



Alexis Balayre

Cloud Computing Assignment

School of Aerospace, Transport and Manufacturing
Computational Software of Techniques Engineering

MSc
Academic Year: 2023 - 2024

Supervisor: Dr Stuart Barnes
2nd January 2024

Table of Contents

Table of Contents	ii
List of Figures	iv
List of Tables	v
1 Introduction	1
2 Methodologies	2
2.1 Data Collecting, Processing & Storing	2
2.1.1 Overview of the Pipeline Architecture	2
2.1.2 Data Collecting	3
2.1.2.1 Data Source	3
2.1.2.2 Ingestion Script	3
2.1.3 Data Processing	3
2.1.3.1 Processing Script	3
2.1.3.2 Leveraging Apache Spark	4
2.1.4 Data Storing	4
2.1.4.1 Database Choice	4
2.1.4.2 Storing Script	5
2.1.5 Pipeline Implementation on AWS	6
2.2 Data Distributing	7
2.2.1 Overview of the Pipeline Architecture	7
2.2.2 Auto Scaling Group Configuration	9
2.2.2.1 AMI Configuration	10
2.2.2.2 Launch Template Configuration	11
2.2.2.3 Auto Scaling Group Configuration	12
2.2.2.4 Target Group Configuration	13
2.2.2.5 Load Balancer Configuration	14
2.3 Pipeline of the Project	15
3 Results & Discussion	16
3.1 Results	16
3.1.1 Accessing the Data - Grafana Dashboard	16
3.1.2 Load Test	21
3.2 Data Security and Sovereignty Considerations	22
3.2.1 Data Security	22

3.2.2	Data Sovereignty	22
3.2.3	Specific Fields for Further Examination	22
4	Conclusion	23
	References	24
A	Documentation	25
A.A	Project tree	25
A.B	Getting Started	25
A.C	Detailed Features of Functions	26
B	Source Codes	27
B.A	Ingestion, Processing & Storing Pipeline Source Code	27
B.B	Data Collecting Source Code	29
B.C	Data Processing Source Code	30
B.D	Data Storing Source Code	33
B.E	Scripts & Services Source Codes	37
B.E.1	Scripts	37
B.E.2	Services	38
B.E.2.1	Get IAM Credentials Service	38
B.E.2.2	Spark Python Job Service	38
B.E.2.3	Grafana Server Service	39
B.F	Load Test Source Codes	40
B.F.1	Artillery Load Test Configuration File	40
B.F.2	Instances Monitoring Script	41

List of Figures

2.1	Data Collecting, Processing & Storing Pipeline Diagram	2
2.2	EC2 Instance Configuration Screenshot	6
2.3	Data Distributing Pipeline Diagram	8
2.4	Auto Scaling Group Pipeline Diagram	9
2.5	AMI Settings Screenshot	10
2.6	Launch Template Settings Screenshot	11
2.7	Auto Scaling Group Settings Screenshot	12
2.8	Scaling Policy Settings Screenshot	13
2.9	Target Group Settings Screenshot	13
2.10	Load Balancer Settings Screenshot	14
2.11	Data Distributing Pipeline Diagram	15
3.1	Grafana Dashboard: Measure Value Evolution, Max & Mean AQI	17
3.2	Grafana Dashboard: AQI Map & IoT Data	17
3.3	Grafana Dashboard AQI Map Panel	18
3.4	Grafana Dashboard IoT Data Panel - Time Range Choice	18
3.5	Grafana Dashboard IoT Data Panel - Measure Value Choice	19
3.6	Grafana Dashboard IoT Data Panel - Apply a Filter	19
3.7	Grafana Dashboard IoT Data Panel - Export Values	20
3.8	Load Test Result	21

List of Tables

2.1 UK air quality index, “Review of the UK Air Quality Index”, 2011	4
--	---

Chapter 1

Introduction

This report investigates the integration of cloud computing and Internet of Things (IoT) technologies for real-time environmental monitoring, with a particular focus on air quality. The main objective of the project is to develop a robust system capable of efficiently managing large volumes of data from IoT environmental sensors and accurately calculating the Air Quality Index (AQI) in real time. This integration is essential for stakeholders such as researchers, policy makers and the public, who need rapid access to environmental data to make informed decisions.

The report describes the methodology, system architecture and implementation strategy adopted to ensure rapid data processing and distribution. Challenges encountered and corresponding solutions are also discussed, as well as the wider implications of the system for public health, urban planning and environmental awareness. This study demonstrates the practical application and benefits of combining cloud computing and IoT in the management and analysis of environmental data.

Chapter 2

Methodologies

2.1 Data Collecting, Processing & Storing

2.1.1 Overview of the Pipeline Architecture

In the **Data Collecting** phase, the latest dataset is obtained from its source, marking the commencement of the data pipeline. Following this, the **Data Processing** phase takes over, where the data undergoes formatting and calculation of the Air Quality Index (AQI) for each particulate matter sensor. The final phase, **Data Storing**, involves meticulous storage of data from each sensor in a time-series database, ensuring its integrity and accessibility for future use.

For a detailed visual representation of this pipeline, refer to Figure 2.1, which illustrates the Data Collecting, Processing, and Storing Pipeline.

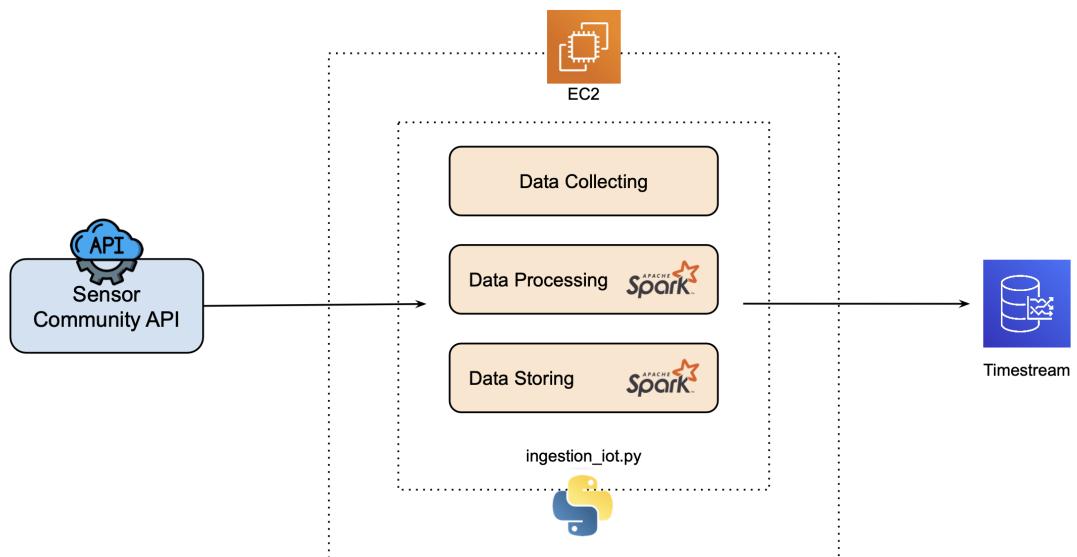


Figure 2.1: Data Collecting, Processing & Storing Pipeline Diagram

2.1.2 Data Collecting

2.1.2.1 Data Source

The Sensor Community network is a global, contributor-driven initiative that collects open environmental data through a vast network of sensors. These sensors, deployed in over 70 countries, collect real-time data on air quality, temperature, humidity and pressure. On average, the sensors send new data every 145 seconds (1). The Sensor Community network offers two main API endpoints for accessing their environmental data:

1. **5-Minute Averaged Data API:** This API provides data averaged over the last 5 minutes for each sensor. This is useful for near real-time analysis or immediate air quality assessments, particularly in test or active monitoring contexts.
2. **24 Hour Averaged Data API:** This API provides data averaged over the last 24 hours for each sensor. It is particularly suited to analysing daily trends and understanding environmental changes over a longer period.

2.1.2.2 Ingestion Script

The data acquisition process in this project involved leveraging both APIs from the Sensor Community. To do this, the Python Requests library was used, enabling efficient data retrieval. Once collected, the data is temporarily stored in a local cache, formatted as a Spark DataFrame for optimised handling and processing. This operation, executed by the `fetch_sensors_data` function, is located within the `collecting.py` script (see Appendix B.B for detailed code). In particular, this function is programmed to run at a constant interval of every 10 seconds, ensuring a regular and up-to-date flow of data from the sensors. This systematic approach not only streamlines the data collection process, but also improves the reliability and timeliness of the information collected.

2.1.3 Data Processing

2.1.3.1 Processing Script

The data processing stage initiates with the activation of the `computeAQI` function, following data ingestion. This crucial function uses user-defined functions (UDFs) to convert PM2.5 and PM10 concentration data into Air Quality Index (AQI) values, adhering to the guidelines specified in Table 2.1. The comprehensive implementation details of this function are available in the `processing.py` script, as detailed in Appendix B.C.

Key to this stage is the utilisation of Spark DataFrames, which offer an efficient means of managing and processing the data due to their distributed structure. The data is first restructured using the `explode` operation, which separates each sensor's data into individual rows, thus preparing it for the application of UDFs. To enhance efficiency, the DataFrame is then cached, significantly reducing the need for repetitive disk read/write operations and thereby accelerating the computation process.

