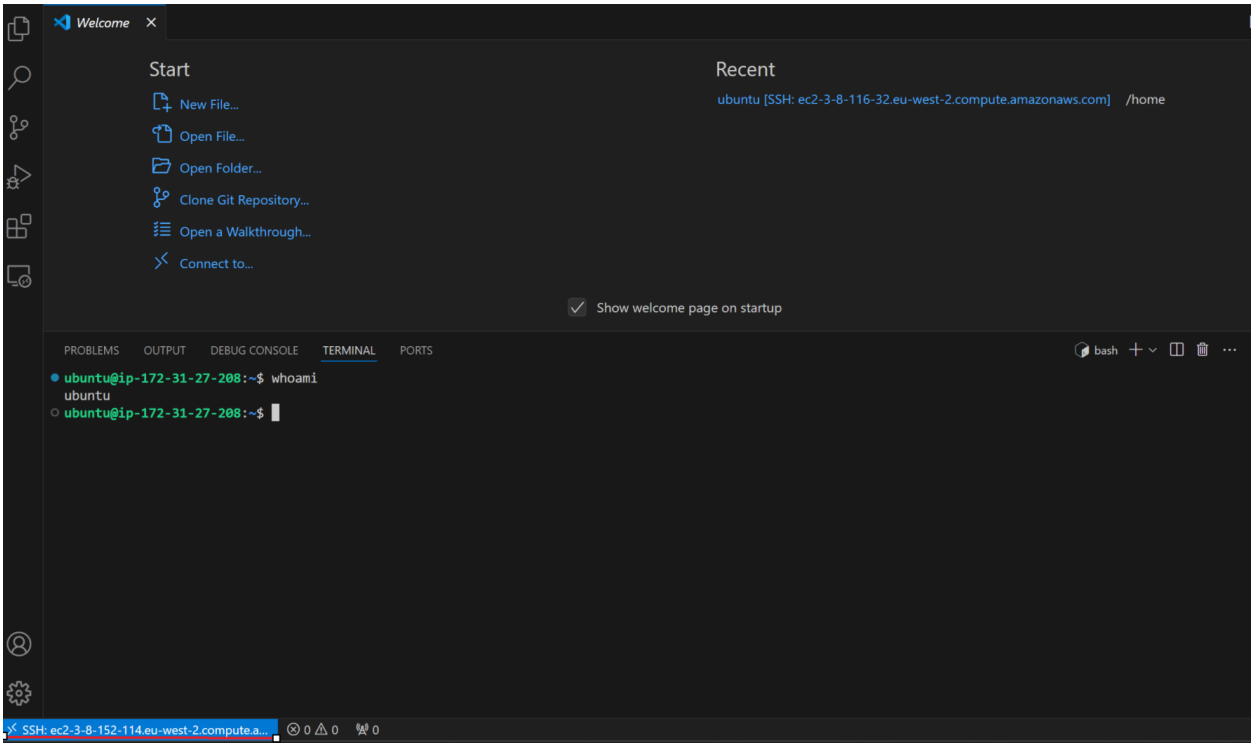
HostName [Public IPv4 DNS name]

IdentityFile [Absolute file path to the EC2 key pair]

User Ubuntu

Add a new host on VSCode and select the updated config file when prompted.

Figure 2.11: Successful remote connection established



Scripting code to extract data

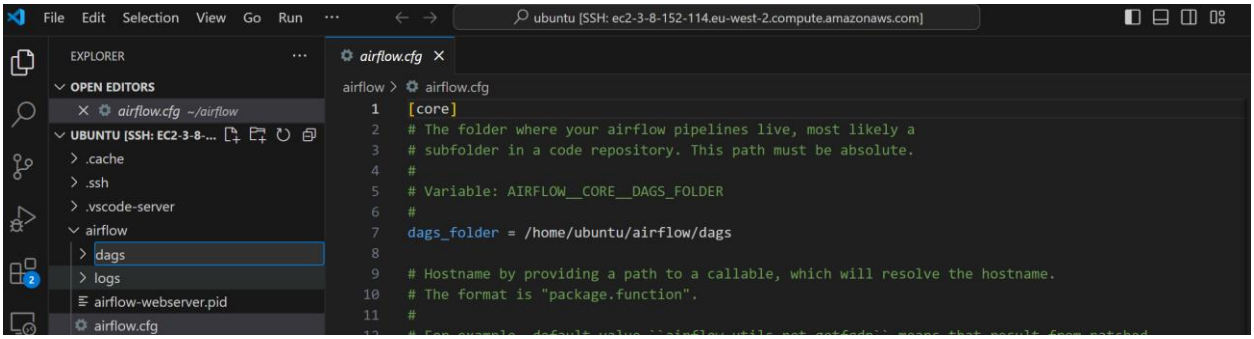
All the necessary infrastructure is now in place. Next step is to script code to extract data. I implemented this section using python code.

