# **ETL-EMR (HDFS) pipeline with Apache Airflow**

## **Introduction**

This project focuses on building an automated and scalable data pipeline to ingest, process, and visualize Big data. Leveraging Apache Airflow for orchestration, this pipeline reliably schedules and executes ETL processes, handling both structured and unstructured data. By constructing this pipeline, I explored how HDFS can efficiently store large volumes of structured and unstructured data and how Spark processes this data for analysis.

The data is processed through an Amazon EMR cluster that scales automatically based on demand, ensuring efficiency and resilience, with continuous health monitoring and auto-replacement for unhealthy nodes. After transformation, the data is stored in an Amazon S3 bucket, triggering a Snowpipe for seamless loading into Snowflake. The pipeline is then connected to Power BI, enabling real-time insights and trend forecasting driving data decisions.

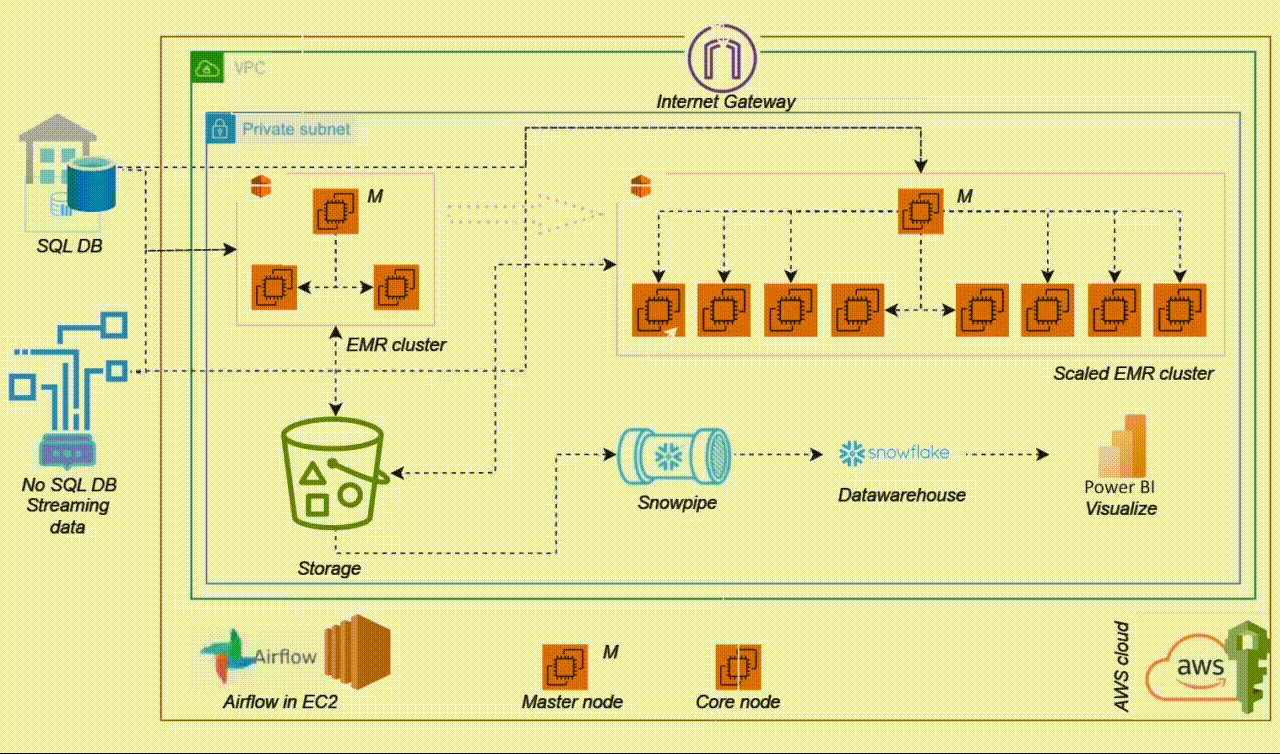
The key services utilized in this project include:

* **Amazon EC2** hosts Apache Airflow to manage and schedule the ETL workflows, providing reliable automation and orchestration.
* **Amazon S3** acted as the centralized storage for both raw, and transformed data, triggering Snowpipe to initiate loading into Snowflake. The S3 bucket also held the cluster logs and execution scripts.
* **Python** handles data extraction, enabling customized data processing from the data sources.
* **VPC** provided a secure, isolated network environment for the project's infrastructure, ensuring controlled access and data flow between services.
* **Snowpipe** automated the process of ingesting data from S3 into Snowflake, enabling near real-time updates to the data warehouse.
* **Snowflake** served as the data warehouse for centralized storage and querying, supporting downstream analytics and visualization.
* **Power BI** used to create visualizations and dashboards, offering insights and trend analysis on Redfin data.
* **Airflow** managed the ETL pipeline by scheduling and orchestrating tasks, ensuring reliable and automated data extraction, transformation, and loading.

### Architecture Diagram

In the *Architecture Diagram* section, the ETL pipeline starts with **Python** ingesting data being ingested into **Apache Airflow** (hosted on **EC2**), which orchestrates the workflow and schedules ETL tasks. From there, data flows to **Amazon EMR** for distributed processing using **HDFS** and **Spark**, where automatic scaling and node health monitoring are implemented to ensure efficiency and resilience. The processed data is then stored in **Amazon S3**, with an event trigger activating **Snowpipe** to load data into **Snowflake** for structured storage and querying. Finally, **Power BI** connects to **Snowflake** to visualize the data, enabling real-time insights and analytics. The **VPC** encompasses all resources to provide a secure, isolated environment, ensuring controlled data flows between components.