Table 2.1: UK air quality index, “Review of the UK Air Quality Index”, 2011

Range	Air Quality Index	PM_{2.5} Particles, 24 hour mean ($\mu\text{g}/\text{m}^3$)	PM₁₀ Particles, 24 hour mean ($\mu\text{g}/\text{m}^3$)
Low	1	0-11	0-16
	2	12-23	17-33
	3	24-35	34-50
Medium	4	36-41	51-58
	5	42-47	59-66
	6	48-53	67-75
High	7	54-58	76-83
	8	59-64	84-91
	9	65-70	92-100
Very High	10	>70	>100

2.1.3.2 Leveraging Apache Spark

The incorporation of Apache Spark is central to the data processing methodology. Its distributed computing framework is adept at handling large datasets, characteristic of environmental sensor data. Spark’s features are leveraged as follows:

1. Spark’s architecture is uniquely designed to manage the fluctuating volume and nature of IoT sensor data. This scalability is critical for ensuring the efficient processing of data in real-time, which is essential for timely AQI calculation.
2. The distributed nature of Spark facilitates the parallel processing of tasks across multiple nodes. This capability is vital for the simultaneous processing of AQI calculations from multiple sensors, enhancing the efficiency and speed of data computation.
3. This methodical approach, leveraging Spark’s scalability and distributed processing capabilities, ensures that the project achieves prompt and accurate AQI values, which are crucial for real-time air quality monitoring and assessment.

2.1.4 Data Storing

2.1.4.1 Database Choice

The difficulty with this project lies in the efficient management of data storage. Each sensor can measure a variety of data, making the structure of a single relational database too complex and inefficient. This complexity increases as the volume of data increases, requiring the creation of multiple linked tables. This structural challenge poses a performance problem, particularly when it comes to processing and analysing large quantities of data in real time, a crucial aspect for applications such as air quality or environmental monitoring.

Faced with these challenges, time series databases (TSDBs) are emerging as an optimal solution. Unlike traditional relational databases, TSDBs are specifically designed to manage time-series data, such as that from IoT sensors. They offer better performance in

tracking, monitoring and aggregating data over time. In addition, TSDBs such as Amazon Timestream stand out for their ability to ingest data quickly and process large volumes efficiently, while ensuring regular updates and in-depth analysis, essential aspects for applications such as Air Quality Index analysis.

Amazon Timestream is a good choice for this project. As a cloud-native time-series database, it offers superior time-series management capabilities tailored to IoT sensor data. What sets Amazon Timestream apart is the speed with which it ingests data and its efficiency in processing large volumes of data, enabling regular updates and in-depth analysis. Its ability to adapt to changing workloads ensures consistent performance, a major advantage for real-time data management. What's more, its advanced security features meet strict standards of confidentiality and data sovereignty, an essential

2.1.4.2 Storing Script

The data storing process represents the final phase in the IoT data collecting and processing pipeline. Once the Air Quality Index (AQI) has been calculated, the data must be stored efficiently and securely to allow subsequent analysis and real-time consultation. The `storing.py` script (see Appendix [refappendix:storing](#) for detailed code) is designed to interact with the Amazon Timestream database.

The `keepOnlyUpdatedRows` function is responsible for checking what data is already in Timestream and keeping only the newly updated values. This process begins by querying Timestream to retrieve the latest data timestamp for each sensor using the ‘`boto3`’ client. The data is then filtered to exclude records that do not reflect new measurements, optimising storage space and database performance.

Once this filtering is complete, the `writeToTimestream` function takes over. It transforms each partition of the Spark DataFrame into a series of structured records, containing the dimensions and measurements corresponding to each sensor. These records are then written to Timestream in batches. The process is carefully managed to ensure that records are written atomically and consistently, with robust exception handling to deal with any rejected records.

In conclusion, the integration of Spark with Amazon Timestream offers a powerful solution for storing environmental sensor data. Spark’s ability to process and filter data in a distributed manner aligns perfectly with Timestream’s high availability and auto-scaling capabilities, ensuring a resilient, high-performance data infrastructure. This arrangement enables reliable data storage and rapid retrieval, which are essential for continuous monitoring of air quality and rapid responses to environmental issues.

2.1.5 Pipeline Implementation on AWS

Setting up an IoT data processing pipeline on Amazon Web Services (AWS) involves several key steps, from configuring the EC2 instance to running and managing the pipeline. The process is described below:

- EC2 Instance Configuration:** An EC2 t4g.small instance has been configured to host the data collecting, processing & storing pipeline. This type of instance was chosen for its balance between performance and energy efficiency. These instances, based on ARM architecture, are ideal for light to moderate workloads, which correspond to the requirements of continuous data processing. In addition, Apache Spark has been installed on this instance to process the data collected. Spark was chosen for its ability to process data in memory and in parallel, optimising performance for data processing workloads.
- Setting Up Automated Services:** Two systemd services were then set up to automate the essential tasks: The `get_iam_credentials.service` (see Appendix B.E.2.1) executes `get_iam_credentials.sh` (see Appendix B.E.1). This script retrieves the instance's IAM credentials, enabling secure integration with other AWS services. The `spark_python_job.service` (see Appendix B.E.2.2) launches `start_spark_job.sh` (see Appendix B.E.1), which starts the main data processing script (see Appendix B.A). This script orchestrates the collection, processing (AQI calculation) and storage of IoT sensor data. These services are activated each time the instance is started.

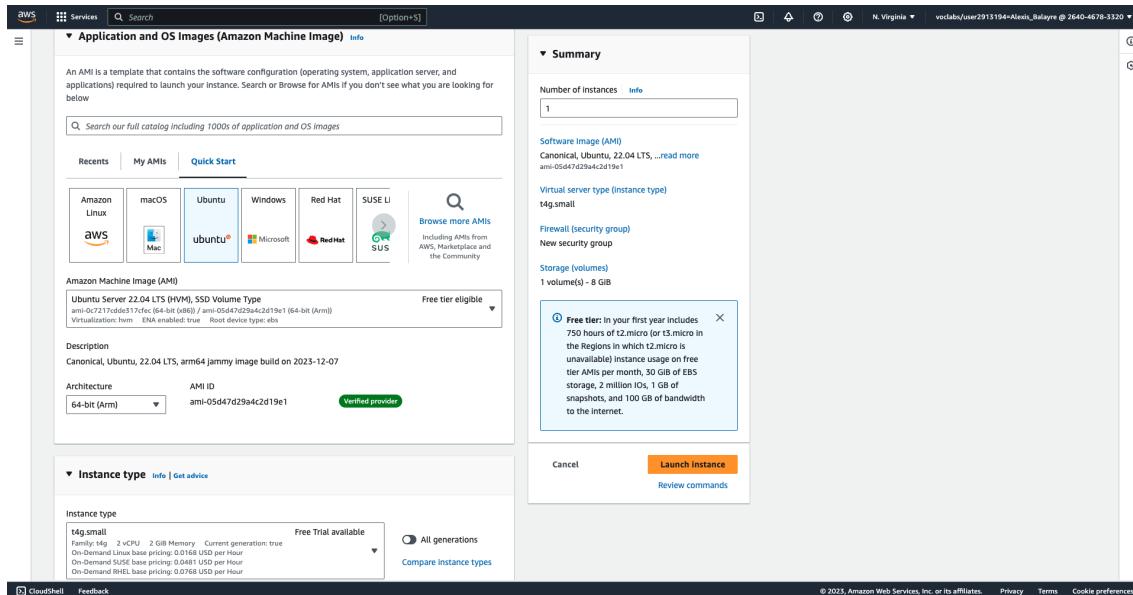


Figure 2.2: EC2 Instance Configuration Screenshot

2.2 Data Distributing

2.2.1 Overview of the Pipeline Architecture

The second pipeline in this project is dedicated to the effective distribution of processed data. This pipeline is comprised of several key components, each integral to the efficient dissemination and monitoring of data.

1. Internet Gateway:

The user's interaction with the system begins at the secure Internet gateway. This gateway serves as the primary access point to the system, directing users to the Grafana dashboards through a specified URL. This approach ensures secure and controlled access to the air quality data visualisations.

2. Load Balancer:

At the heart of the data distribution pipeline is the Load Balancer. This component is tasked with evenly distributing incoming traffic across multiple EC2 instances. The Load Balancer plays a crucial role in balancing the workload and optimising resource utilisation, ensuring that no single instance is overwhelmed, thus maintaining smooth and efficient data flow.

3. Grafana Dashboard Access:

For data visualisation, the Grafana dashboard is utilised. Hosted on a Load Balancer, the dashboard presents the data stored in the Amazon Timestream database. It offers real-time visualisation of air quality metrics, enabling users to interact with and analyse the data in a user-friendly manner. This real-time access is critical for timely decision-making and assessment of air quality trends.

4. CloudWatch Monitoring:

Finally, the AWS CloudWatch service continuously monitors the overall system performance. It specifically tracks the CPU utilization of all EC2 instances within the Auto Scaling Group. Based on the aggregated CPU usage data, CloudWatch dynamically adjusts the system's resources. This adaptive scaling is essential for maintaining the balance between system efficiency and cost-effectiveness, ensuring the infrastructure responds appropriately to varying loads.

For a comprehensive visual overview of this pipeline, please refer to Figure 2.3, which depicts the Data Distributing Pipeline.