House- keeping rules on the airflow config file:-

Open the folders in the EC2 using VSCode and open the airflow config file. This is the file that holds all the configuration settings of the airflow.

Airflow requires that the extract script be stored in the dags_folder referenced by the absolute file path /home/ubuntu/airflow/dags. I proceeded to create a folder called dags in the airflow directory.

Fig 2.12: Dags folder creation



Under the dags folder create a file with the extension .py this is the file that will hold the extraction script.

Next, I went to rapidapi.com to get the api that I will use in this project. RapidAPI is a website that has consolidated multiple APIs for use, at a price of course. For this project I used the Zillow API, this API extracts data from Zillow. Zillow is a website that has real estate information about US states. You can customize which state you'd like.

RapidAPI also gives you a code snippet in the language of choice on how the API calls are to be implemented. I considered this in my script. Get my script under source extraction script

Code explained

First, the necessary libraries and modules are imported, including Airflow's ***DAG*** class to structure the workflow and ***PythonOperator*** and ***BashOperator*** to define specific tasks within the DAG. The ***requests*** and ***json*** libraries are included to handle the API call and data processing steps, while ***timedelta*** and ***datetime*** are used for time-based scheduling and naming.

The code begins by loading API configuration details from a JSON file named ***config_api.json***, stored locally on the server. This configuration file contains API-specific information, such as authentication headers, necessary for accessing the Zillow API. The ***datetime.now()*** function is called to capture the current date and time, formatted as a string. This timestamp is appended to filenames generated within the pipeline to ensure unique naming, avoiding overwriting or conflicts with previous runs.

The main data extraction function, ***extract_zillow_data***, is defined to perform the API call to Zillow. It accepts several keyword arguments, including the API URL, headers for authentication, query parameters specifying the data location, and the formatted date string. The function sends an HTTP ***GET*** request to the Zillow API using ***requests.get***, then parses the JSON response and saves it to a uniquely named JSON file in the designated directory. This file, along with an additional CSV filename string, is returned as a list for use by downstream tasks.

In setting up the Airflow ***DAG*** structure, default arguments are defined to specify the task owner, email notifications, retry policy, and other scheduling parameters. The DAG is configured with a daily schedule interval ***(`@daily`)*** and ***catchup=False*** to skip missed runs if any are encountered.

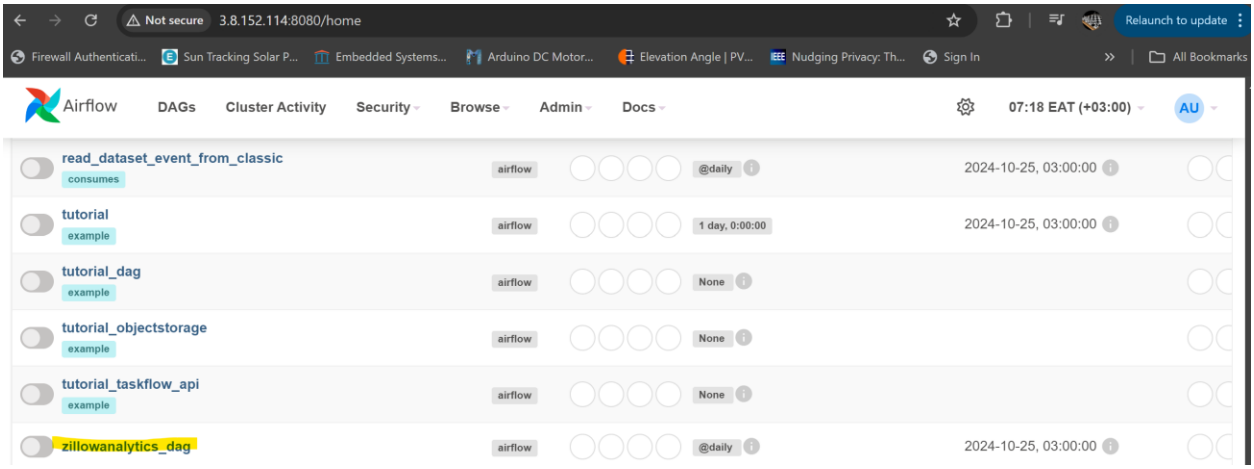
Within the DAG definition, two tasks are established. The first task, ***extract_zillow_data_var***, is a ***PythonOperator*** that triggers the ***extract_zillow_data*** function to fetch data from the Zillow API. Parameters such as the API URL, query parameters, headers, and date string are passed in using ***op_kwargs***. The second task, ***load_to_s3***, is a ***BashOperator*** responsible for transferring the generated JSON file from the local directory to an Amazon S3 bucket. It uses the ***aws s3 mv*** command along with Airflow's ***xcom_pull*** function to dynamically fetch the file path from the output of the ***extract_zillow_data_var*** task. This task moves the JSON file to the specified S3 bucket, facilitating secure and scalable storage for further processing or analysis.