Figure 1.0: Architecture diagram.



## **Execution**

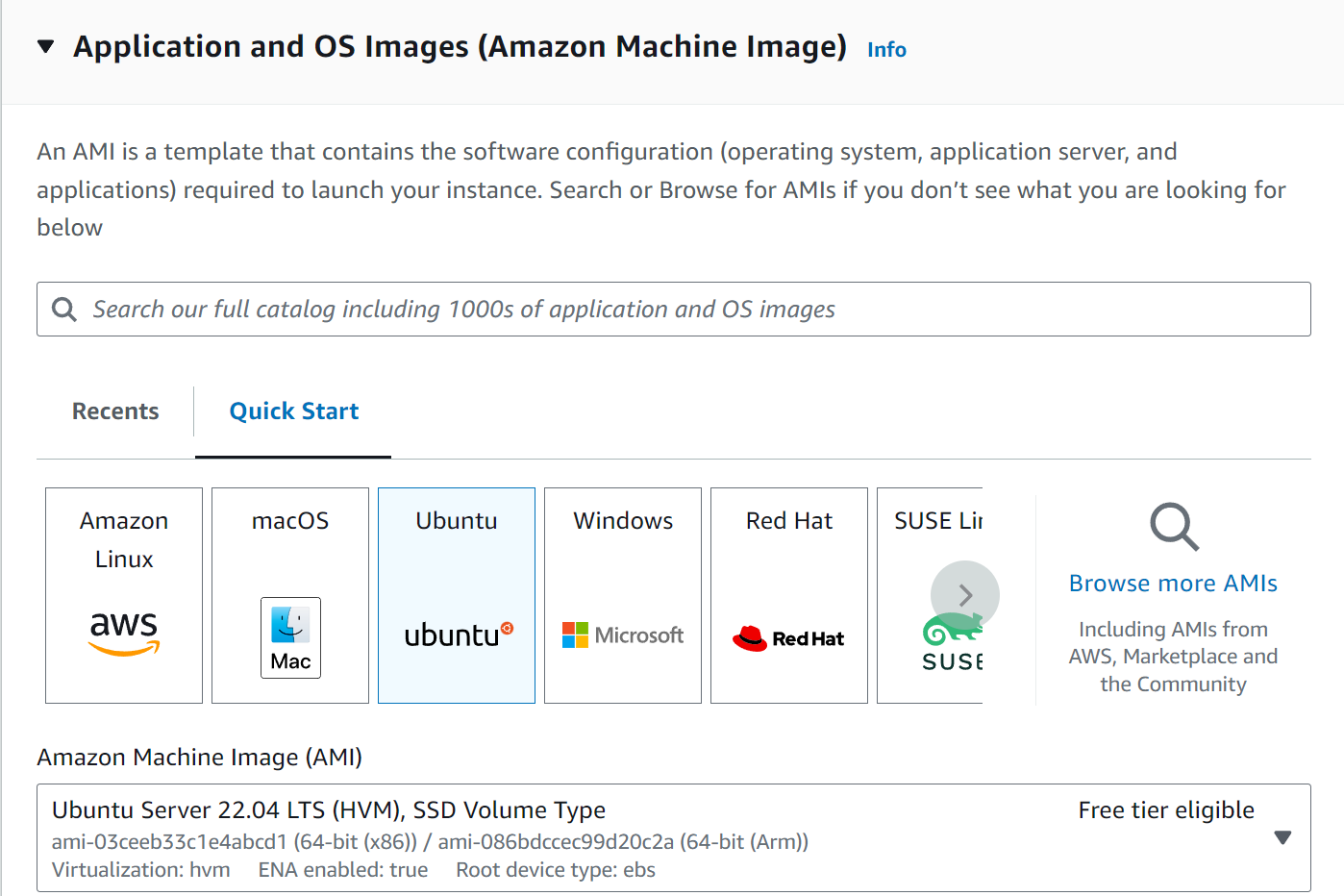
### Launching an EC2 instance

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.

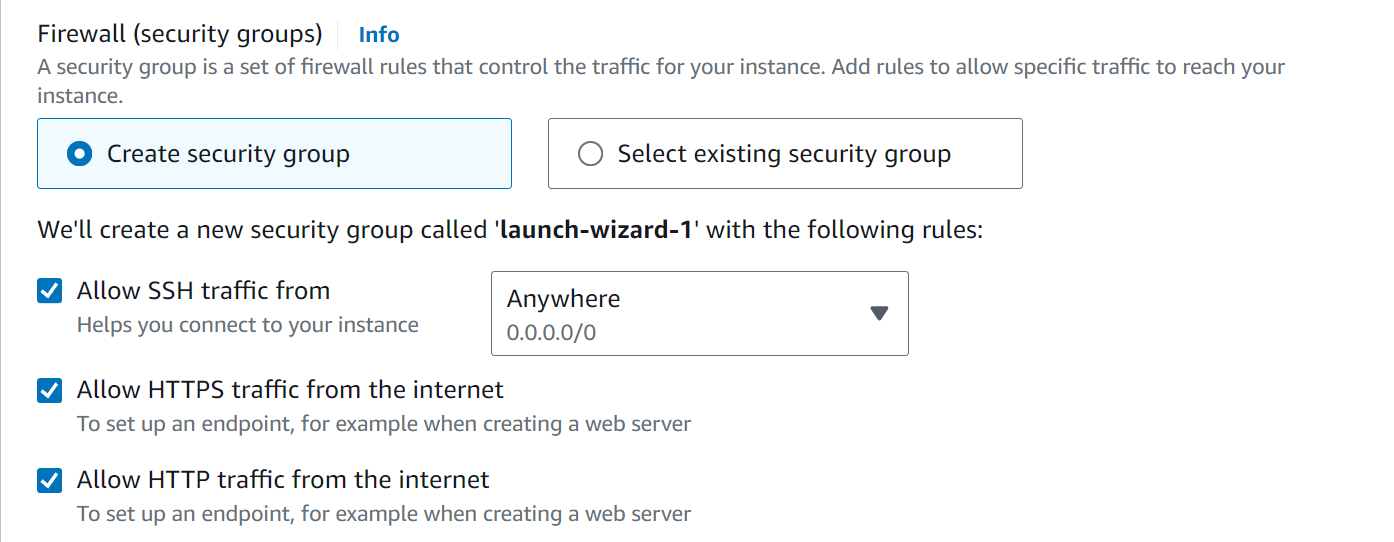
For this project I used the Ubuntu server 22.04 AMI which is free tier eligible. You can also choose to use the Ubuntu server 24.04 which is the latest Ubuntu AMI as at the time of this project.