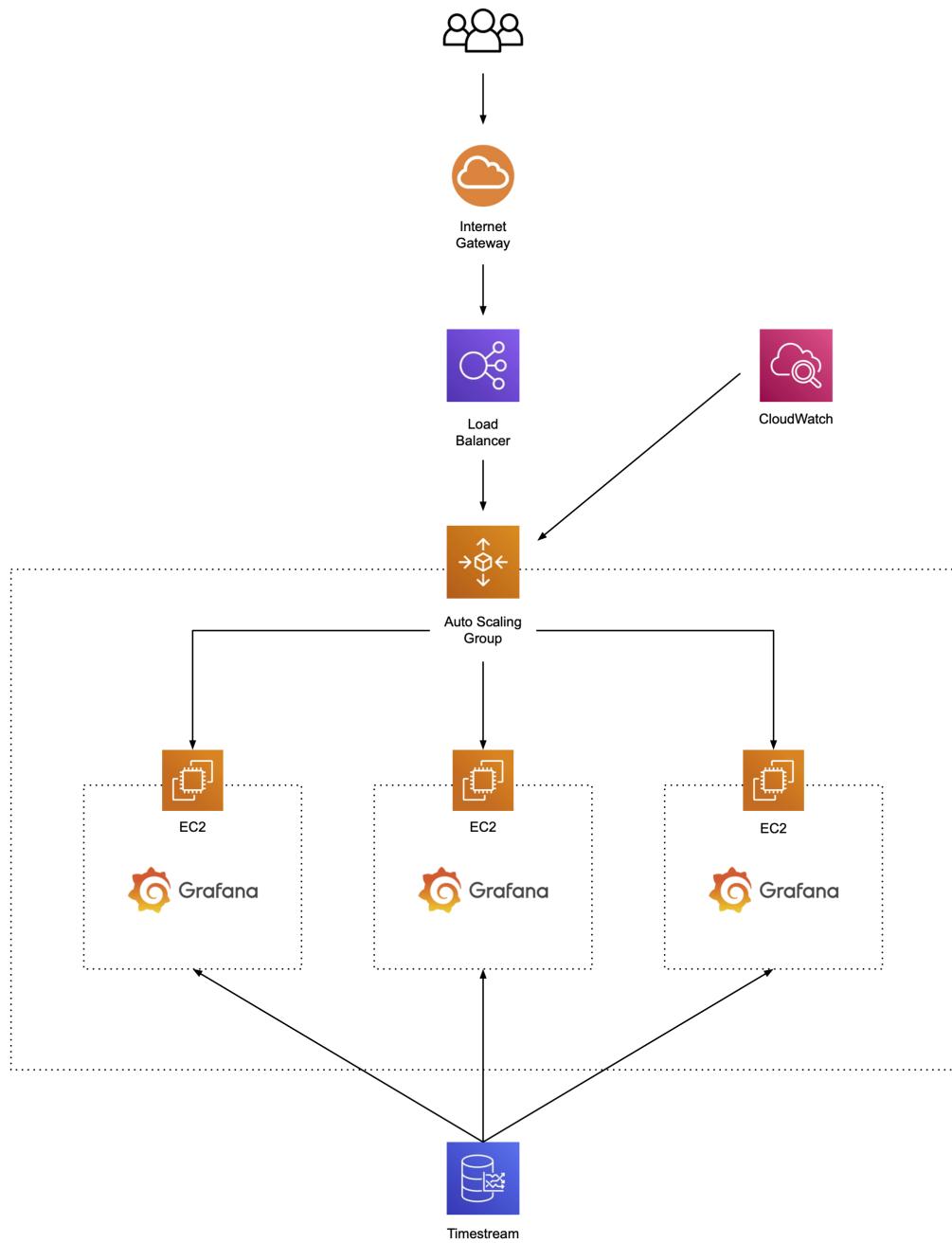


Figure 2.3: Data Distributing Pipeline Diagram

2.2.2 Auto Scaling Group Configuration

The Auto Scaling Group plays a crucial role in dynamically managing the EC2 instances of the Data Distributing Pipeline. This section outlines the configurations for the Auto Scaling Group and its associated components. The figure below(?) illustrates the Auto Scaling Group Pipeline.

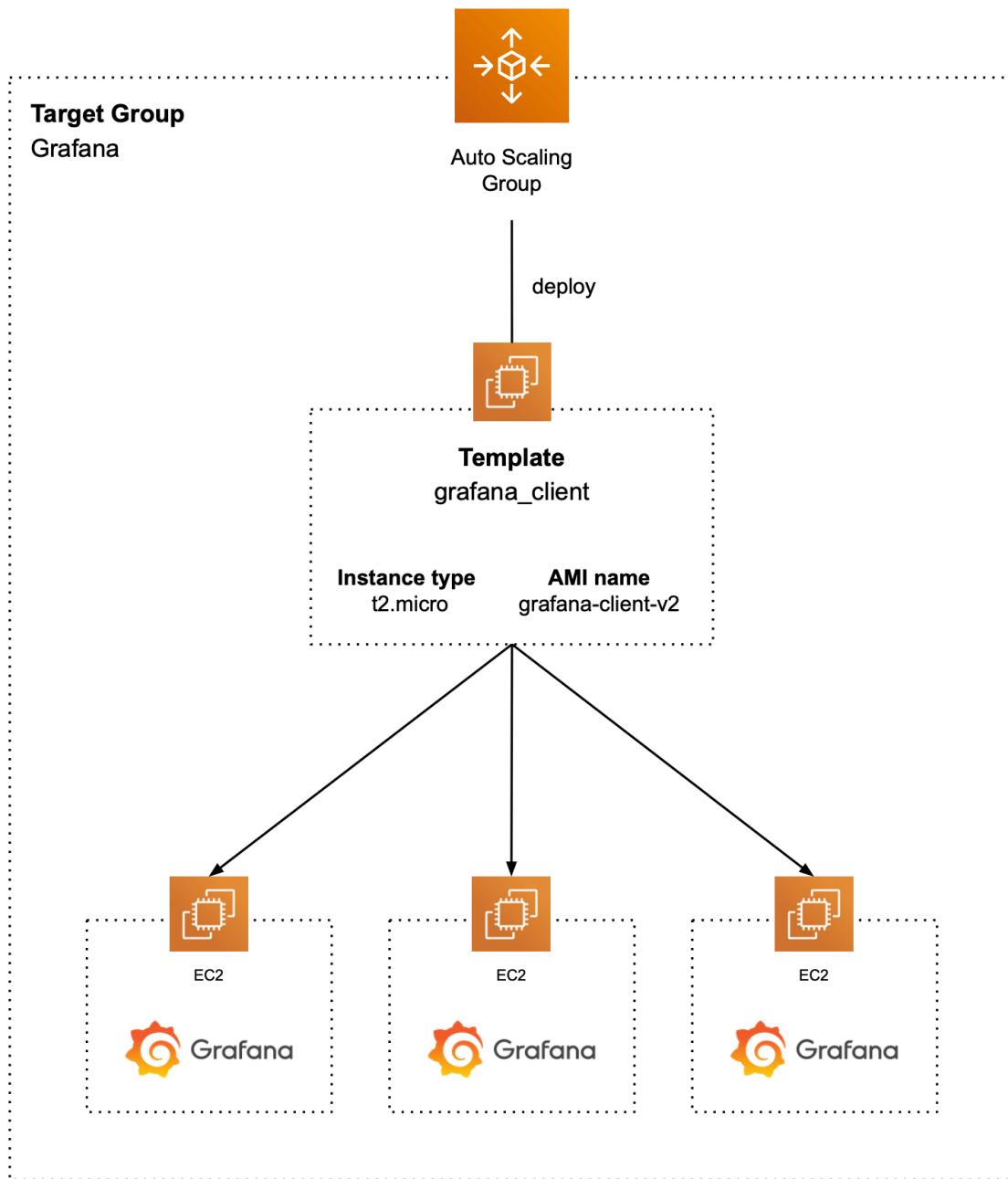


Figure 2.4: Auto Scaling Group Pipeline Diagram

2.2.2.1 AMI Configuration

An Amazon Machine Image (AMI) serves as a blueprint for launching an EC2 instance, containing the necessary software configuration. For this project, a custom AMI was created using the Amazon Linux 2 operating system, specifically tailored to host the Grafana server for efficient data visualisation.

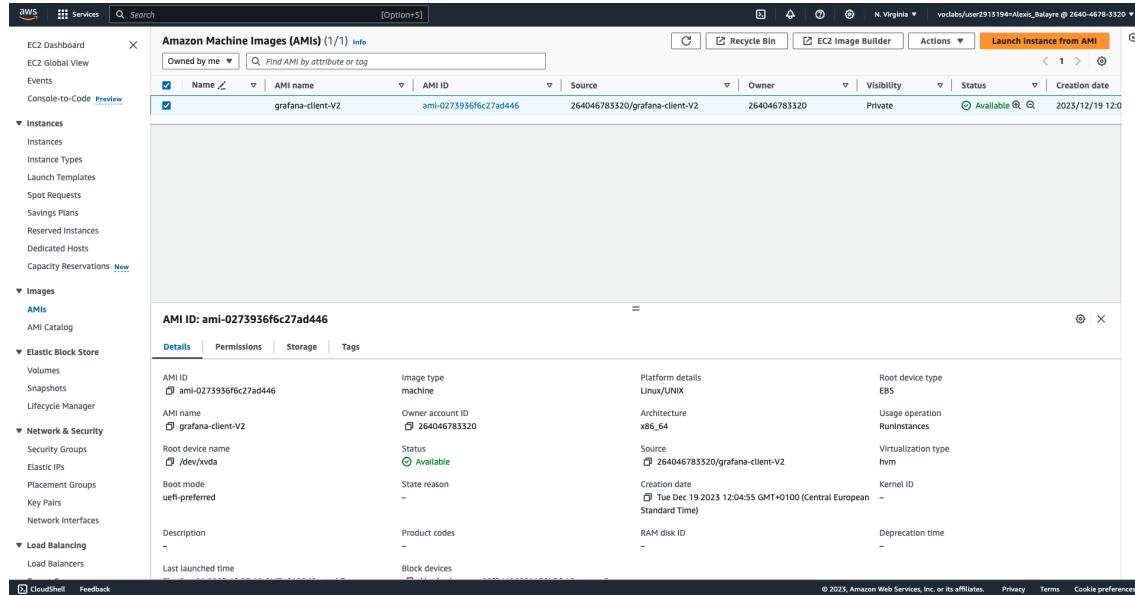


Figure 2.5: AMI Settings Screenshot

2.2.2.2 Launch Template Configuration

The launch template defines the specifications for deploying EC2 instances within the Auto Scaling Group. It incorporates the custom AMI and delineates instance type, security group, key pair, and user data. This template deploys t2.micro instances, which are ideal for light workloads, such as hosting the Grafana server. The template also specifies the security group, which allows inbound traffic on port 3000, enabling access to the Grafana dashboard.