Finally, the pipeline's task dependencies are established, ensuring that the data extraction task completes before the file transfer to S3 occurs.

Extraction

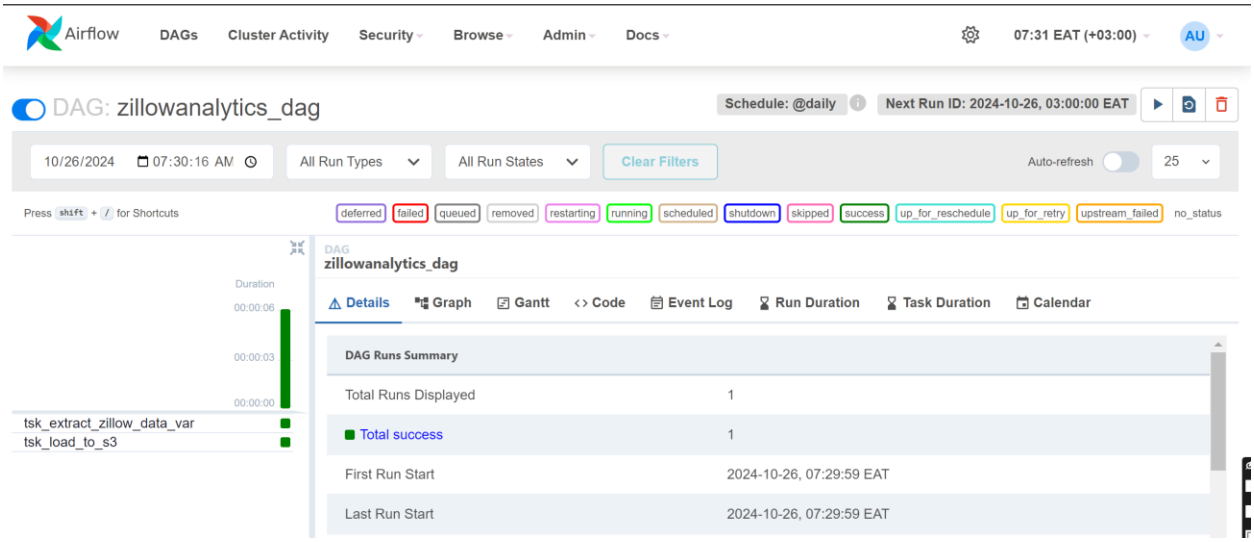
My DAG has been propagated to the airflow. See image below.