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

### 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:

### 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.

### sudo apt install python3-pip – Installs python 3 on the server.

### sudo apt install python3.10.venv – Installs the python 3.10 virtual environment.

### python 3 -m venv [name of environment] – Creates a custom name for the venv.

### source [name of environment]/bin/activate – Activates the venv.

### pip install --upgrade awscli – Installs the AWS CLI.

### pip install boto3

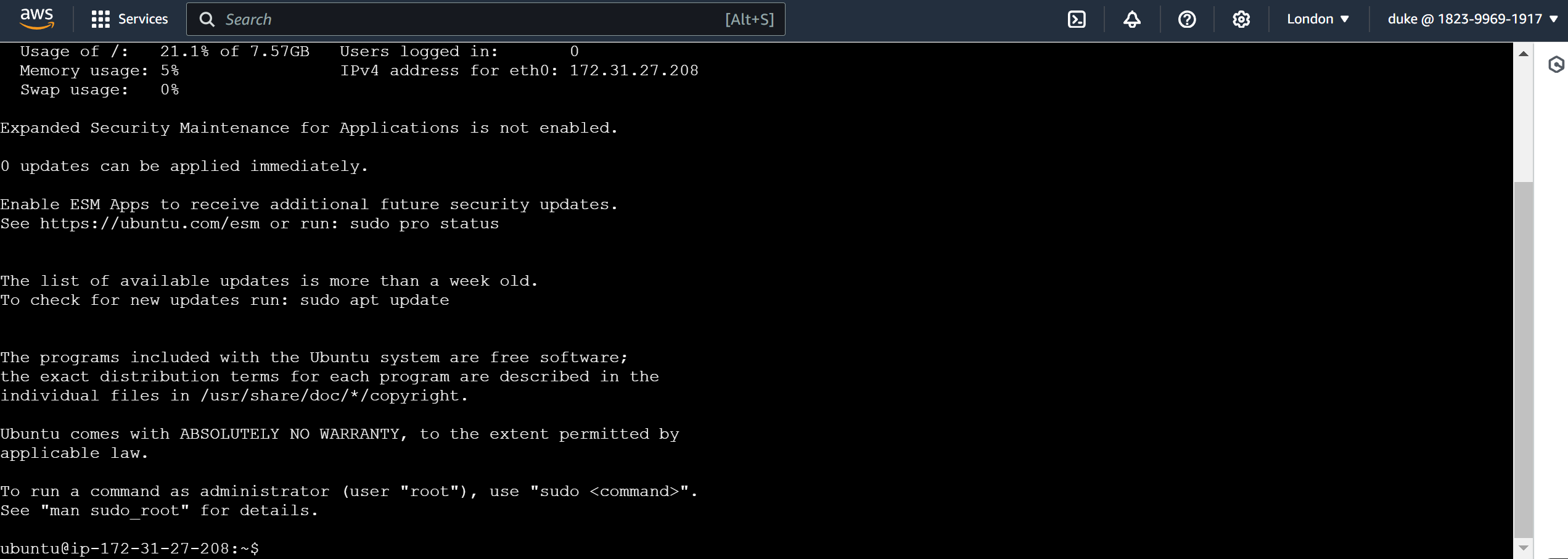