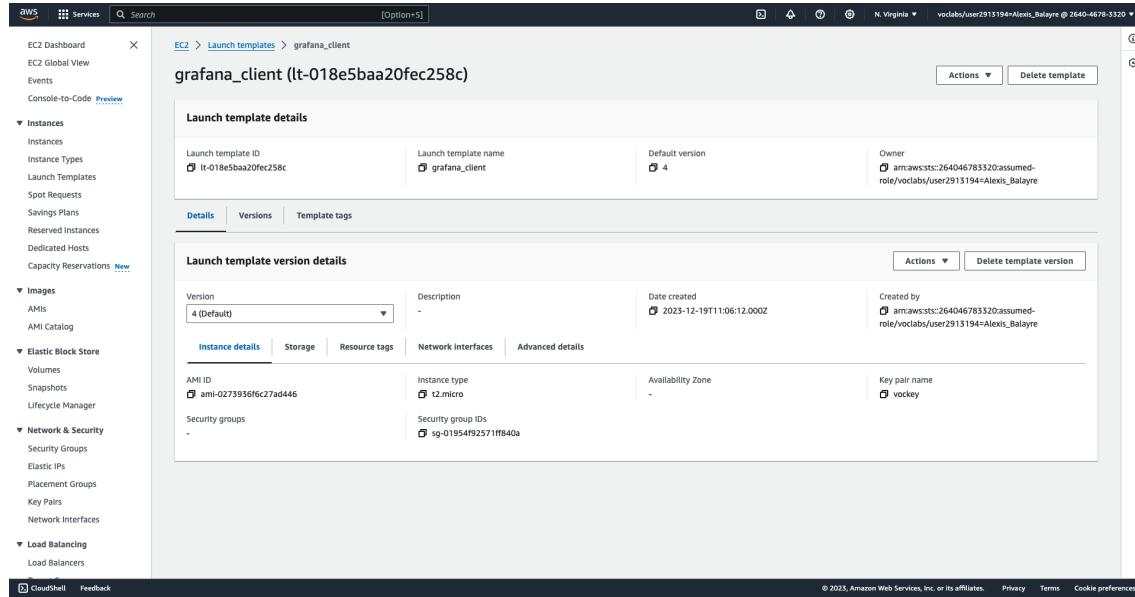


Figure 2.6: Launch Template Settings Screenshot

2.2.2.3 Auto Scaling Group Configuration

The Auto Scaling Group is configured to automatically adjust the number of EC2 instances in response to the changing demand. It scales between 1 and 5 instances based on CPU utilisation, enhancing the system's responsiveness and efficiency. The group is also linked to the specified target group and load balancer, facilitating effective traffic distribution.

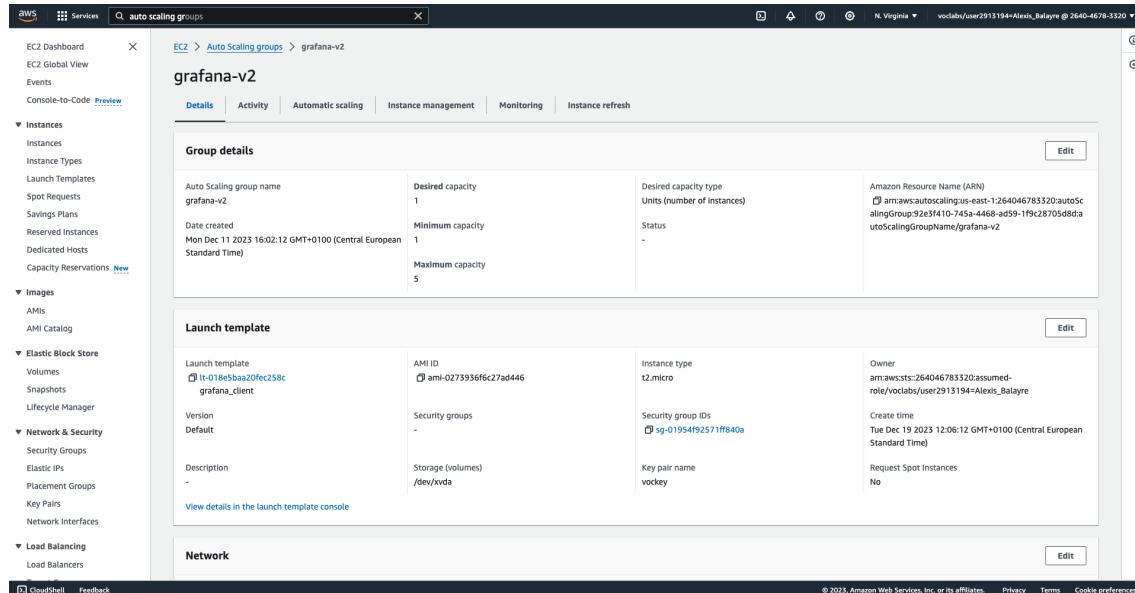


Figure 2.7: Auto Scaling Group Settings Screenshot

The Auto Scaling Group employs the following policies for scaling:

- **Scale-out Policy:** Activates when average CPU utilisation exceeds 60% for 3 minutes, adding an instance to the group.
- **Scale-in Policy:** Triggers when average CPU utilisation drops below 42% for 15 minutes, removing an instance.
- **Coldown Period:** Set at 150 seconds to maintain stability by preventing frequent scaling.

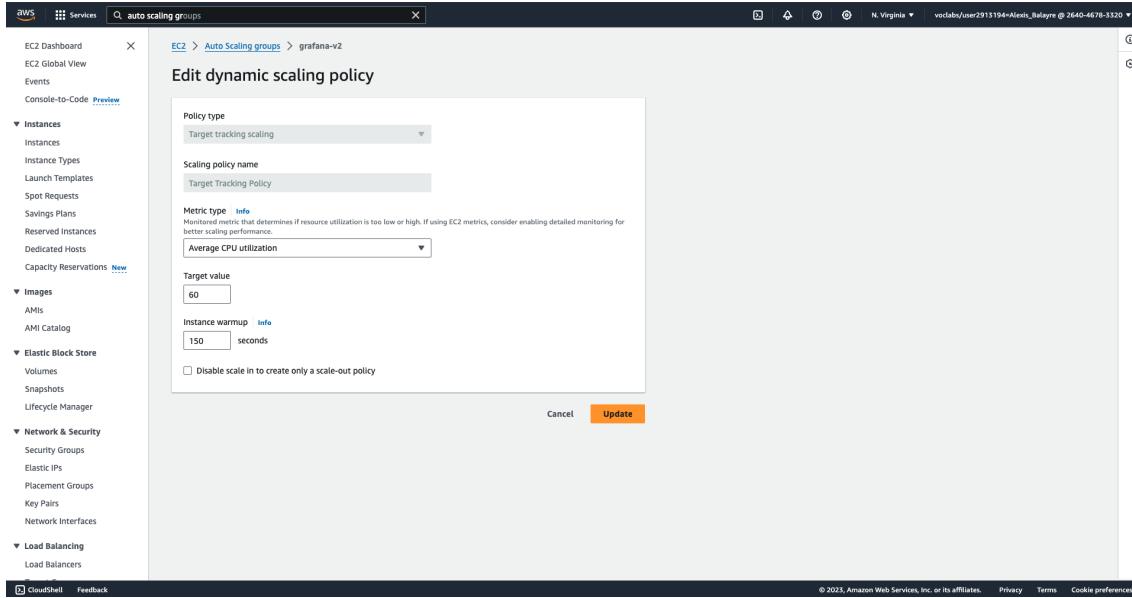


Figure 2.8: Scaling Policy Settings Screenshot

2.2.2.4 Target Group Configuration

A target group was created to distribute traffic between the EC2 instances of the Auto Scaling Group.

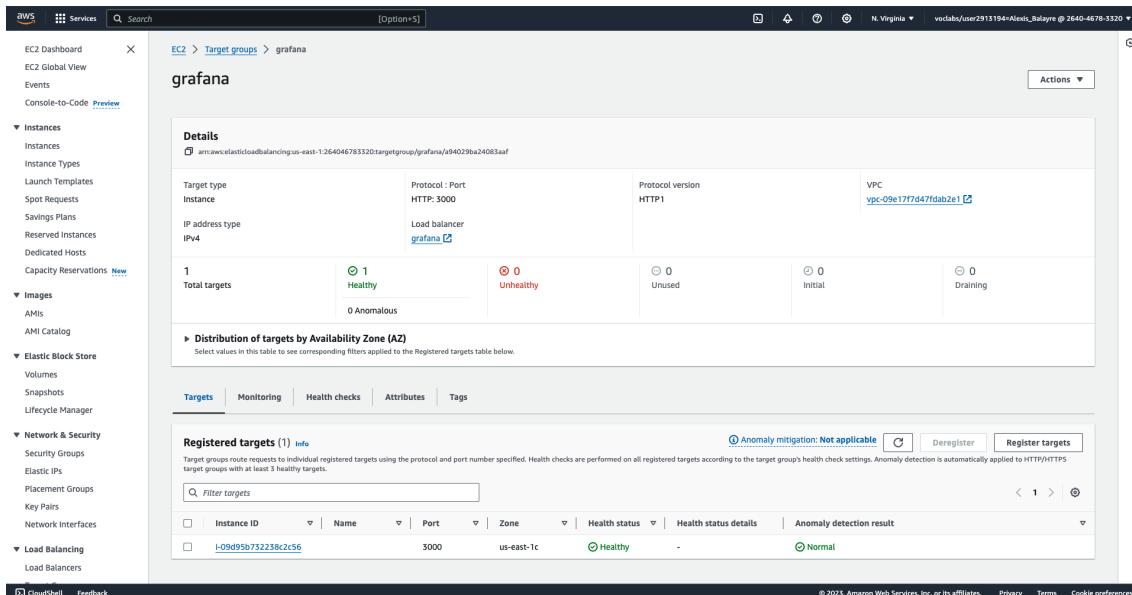


Figure 2.9: Target Group Settings Screenshot

2.2.2.5 Load Balancer Configuration

A load balancer was created to distribute traffic between the EC2 instances of the Auto Scaling Group. The load balancer is configured to use the target group and to listen on port 80. This is useful for ensuring that the Grafana dashboard is accessible to users through a user-friendly URL.