Figure 2.13: Successful DAG propagation



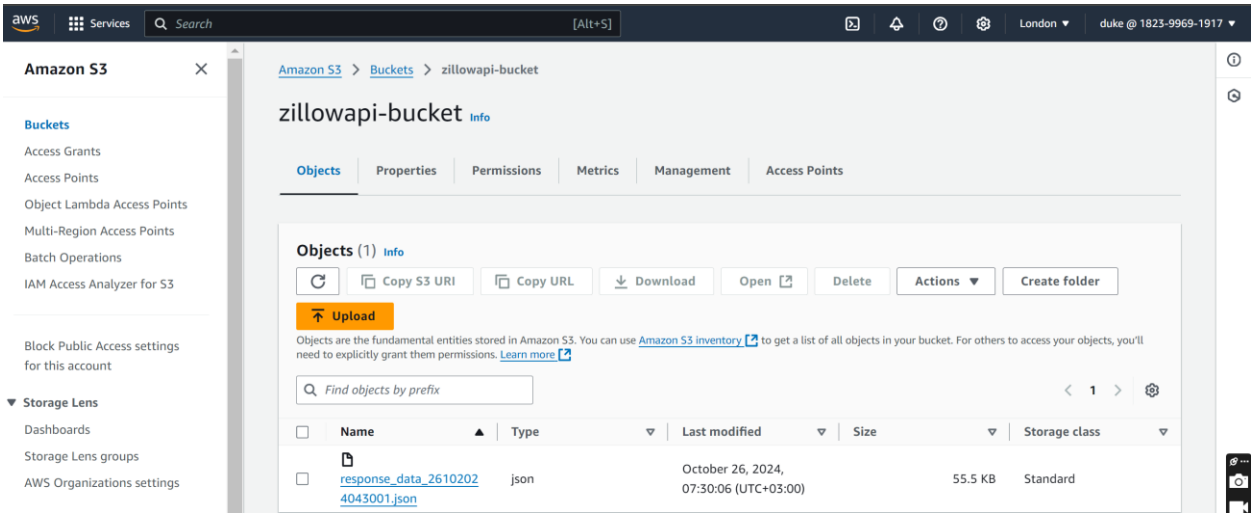
Manually trigger the DAG run and monitor the progress. However, I have also configured the run to trigger automatically once every day at midnight. This means any changes in the data will be propagated downstream.

Figure 2.14: Successful DAG run



I looked into my S3 bucket and noted that indeed there was data, this means that the raw data was successfully extracted and stored in an S3 bucket as a json file.

Figure 2.15: Contents of S3 bucket

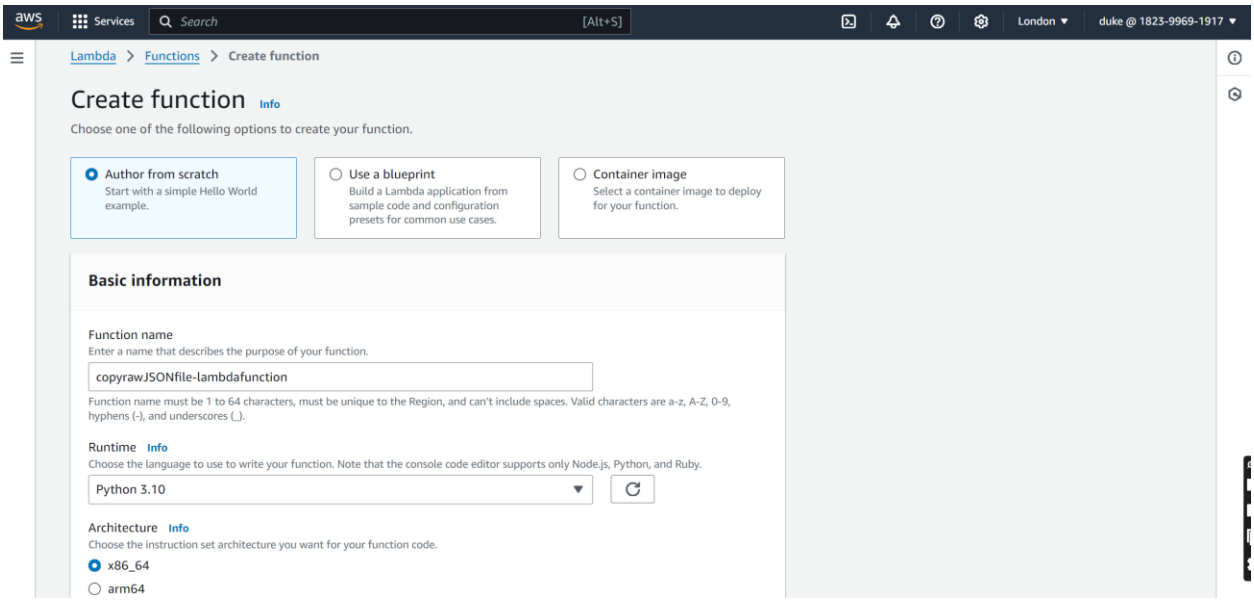


Copy raw json file to a staging area

Now that the raw json file has been ingested into the pipeline, the next step in the architecture diagram was to create a function that copies this raw data to a staging area. I achieved this using lambda functions. The reason for moving this data to a staging area is to ensure that the raw data is unadulterated in the event I need to reference it. This is a very good data practice.