### aws configure

### pip install apache-airflow-provides-amazon

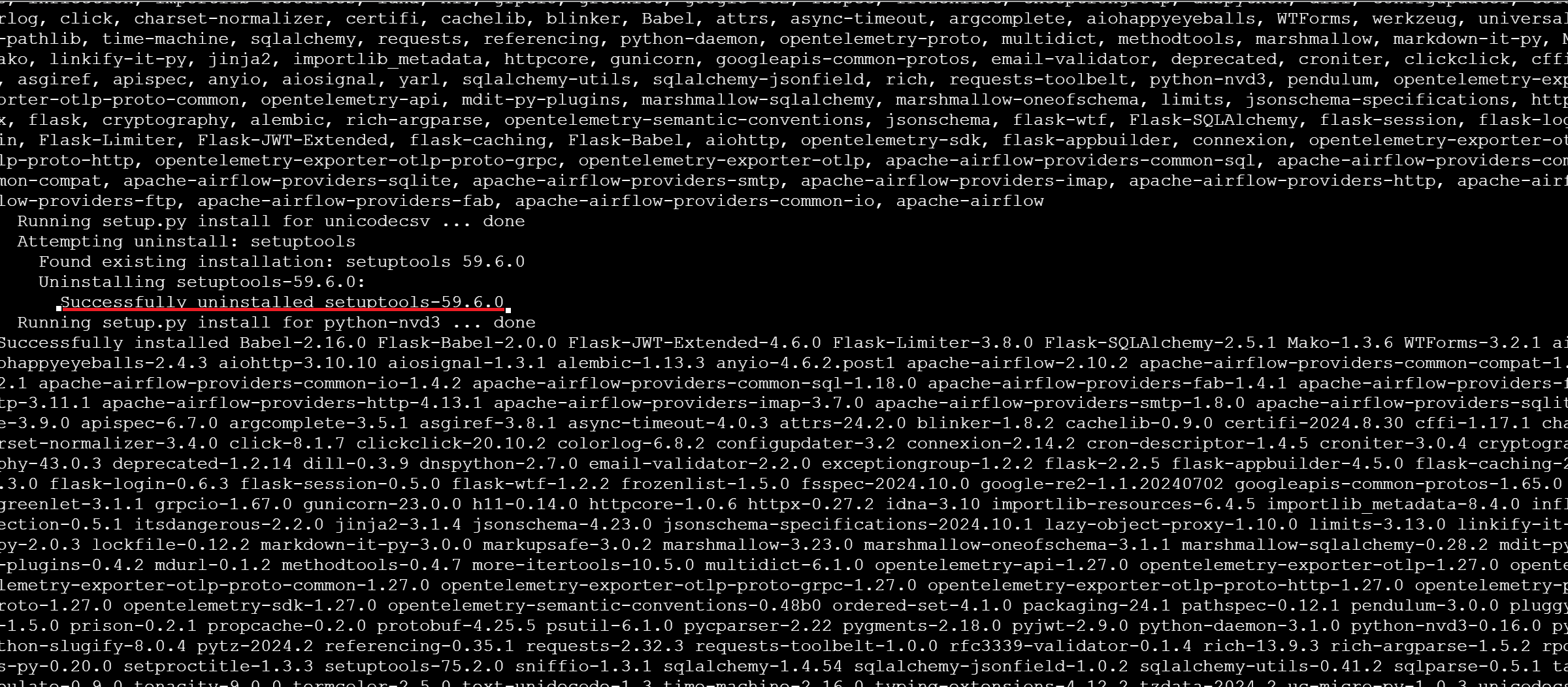
### pip install apache-airflow – Installs apache airflow service.

### Airflow standalone – Instantiates the airflow ready for DAGs.

Figure 2.3: EC2 instance connect



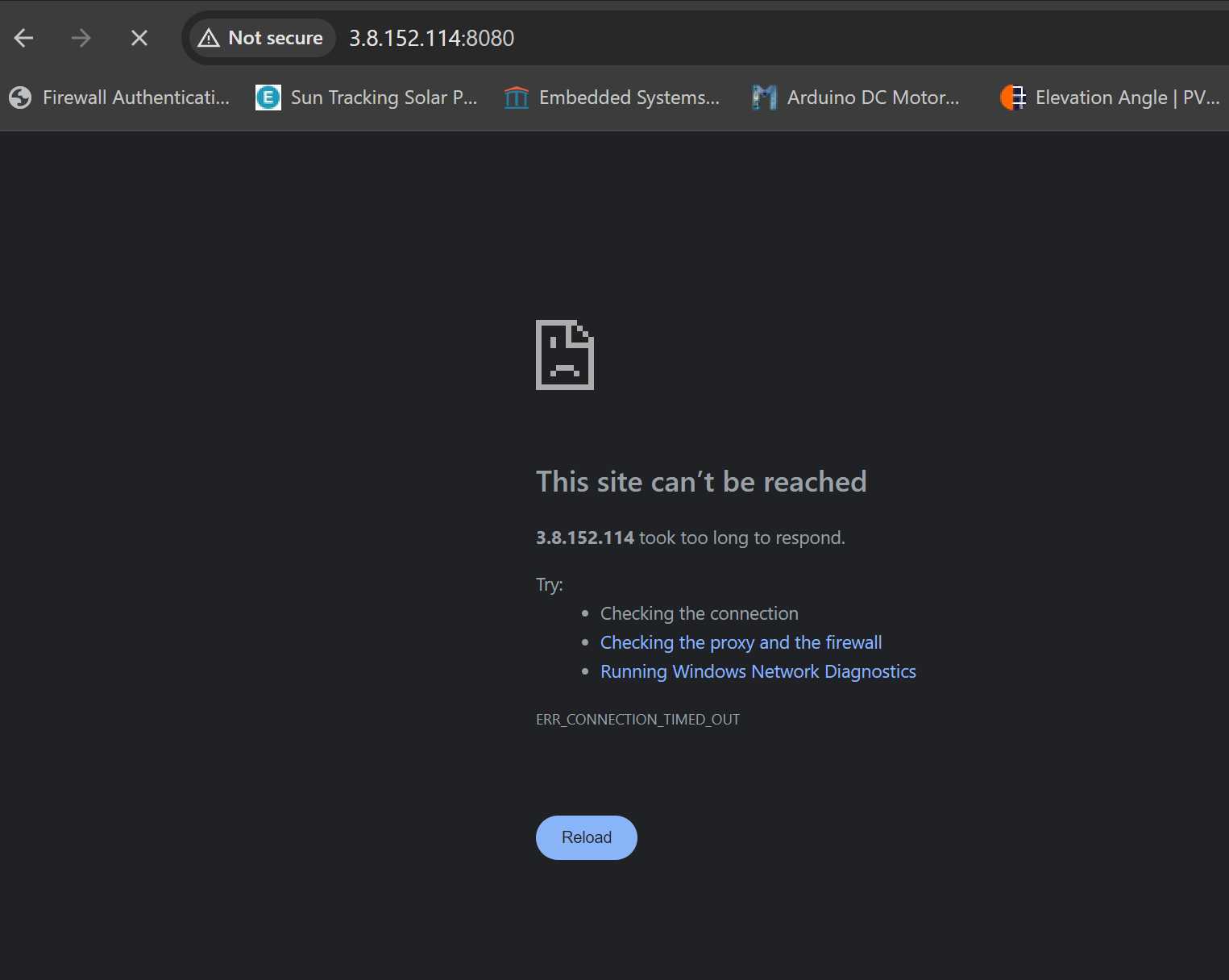
I needed to connect to my EC2 in two simultaneous sessions. One session is to monitor the performance of my airflow service and the other session to be able to pass commands to the instance if needed. This will clearly be understood when I install apache airflow on the server.