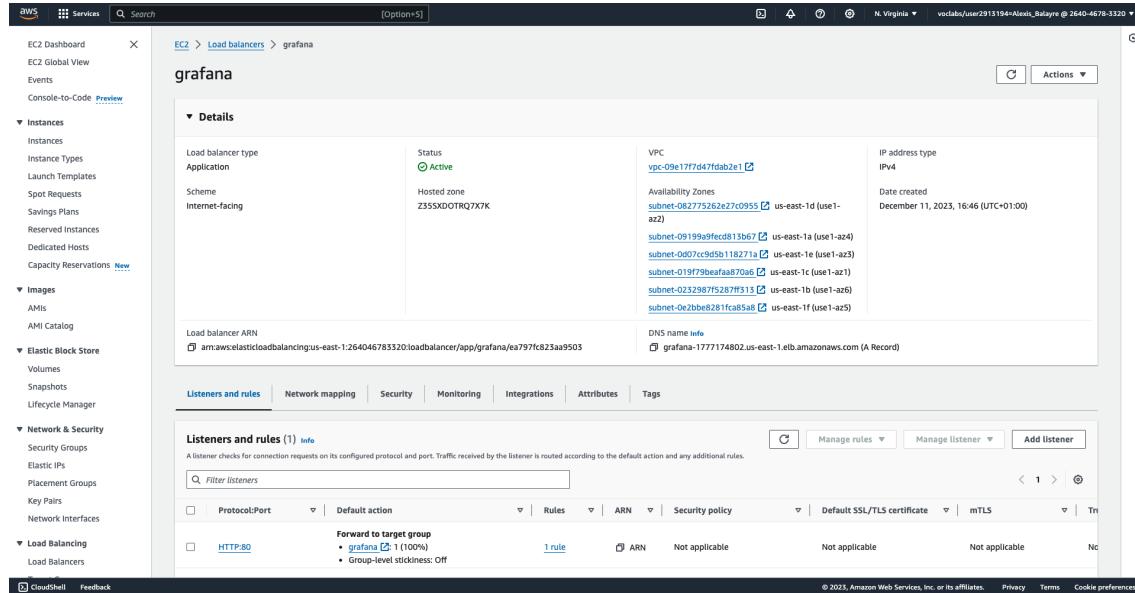


Figure 2.10: Load Balancer Settings Screenshot

2.3 Pipeline of the Project

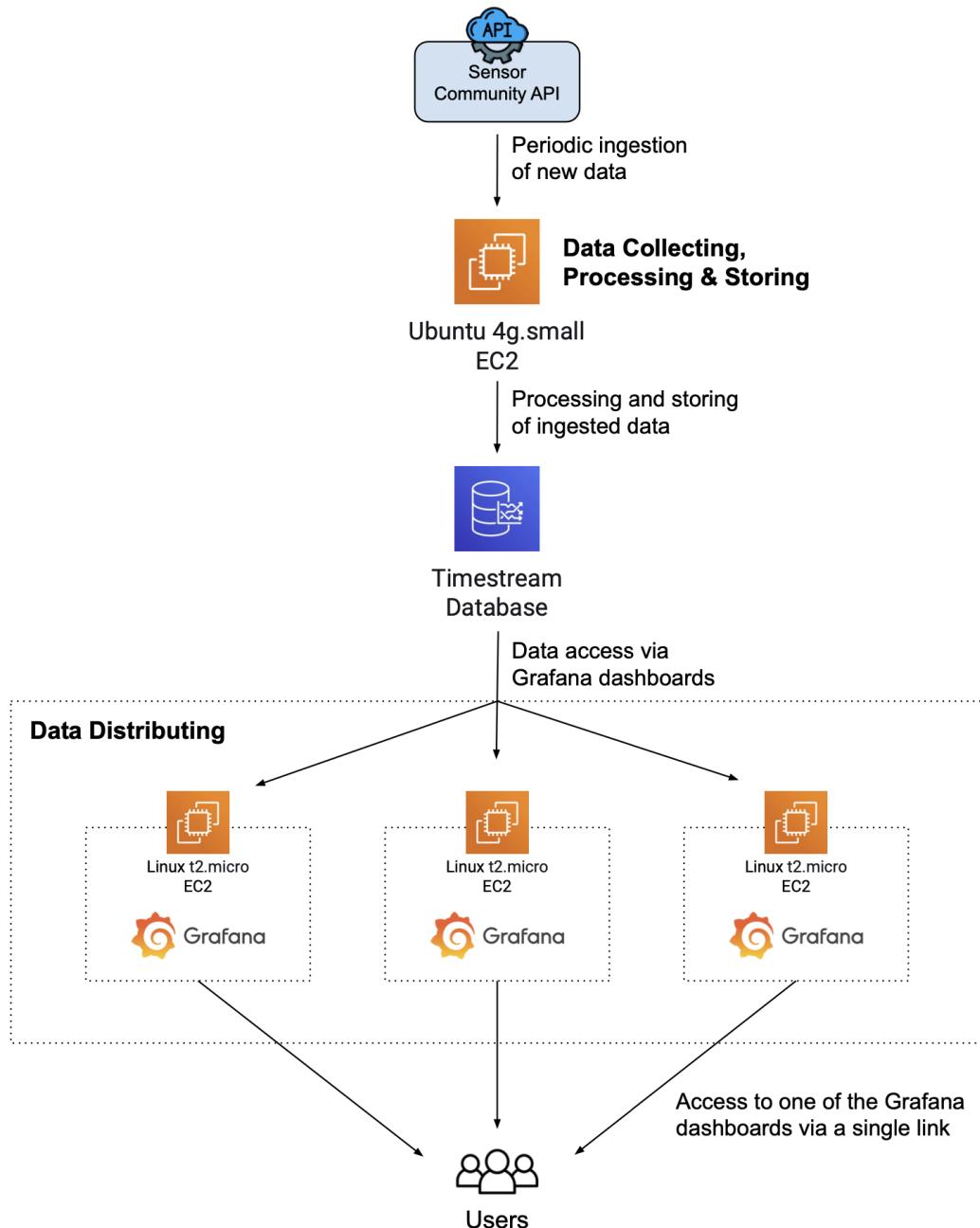


Figure 2.11: Data Distributing Pipeline Diagram

Chapter 3

Results & Discussion

3.1 Results

3.1.1 Accessing the Data - Grafana Dashboard

The Grafana dashboard, a vital tool in air quality monitoring, is accessible at the following URL: Air Quality Monitoring Dashboard.

The Grafana dashboard provides an interactive interface enabling users to analyse air quality data. Users can customise the time interval, measurement value and location ID. The dashboard consists of five main panels, each providing unique information:::

1. **Measure Value Evolution:** This panel shows the evolution of the selected measure value in the selected location over the selected time range.
2. **Max AQI:** This panel shows the maximum AQI value in the selected location over the selected time range.
3. **Mean AQI:** This panel shows the mean AQI value in the selected location over the selected time range.
4. **AQI Map:** This panel shows the AQI value of each location on a map.
5. **IoT Data:** This panel shows all the data collected by the IoT sensors related to the selected