Setting up a copy lambda function

Figure 2.16: Lambda function setup



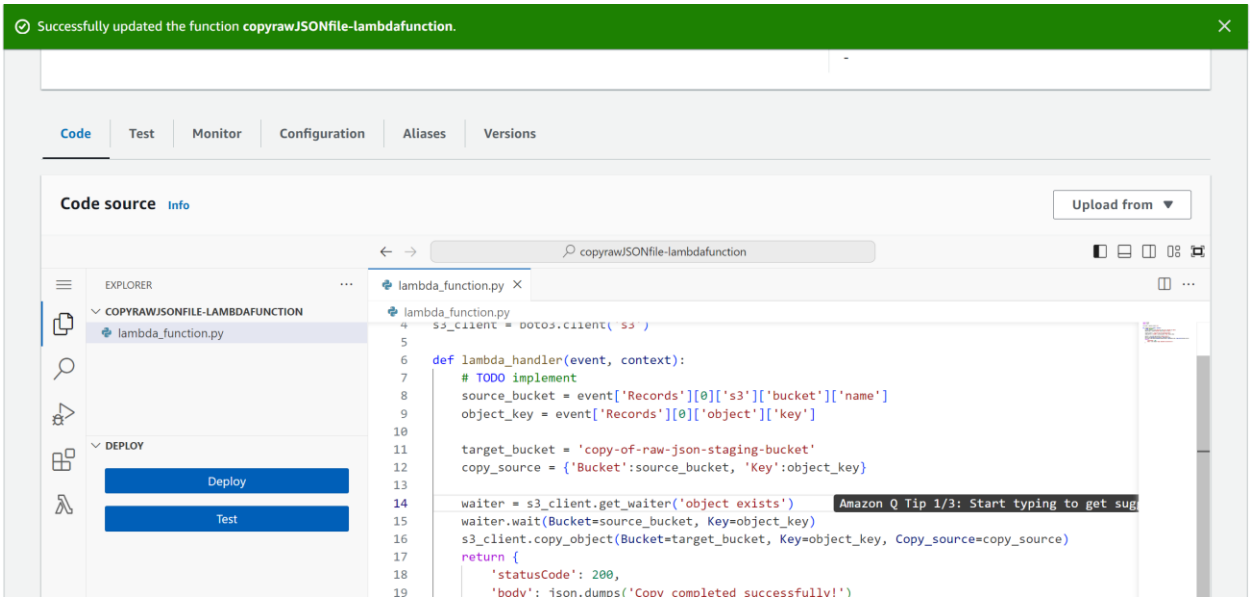
I assigned the following permissions to this role:-

- AmazonS3FullAccess - Provides Lambda functions with full access to all S3 buckets.
- AWSLambdaBasicExecuteRole – Allows Lambda functions to write to CloudWatch logs.

I then initialized an S3 trigger to the function. This trigger is such that if the S3 bucket is updated with new content, the lambda function is executed.

Once the lambda function is update, I deployed it to production.

Figure 2.17: Successful lambda deployment to production



Next step is to run the airflow again and check if any changes to the landing bucket at propagated downstream to the staging bucket.

Figure 2.18: New bucket object

New bucket object written at 9:50:46 AM.