Figure 2.4: 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.

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.

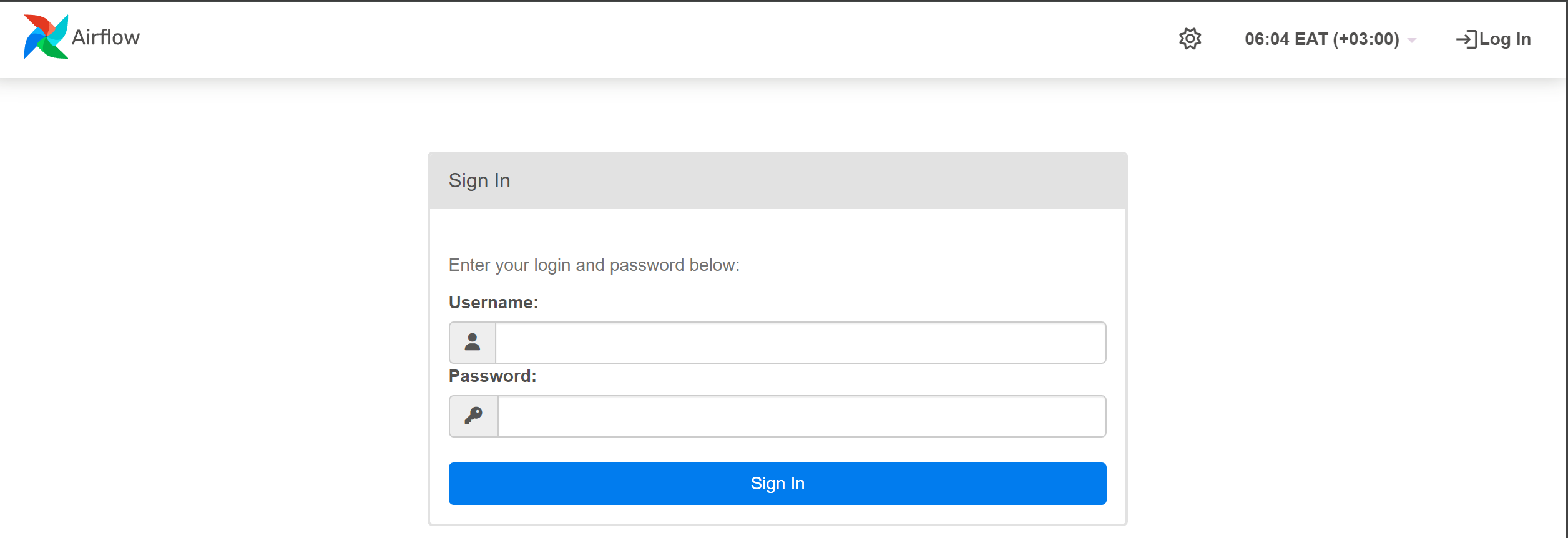
Fig 2.5: Airflow UI connection timeout



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 2.6: 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 2.7: 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]