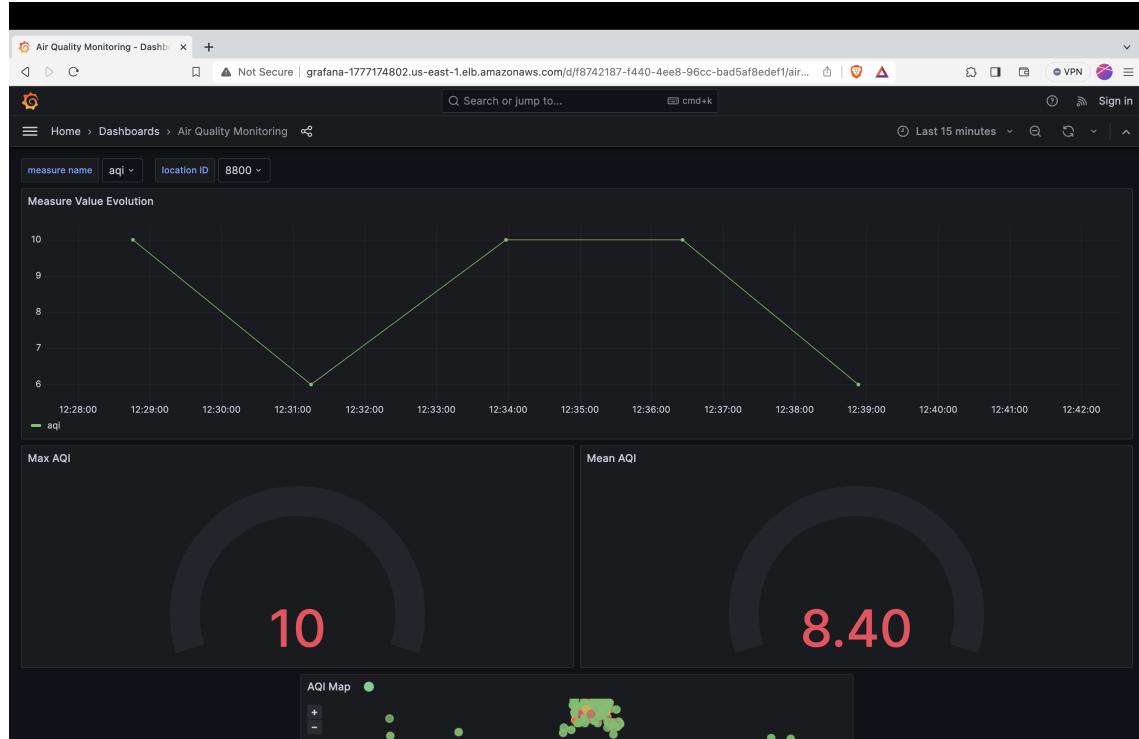


Figure 3.1: Grafana Dashboard: Measure Value Evolution, Max & Mean AQI

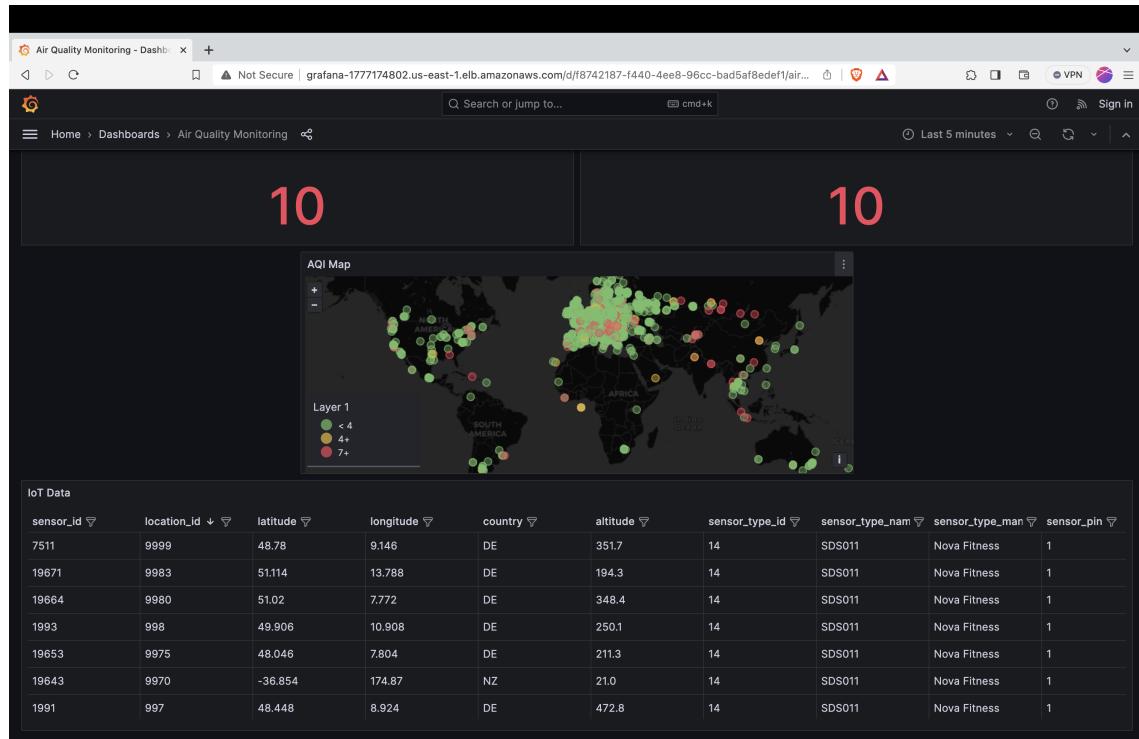


Figure 3.2: Grafana Dashboard: AQI Map & IoT Data

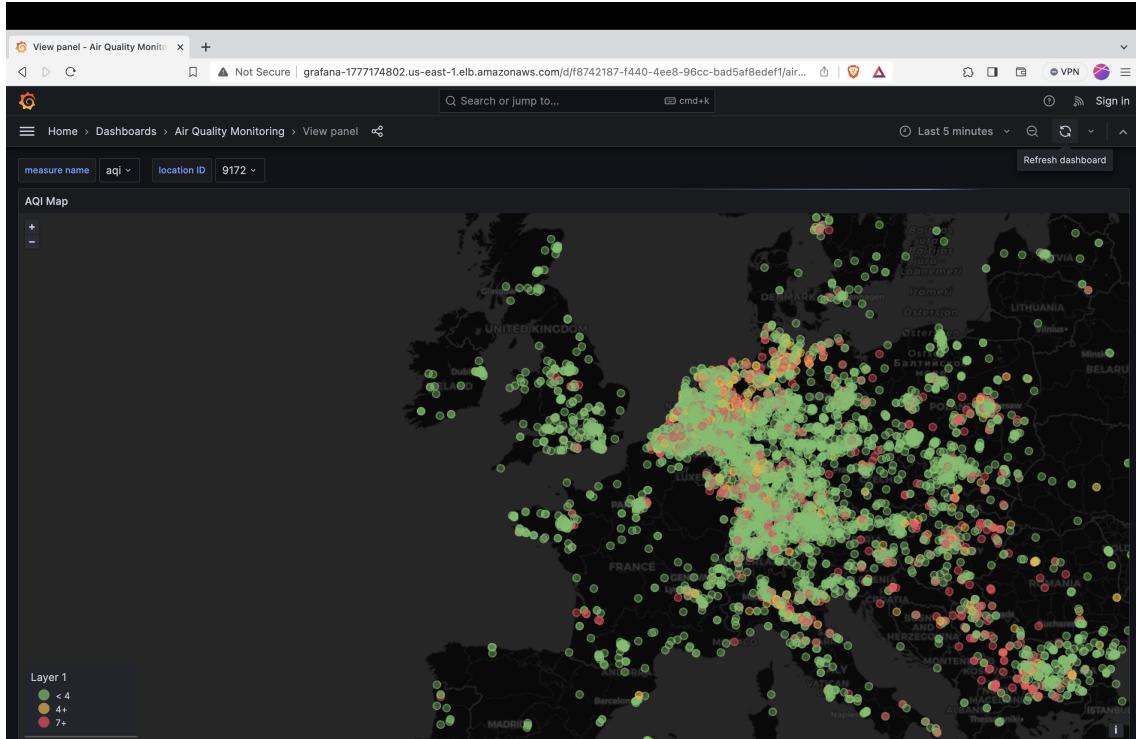


Figure 3.3: Grafana Dashboard AQI Map Panel

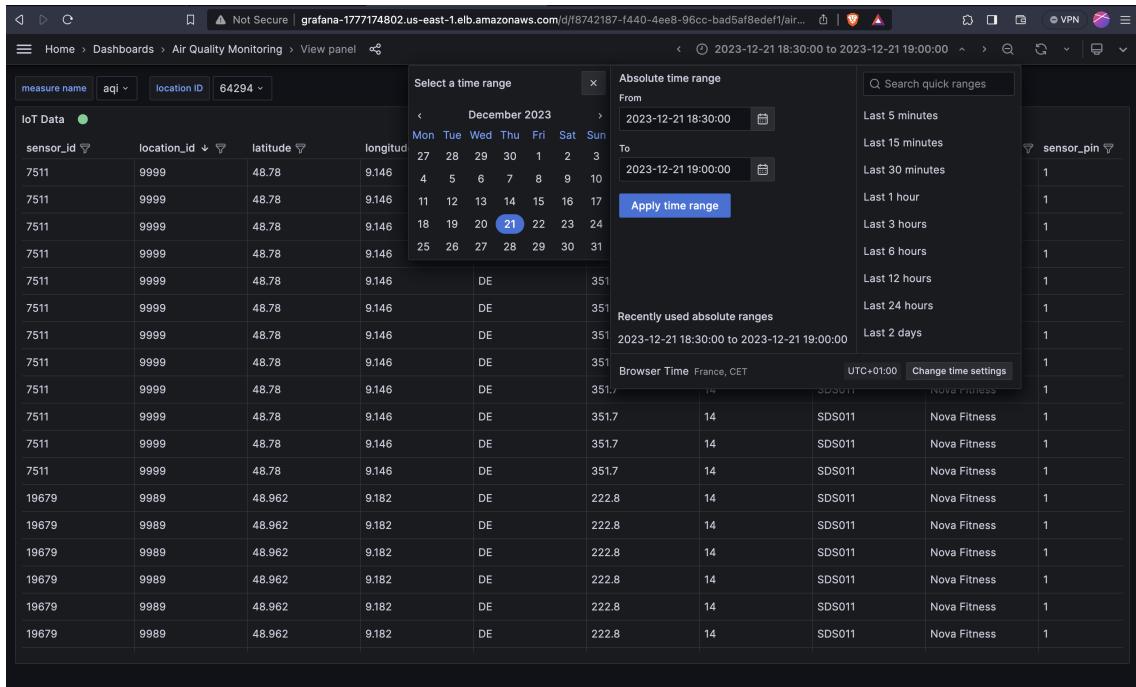


Figure 3.4: Grafana Dashboard IoT Data Panel - Time Range Choice

Chapter 3. Results & Discussion

3.1. Results

Figure 3.5: Grafana Dashboard IoT Data Panel - Measure Value Choice

Figure 3.6: Grafana Dashboard IoT Data Panel - Apply a Filter

The screenshot shows a Grafana dashboard titled "Air Quality Monitoring". On the left, there is a table titled "IoT Data" with columns: measure_name (aqi), location_ID (64294), sensor_id, location_id, latitude, longitude, and country. The data in the table consists of multiple rows of sensor readings. On the right, a modal window titled "Inspect: IoT Data" is open, showing the same data in a more detailed format. The modal has tabs for "Data", "Stats", "Meta data", and "JSON", with "Data" selected. It also includes "Data options" like "Formatted data" and "Download CSV" and "Download for Excel". The data in the modal is identical to the table on the left.

sensor_id	location_id	latitude	longitude	country
7511	9999	48.78	9.146	DE
7511	9999	48.78	9.146	DE
7511	9999	48.78	9.146	DE
7511	9999	48.78	9.146	DE
7511	9999	48.78	9.146	DE
7511	9999	48.78	9.146	DE
7511	9999	48.78	9.146	DE
7511	9999	48.78	9.146	DE
7511	9999	48.78	9.146	DE
7511	9999	48.78	9.146	DE
7511	9999	48.78	9.146	DE
7511	9999	48.78	9.146	DE
7511	9999	48.78	9.146	DE
7511	9999	48.78	9.146	DE
19679	9989	48.962	9.182	DE
19679	9989	48.962	9.182	DE
19679	9989	48.962	9.182	DE
19679	9989	48.962	9.182	DE
19679	9989	48.962	9.182	DE
19679	9989	48.962	9.182	DE
19679	9989	48.962	9.182	DE
19679	9989	48.962	9.182	DE
19679	9989	48.962	9.182	DE
15020	73813	48.50624640000	8.86927477149	DE
15020	73813	48.50624640000	8.86927477149	DE

Figure 3.7: Grafana Dashboard IoT Data Panel - Export Values

3.1.2 Load Test

In order to test the scalability of the system, a load test was performed using the Artillery load testing framework. The test consisted in simulating an increasing number of customers consuming the grafana dashboards simultaneously, to demonstrate the elasticity of the solution developed. The results of the test are shown in Figure 3.8.