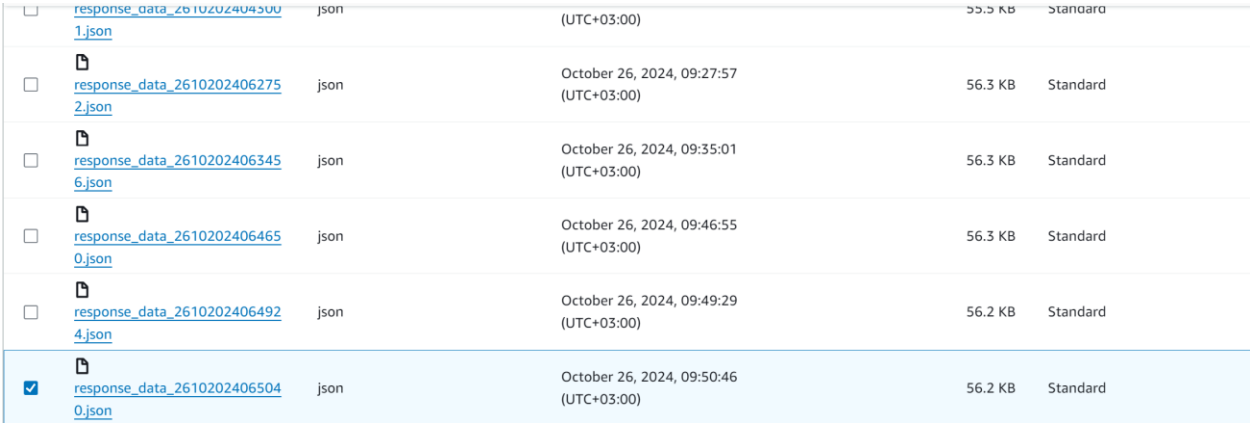
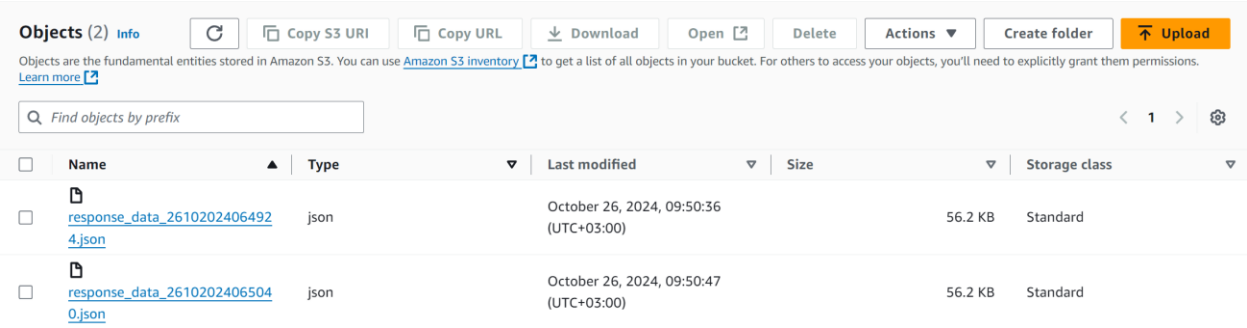


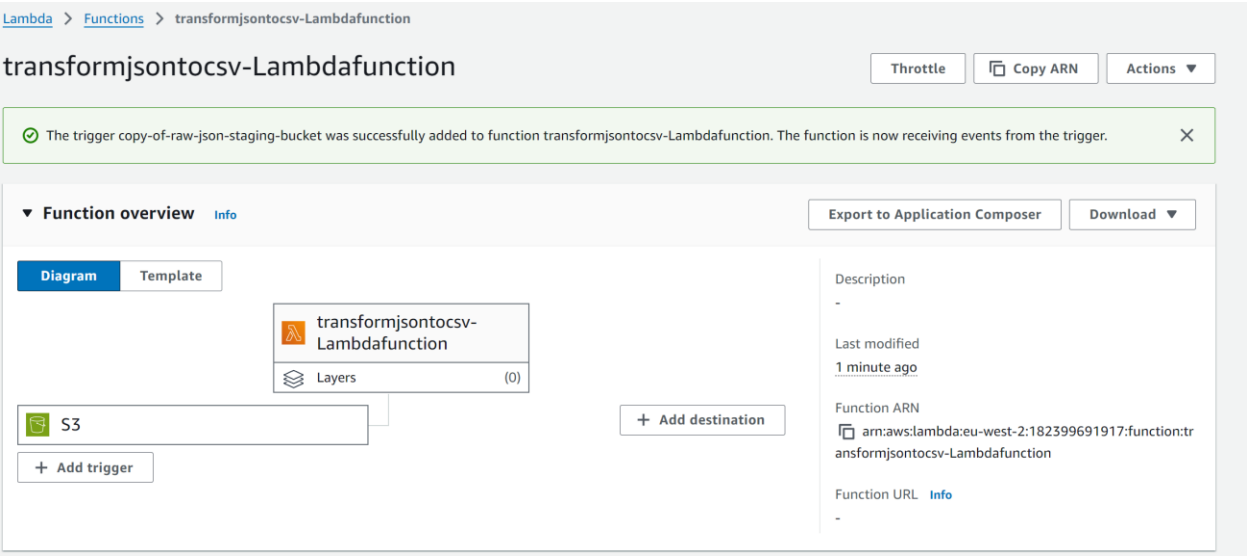
Figure 2.19: New bucket object copied to staging bucket

Copied bucket object written at 9:50:47 AM. This shows that the copy pipeline is working.