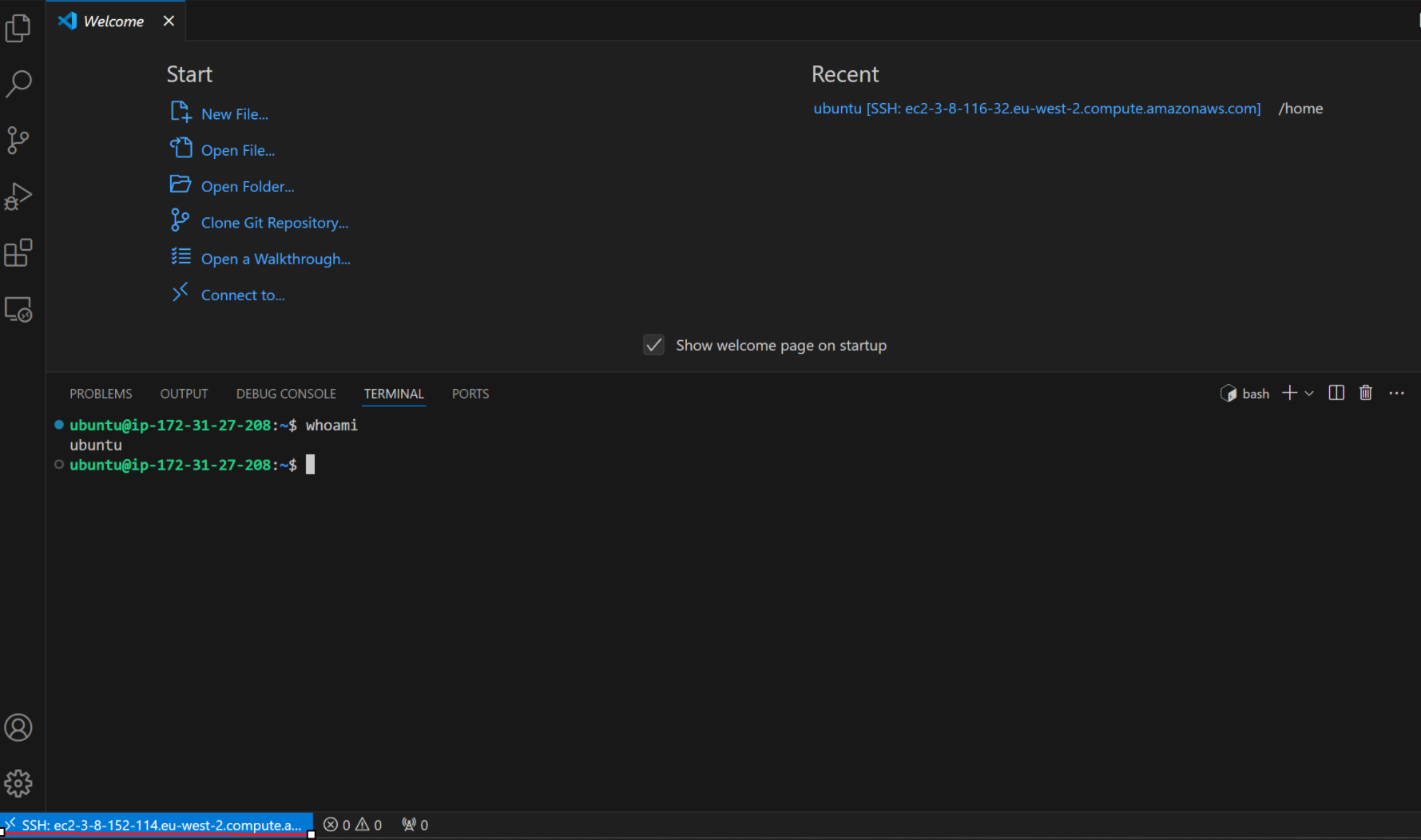
HostName [Public IPv4 DNS name]

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 2.8: Successful remote connection established



### Scripting pyspark code to extract and transform data

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

I then created an S3 bucket to store all my data. I partitioned the bucket into the following folder:

1. Scripts – This folder held the extraction and transformation scripts.
2. raw-data – Contains the ingested raw data.
3. transformed-data – Contains the transformed data.

Figure 2.9: S3 buckets

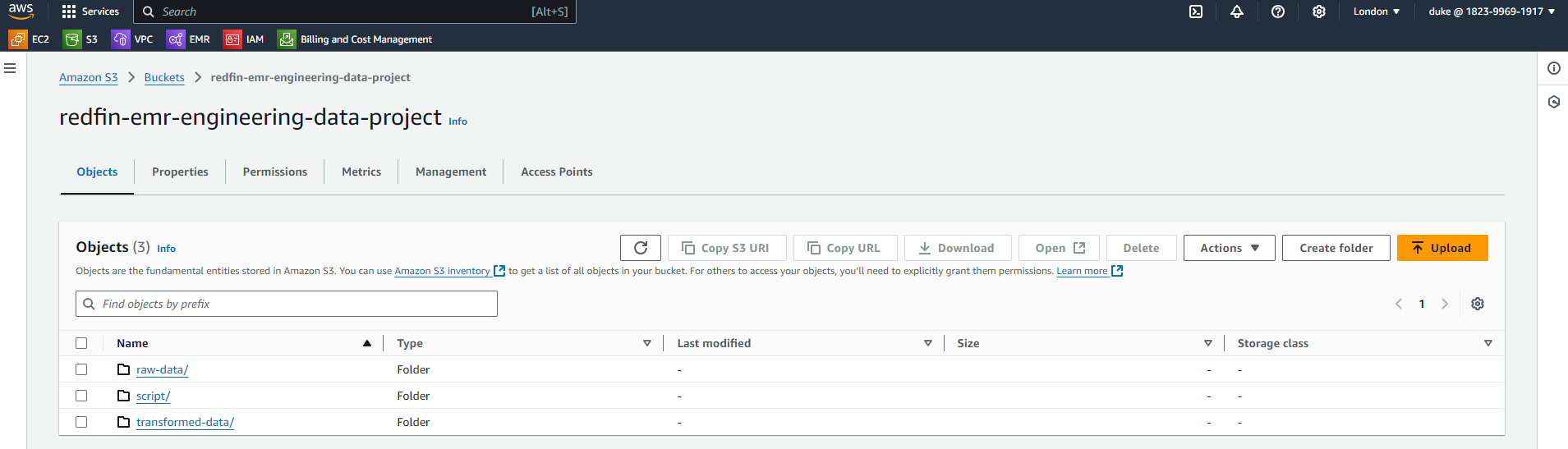
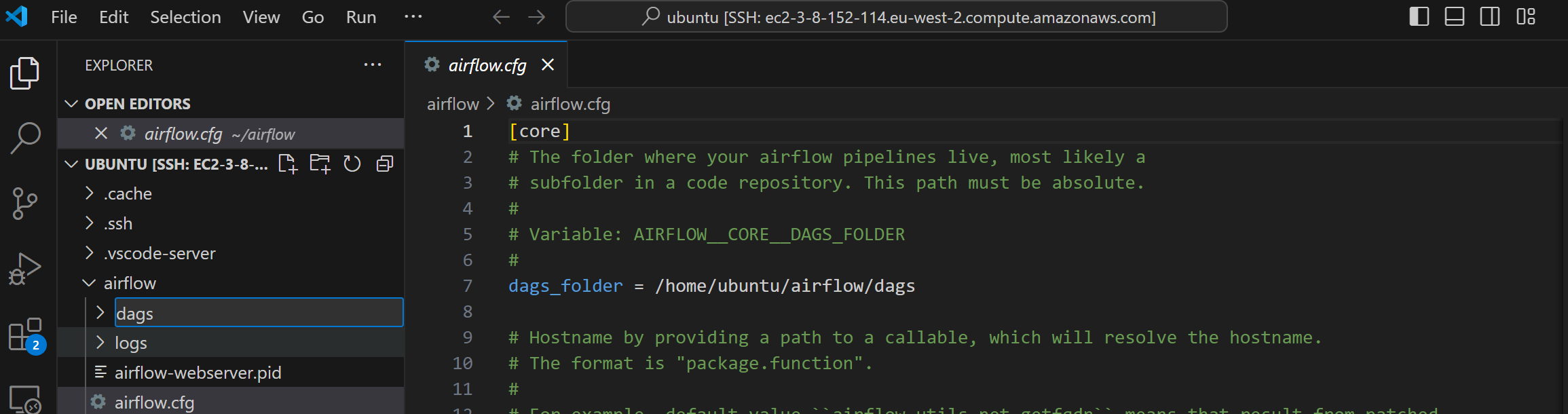