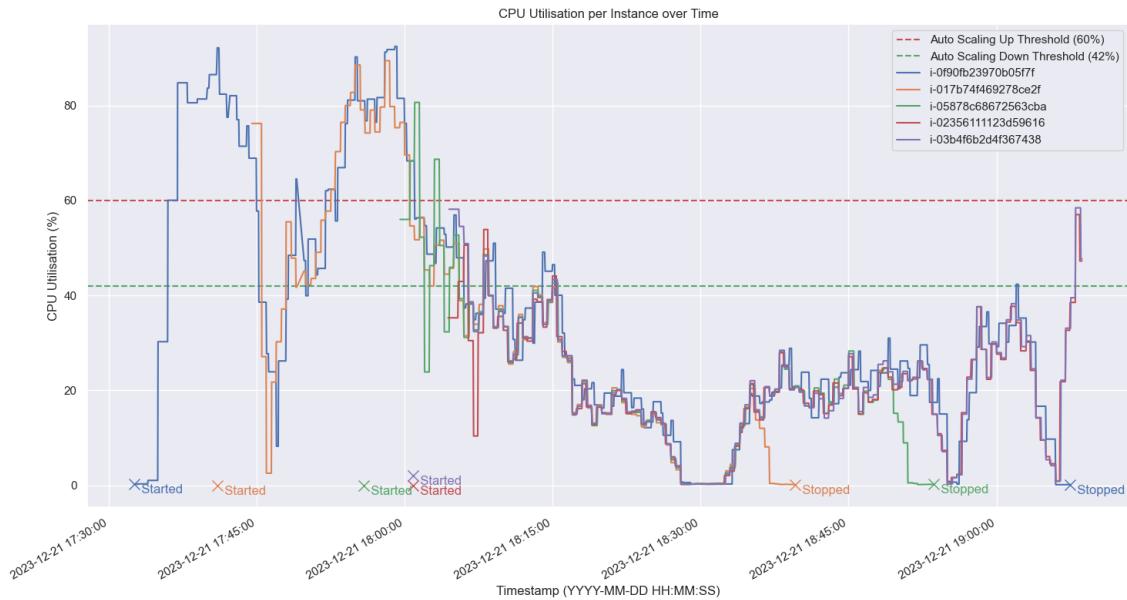


Figure 3.8: Load Test Result

3.2 Data Security and Sovereignty Considerations

Data security and sovereignty are critical in the management of environmental sensor data. Recent research trends in IoT security highlight the evolving landscape of threats and protections in this field (2).

3.2.1 Data Security

In the realm of IoT, ensuring data security involves advanced encryption, robust access control, and maintaining data integrity. The systematic mapping study by (**author?**) provides insights into current methodologies and research directions in IoT security, relevant to environmental sensor data.

3.2.2 Data Sovereignty

Data sovereignty is influenced by varying international legislations, such as GDPR, affecting data storage and cross-border data transfer. The study by (**author?**) underscores the importance of understanding these legal complexities in IoT environments.

3.2.3 Specific Fields for Further Examination

Areas needing heightened attention include personal data protection, sensitivity of environmental data, and metadata security, as indicated by recent IoT security research trends (2).

Adhering to the latest security trends and regulatory frameworks is vital for the ethical management of environmental sensor data. The evolving research in IoT security serves as a guide for future developments in this field (2).

Chapter 4

Conclusion

In conclusion, this study presents a significant advancement in air quality monitoring through IoT technologies, demonstrating the effectiveness of the Grafana dashboard in data visualisation and analysis. The successful load test indicates the system's scalability, catering to simultaneous user access. Addressing data security and sovereignty, the project highlights the importance of robust data management practices in IoT environments. While acknowledging certain limitations, this research lays a foundation for future explorations in IoT security and data handling. Overall, it contributes valuable insights to the field of environmental monitoring and data security, underscoring the potential of IoT in addressing critical ecological issues.

References

1. peoter. Are all Sensor Community sensors synchronized?. Sensor Community Forum; 2023. Available at: <https://forum.sensor.community/t/are-all-sensor-community-sensors-synchronized/2479/2>. (Accessed: December 18, 2023).
2. Vishal Sharma JYL, Lee J. Current Research Trends in IoT Security: A Systematic Mapping Study. Mobile Information Systems. 2021 Mar;2021:8847099. Available at: <https://doi.org/10.1155/2021/8847099>. (Accessed: December 26, 2023).

Appendix A

Documentation

Appendix A.A Project tree

```
lib /  
    collecting.py  
    processing.py  
    storing.py  
scripts /  
    get_iam_credentials.sh  
    start_spark_job.sh  
services /  
    get_iam_credentials.service  
    spark_python_job.service  
    grafana_server.service  
test /  
    artillery_load_test.yml  
    monitoring.py  
    metrics.csv  
    results.json  
    visualisation_load_test.ipynb  
main.py  
README.md  
requirements.txt
```

Appendix A.B Getting Started

To run the program, follow these steps:

1. Create a virtual environment using `python3 -m venv venv`.
2. Activate the virtual environment using `source venv/bin/activate`.
3. Install the required dependencies using `pip3 install -r requirements.txt`.
4. Run the program using `python3 main.py`.
5. Visualise the results using `visualisation.ipynb` (Jupyter Notebook).

Appendix A.C Detailed Features of Functions

collecting.py

- `fetch_sensors_data(sparkSession)`: Function to ingest the latest data from the sensors and returns it as a Spark DataFrame.

processing.py

- `get_aqi_value_p25(value)`: Function for calculating the AQI value for PM2.5.
- `get_aqi_value_p10(value)`: Function for calculating the AQI value for PM10.
- `computeAQI(df)`: Function for calculating the AQI value for each particulate matter sensor and returning the DataFrame with the AQI column.

storing.py

- `keepOnlyUpdatedRows(database_name, table_name, df)`: Function for keeping only the rows that have been updated in the DataFrame.
- `_print_rejected_records_exceptions(err)`: Internal function for printing the rejected records exceptions.
- `write_records(database_name, table_name, client, records)`: Internal function for writing a batch of records to the Timestream database.
- `writeToTimestream(database_name, table_name, partitionned_df)`: Function for writing the DataFrame to the Timestream database.

Appendix B

Source Codes

Appendix B.A Ingestion, Processing & Storing Pipeline Source Code

```
1 import findspark
2
3 findspark.init() # Initializing Spark
4
5 from pyspark.sql import SparkSession
6
7 import datetime as dt
8 import time
9
10 from lib.collecting import fetch_sensors_data
11 from lib.processing import computeAQI
12 from lib.storing import keepOnlyUpdatedRows, writeToTimestream
13
14 if __name__ == "__main__":
15     # Define the Timestream database and table names
16     DATABASE_NAME = "iot_project"
17     TABLE_NAME = "iot_table"
18
19     # Initializing Spark Session
20     sparkSession = (
21         SparkSession.builder.appName("Cloud Computing Project")
22             .master("local[*]")
23             .config("spark.sql.inMemoryColumnarStorage.compressed", "true")
24             .config("spark.sql.inMemoryColumnarStorage.batchSize", "10000")
25             .config("spark.serializer", "org.apache.spark.serializer.KryoSerializer")
26             .config("spark.ui.enabled", "true")
27             .config("spark.io.compression.codec", "snappy")
28             .config("spark.rdd.compress", "true")
29             .getOrCreate()
30     )
31
32     while True:
33         try:
34             print(
35                 dt.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
36                 + " Starting the pipeline..."
37             )
38             # Fetch the data from the sensors
39             iotDfRaw = fetch_sensors_data(sparkSession)
40
41             # Compute the AQI for each sensor
42             iotDfFormatted = computeAQI(iotDfRaw)
43
44             # Filter the data to keep only the updated rows
```

Appendix B. Source Codes B.A. Ingestion, Processing & Storing Pipeline Source Code

```
45         dataFiltered = keepOnlyUpdatedRows(
46             DATABASE_NAME, TABLE_NAME, iotDfFormatted
47         )
48
49     # Write the data to Timestream
50     print(
51         dt.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
52         + " 4. Writing the data to Timestream..."
53     )
54     dataFiltered.foreachPartition(
55         lambda partition: writeToTimestream(
56             DATABASE_NAME, TABLE_NAME, partition
57         )
58     )
59     print(
60         dt.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
61         + " Done writing the data to Timestream.\n"
62     )
63
64     # Sleep for 10 seconds
65     print(
66         dt.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
67         + " Done with the pipeline. Waiting for 10 seconds.\n"
68     )
69     time.sleep(10)
70 except Exception as e:
71     print(f"Exception: {e}")
```