Setting up a transform lambda function

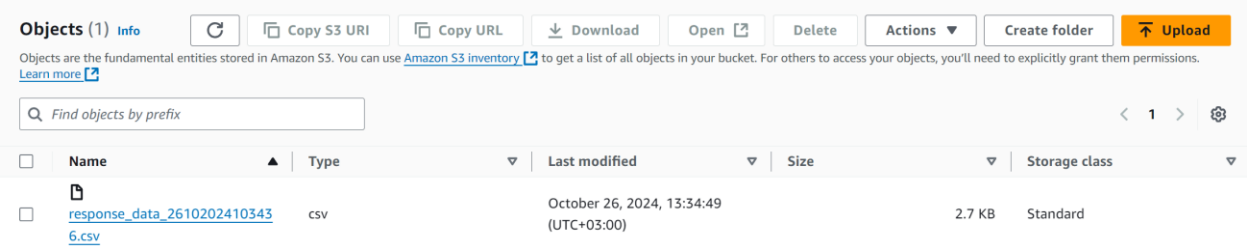
Figure 2.20: Transform lambda function



This function once triggered transforms the data in the staging bucket to a .csv file.

Next, I created a DAG activity called S3KeySensor. This activity monitors the contents of the final bucket and ensures that only when the final bucket has been loaded with data does is the data moved to Redshift. See additional DAG activity.

Figure 2.21: Successful transformation loaded in final bucket



Setting up a RedShift Cluster

Next, I set up a RedShift cluster. This datawarehouse is provisioned depending on the size of the data to be stored. For this project, I went with the dc2.large single node type cluster. After creating the cluster, I connected to the DW ready for query execution. See image below.

Figure 2.22: Successful connection to RedShift cluster

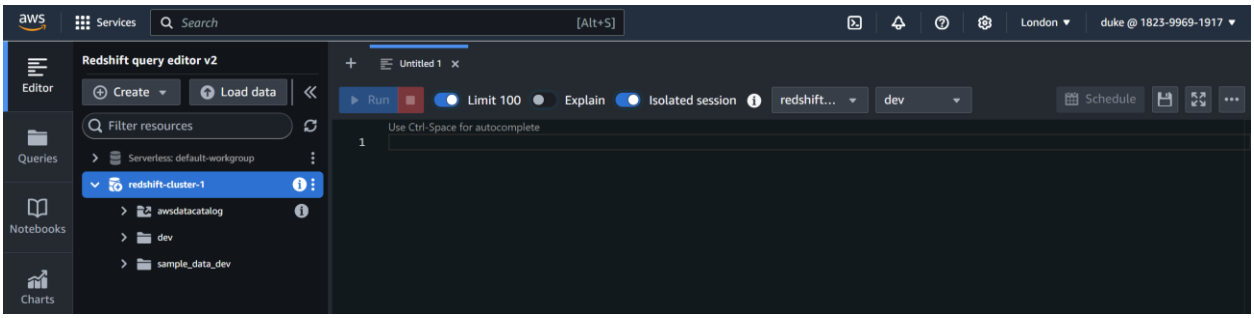
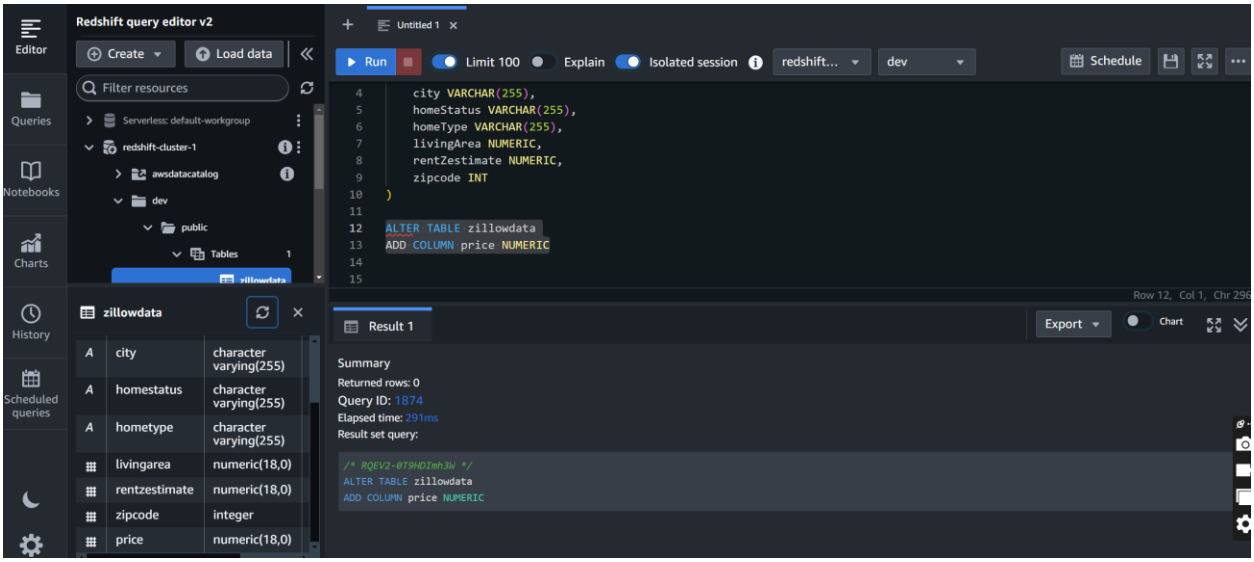