Fig 2.10: Dags folder creation



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

**Code explained**

This Apache Airflow DAG is designed to automate the workflow of provisioning an EMR cluster, performing data extraction and transformation using Spark, and then terminating the cluster once the tasks are complete. The workflow begins with a ***DummyOperator***, `start\_pipeline`, to signify the start of the process. The first significant task is the ***EmrCreateJobFlowOperator***, which creates an EMR cluster based on the `job\_flow\_overrides` configuration. This cluster is set up with specific instance types and roles, such as a master and core node, and includes necessary applications like Spark and Jupyter Enterprise Gateway. The cluster will also log activity to an S3 bucket.

Once the cluster creation is triggered, the ***EmrJobFlowSensor*** is used to monitor the status of the cluster creation. It checks the state of the cluster, waiting until the cluster is in the "WAITING" state before proceeding. After the cluster is ready, the ***EmrAddStepsOperator*** triggers the first EMR step, which is a data extraction process using a custom shell script hosted in an S3 bucket. The ***EmrStepSensor*** monitors the status of this extraction step, ensuring it completes successfully before moving on to the next step.

Once extraction is complete, the workflow proceeds to the data transformation step using Spark. The ***EmrAddStepsOperator*** again adds a new step that submits a Spark job to transform the extracted data, running a Python script also hosted in S3. The transformation process is monitored by another ***EmrStepSensor***, which ensures the job completes successfully before proceeding to terminate the cluster.

Finally, the cluster is terminated using the ***EmrTerminateJobFlowOperator***, which ensures that the EMR cluster is shut down after all the processing is complete. The ***EmrJobFlowSensor*** again monitors the cluster's state, checking for the "TERMINATED" status to confirm that the cluster has been properly terminated. The workflow ends with a final ***DummyOperator***, `end\_pipeline`, indicating the conclusion of the pipeline.

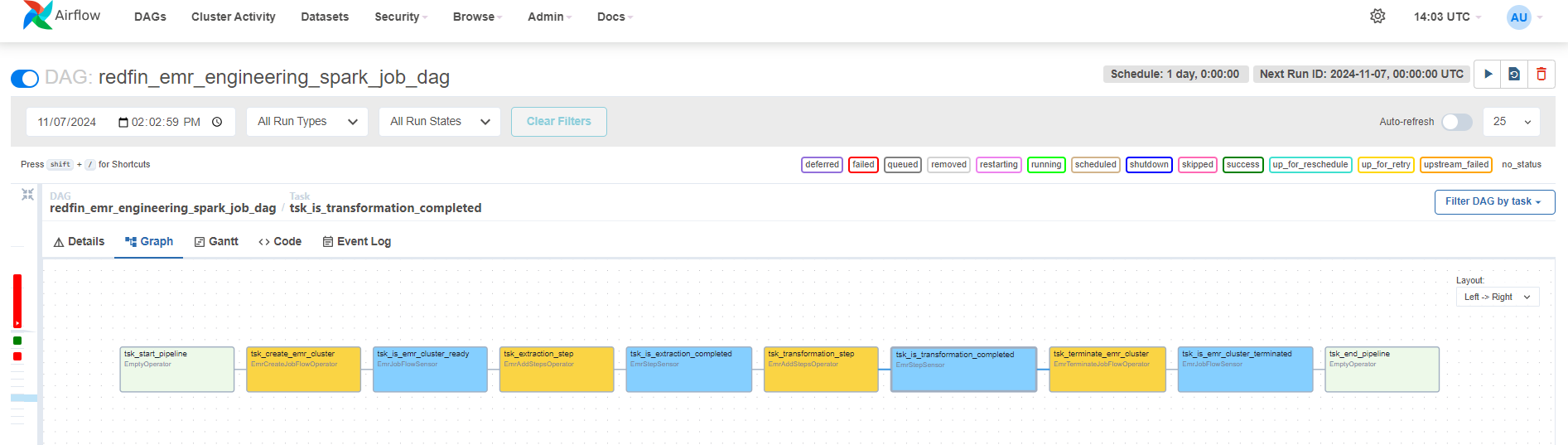
The entire DAG is designed to execute these tasks in sequence, with each task depending on the successful completion of the previous task. The code handles retries and includes error handling to ensure robust execution of the pipeline.

**Extraction and transformation in EMR cluster**

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.

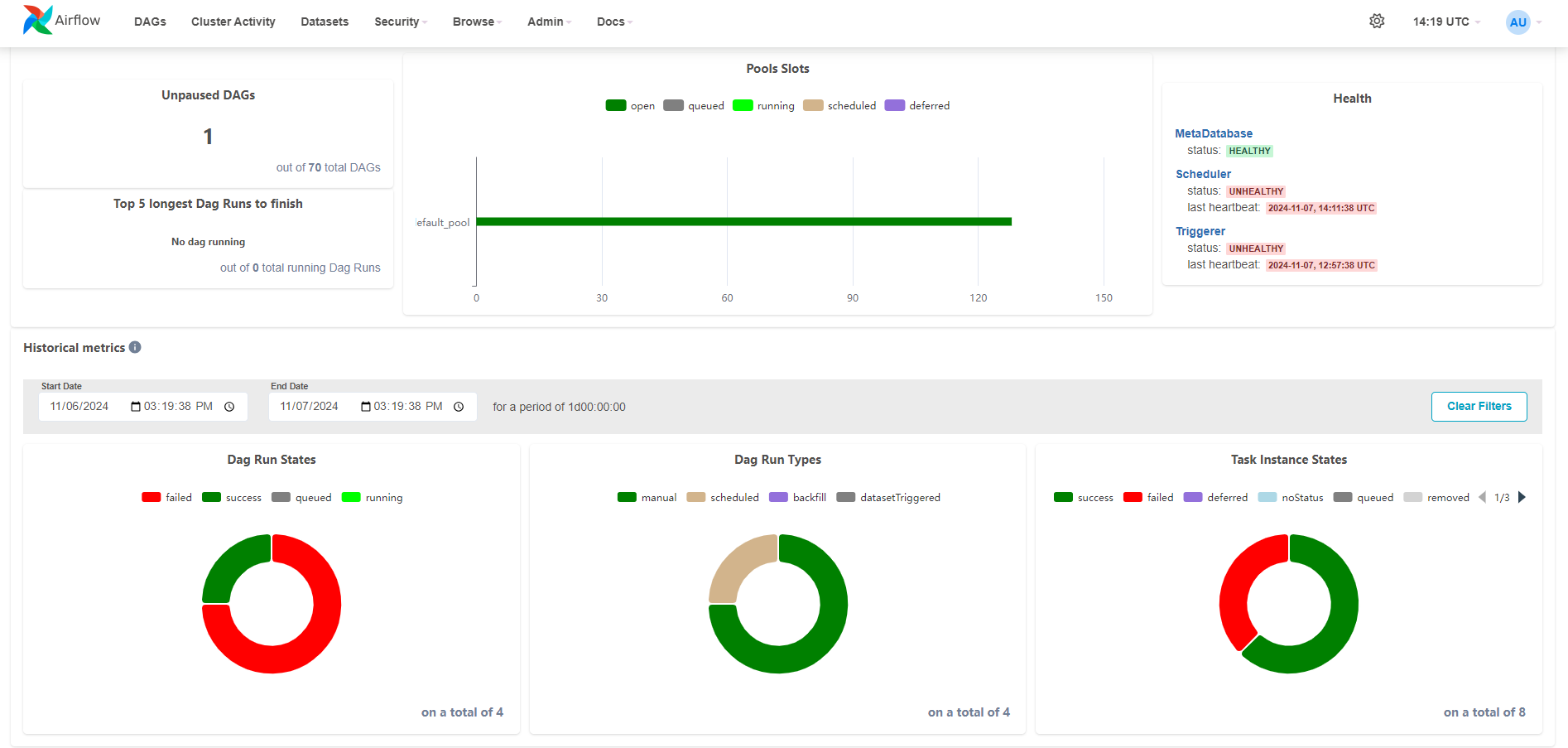
See the complete DAG in image below:

Fig 2.11: Complete DAG



I also monitored the DAG health in the airflow UI. See image below.

Fig 2.12: DAG health



## **Conclusion**

In conclusion, the ETL-EMR project provided valuable hands-on experience with big data frameworks, particularly HDFS and Spark. By utilizing HDFS for distributed storage, the project was able to efficiently handle vast amounts of structured and unstructured data, ensuring high availability and fault tolerance across a large cluster. Spark was employed for data processing, leveraging its powerful distributed computing capabilities to transform and analyze the data at scale. The use of Amazon EMR facilitated the deployment and management of these frameworks in a cloud environment, allowing for automatic scaling based on processing demand. This project not only deepened my understanding of the intricacies of big data frameworks but also highlighted their ability to provide scalable, efficient solutions for processing large datasets in real-time.

The documentation of airflow implementation in this project shallow. To read more on a project I documented well on how you can automate ETL jobs using Airflow click on link alongside: [Zillow rapid API ETL job with apache airflow](https://github.com/Dnurrein/Zillow-rapidapi-ETL-pipeline-with-AWS/blob/main/README.md)