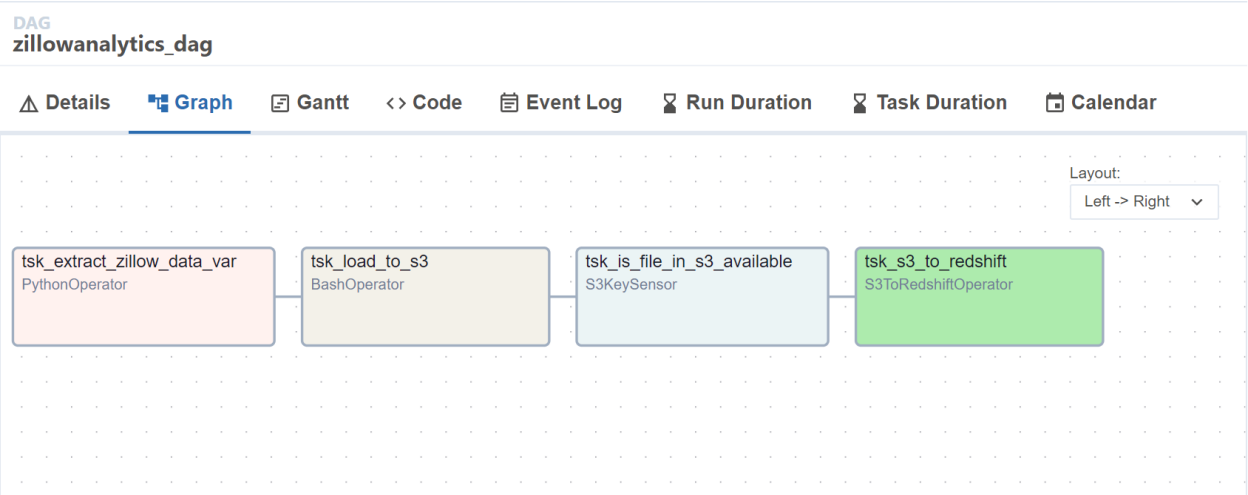
Figure: 2.23: Tables in the DW

Next I created a table within the cluster that will hold the data.



Finally, I created a DAG activity to copy data to the RedShift datawarehouse. See complete DAG profile below.

Figure 2.24: Complete profile of all DAG activities.



I triggered the DAG and the whole data was moved to the datawarehouse. The final step involved presenting this data using QuickSight. I have done a similar project on AWS Quicksight linked here: AWS Quicksights

Conclusion

In conclusion, the Zillow API ETL pipeline project successfully demonstrates an automated data extraction, transformation, and loading (ETL) workflow using Apache Airflow to orchestrate the process, and various AWS services to securely store and analyze real estate data. This project highlights efficient use of EC2 for computational tasks, Amazon S3 for data storage, Redshift for data warehousing, and Quicksight for data visualization, providing a robust and scalable solution for handling high-volume, dynamic data. Through careful design and integration of AWS services, this ETL pipeline offers a practical and efficient approach for analyzing real estate market trends and can be adapted for other similar data integration needs.

The source code for this project can be found in the GitHub link provided:- GitHub