Appendix B.B Data Collecting Source Code

```

1 # collecting.py
2 # The first step of the pipeline
3
4 from requests import Session
5 import datetime as dt
6
7
8 def fetch_sensors_data(sparkSession):
9     """
10     Fetches the latest data from the sensors and returns it as a Spark DataFrame
11
12     Args:
13         sparkSession (SparkSession): The SparkSession instance
14
15     Returns:
16         df (DataFrame): The DataFrame containing the last data from the sensors
17     """
18
19     # Fetches the latest data from the data.sensor.community API
20     url = "https://data.sensor.community/static/v2/data.24h.json"
21     # Use a session to avoid creating a new connection for each request
22     session = Session()
23     try:
24         print(
25             dt.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
26             + " 1. Fetching the latest data..."
27         )
28         response = session.get(url)
29         # If the response was successful, no Exception will be raised
30         if response.status_code == 200 and response.content:
31             # Convert the response to a Spark DataFrame
32             df = sparkSession.read.option("multiline", "true").json(
33                 sparkSession.sparkContext.parallelize([response.text])
34             )
35             print(
36                 dt.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
37                 + " Done fetching the latest data.\n"
38             )
39             return df
40     except Exception as e:
41         print(f"Request failed with exception {e}")
42     finally:
43         session.close()
44     return None

```

Appendix B.C Data Processing Source Code

```

1 # collecting.py
2 # The second step of the pipeline
3
4 from pyspark.sql.types import FloatType, IntegerType
5 import pyspark.sql.functions as F
6 import datetime as dt
7
8
9 # Defining a UDF to compute the AQI value for PM2.5
10 @F.udf(returnType=IntegerType())
11 def get_aqi_value_p25(value):
12     """
13         Computes the AQI value for PM2.5
14
15     Args:
16         value (float): The value of PM2.5
17     Returns:
18         aqi (int): The AQI value
19     """
20
21     if value is None:
22         return None
23     if 0 <= value <= 11:
24         return 1
25     elif 12 <= value <= 23:
26         return 2
27     elif 24 <= value <= 35:
28         return 3
29     elif 36 <= value <= 41:
30         return 4
31     elif 42 <= value <= 47:
32         return 5
33     elif 48 <= value <= 53:
34         return 6
35     elif 54 <= value <= 58:
36         return 7
37     elif 59 <= value <= 64:
38         return 8
39     elif 65 <= value <= 70:
40         return 9
41     return 10
42
43
44 # Defining a UDF to compute the AQI value for PM10
45 @F.udf(returnType=IntegerType())
46 def get_aqi_value_p10(value):
47     """
48         Computes the AQI value for PM10
49
50     Args:
51         value (float): The value of PM10
52
53     Returns:
54         aqi (int): The AQI value
55     """
56
57     if value is None:
58         return None
59     if 0 <= value <= 16:
60         return 1
61     elif 17 <= value <= 33:
62         return 2
63     elif 34 <= value <= 50:
64         return 3
65     elif 51 <= value <= 58:
66         return 4
67     elif 59 <= value <= 66:

```

```

68     return 5
69 elif 67 <= value <= 75:
70     return 6
71 elif 76 <= value <= 83:
72     return 7
73 elif 84 <= value <= 91:
74     return 8
75 elif 92 <= value <= 99:
76     return 9
77 return 10
78
79
80 def computeAQI(df):
81 """
82     Computes the AQI for each particulate matter sensor
83
84     Args:
85         df (DataFrame): The DataFrame containing the data from the sensors
86
87     Returns:
88         df_grouped (DataFrame): The DataFrame containing the AQI for each sensor
89     """
90
91     print(dt.datetime.now().strftime("%Y-%m-%d %H:%M:%S") + " 2. Computing the AQI
92     ...")
93     df_exploded = df.withColumn(
94         "sensordatavalue",
95         F.explode("sensordatavalues"), # Explode the sensordatavalues column
96     ).withColumn(
97         "aqi",
98         F.when(
99             F.col("sensordatavalue.value_type") == "P1",
100             get_aqi_value_p25(
101                 F.col("sensordatavalue.value").cast(FloatType())
102             ), # Cast the value to float and compute the AQI of PM2.5
103         ).when(
104             F.col("sensordatavalue.value_type") == "P2",
105             get_aqi_value_p10(
106                 F.col("sensordatavalue.value").cast(FloatType())
107             ), # Cast the value to float and compute the AQI of PM10
108         )
109     df_exploded.cache() # Cache the DataFrame to avoid recomputing it
110     df_grouped = (
111         df_exploded.groupBy("sensor.id", "timestamp") # Group by sensor and
112         timestamp
113         .agg(
114             F.first("id").alias("id"),
115             F.first("location").alias("location"),
116             F.first("sensor").alias("sensor"),
117             F.max("aqi").alias("aqi"), # Compute the maximum AQI between PM2.5 and
118             PM10
119             F.collect_list("sensordatavalue").alias("sensordatavalues"),
120         ) # Aggregate the AQI and the sensordatavalues
121         .selectExpr(
122             "sensor.id as sensor_id",
123             "sensor.pin as sensor_pin",
124             "sensor.sensor_type.id as sensor_type_id",
125             "sensor.sensor_type.manufacturer as sensor_type_manufacturer",
126             "sensor.sensor_type.name as sensor_type_name",
127             "location.country as country",
128             "location.latitude as latitude",
129             "location.longitude as longitude",
130             "location.altitude as altitude",
131             "location.id as location_id",
132             "aqi",
133             "sensordatavalues",
134             "timestamp",
135         ) # Select the columns to keep
136     )

```

```
135     df_exploded.unpersist() # Unpersist the DataFrame to free memory
136     print(
137         dt.datetime.now().strftime("%Y-%m-%d %H:%M:%S") + " Done computing the AQI
138         .\n"
139     )
140     return df_grouped
```

Appendix B.D Data Storing Source Code

```

1 # storing.py
2 # The last step of the pipeline
3
4 from pyspark.sql.types import BooleanType
5 import pyspark.sql.functions as F
6 from pyspark.sql import Row
7 from botocore.config import Config
8 import boto3
9 import time
10 import datetime as dt
11
12
13 def keepOnlyUpdatedRows(database_name, table_name, df):
14     """
15         Verifies if the data is already stored in Timestream and keeps only the updated
16         values
17
18     Args:
19         database_name (string): The name of the database
20         table_name (string): The name of the table
21         df (DataFrame): The DataFrame containing the data to be stored
22
23     Returns:
24         df_updated (DataFrame): The DataFrame containing only the updated rows
25     """
26
27     print(
28         dt.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
29         + " 3. Filtering the data to keep only the updated rows..."
30     )
31     query = """
32         SELECT sensor_id, MAX(time) as last_timestamp
33         FROM {}({})
34         GROUP BY sensor_id
35     """.format(
36         database_name, table_name
37     )
38
39     # Initialize the boto3 client
40     session = boto3.Session() # Create a boto3 session
41     query_client = session.client(
42         "timestream-query", config=Config(region_name="us-east-1")
43     ) # Create a boto3 client
44     paginator = query_client.getPaginator("query") # Create a paginator
45
46     # Get the last timestamp for each sensor
47     last_timestamps = (
48         {}
49     ) # Initialize a dictionary to store the last timestamp for each sensor
50     response_iterator = paginator.paginate(QueryString=query) # Paginate the query
51     for response in response_iterator:
52         for row in response["Rows"]:
53             sensor_id = row["Data"][0]["ScalarValue"]
54             last_timestamps[sensor_id] = row["Data"][1]["ScalarValue"]
55
56     # If there is no data in Timestream, return the DataFrame as is
57     if len(last_timestamps) == 0:
58         print("No data in Timestream")
59         return df
60
61     # Define an UDF to check if the row is updated
62     @F.udf(returnType=BooleanType())
63     def isUpdated(sensor_id, timestamp):
64         """
65             Checks if the row is updated
66
67         Args:

```

```

67         sensor_id (string): The sensor ID
68         timestamp (string): The timestamp of the row
69
70     Returns:
71         isUpdated (boolean): True if the row is updated, False otherwise
72     """
73
74     if str(sensor_id) not in last_timestamps:
75         return True
76     current_timestamp = dt.datetime.strptime(timestamp, "%Y-%m-%d %H:%M:%S")
77     last_timestamp_micro = last_timestamps[str(sensor_id)][
78         :26
79     ] # Keep only up to microseconds
80     last_sensor_timestamp = dt.datetime.strptime(
81         last_timestamp_micro, "%Y-%m-%d %H:%M:%S.%f"
82     )
83     return (
84         current_timestamp > last_sensor_timestamp
85     ) # Return True if the row is updated
86
87     df_updated = df.filter(
88         isUpdated("sensor_id", "timestamp")
89     ) # Filter the DataFrame to keep only the updated rows
90     print(
91         dt.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
92         + " Done filtering the data to keep only the updated rows.\n"
93     )
94     return df_updated
95
96
97 def _print_rejected_records_exceptions(err):
98     """
99     Prints the rejected records exceptions
100
101    Args:
102        err (RejectedRecordsException): The RejectedRecordsException
103    """
104
105    print("RejectedRecords: ", err)
106    for rr in err.response["RejectedRecords"]:
107        print("Rejected Index " + str(rr["RecordIndex"]) + ": " + rr["Reason"])
108        if "ExistingVersion" in rr:
109            print("Rejected record existing version: ", rr["ExistingVersion"])
110
111
112 def write_records(database_name, table_name, client, records):
113     """
114     Helper function to write records to Timestream
115
116    Args:
117        database_name (string): The name of the database
118        table_name (string): The name of the table
119        client (TimestreamWriteClient): The TimestreamWriteClient
120        records (list): The list of records to write
121    """
122    try:
123        result = client.write_records(
124            DatabaseName=database_name,
125            TableName=table_name,
126            CommonAttributes={},
127            Records=records,
128        )
129        print(
130            "WriteRecords Status: [%s]" % result["ResponseMetadata"]["
131            HTTPStatusCode"]
132        )
133    except client.exceptions.RejectedRecordsException as err:
134        _print_rejected_records_exceptions(err)
135    except Exception as err:
136        print("Error:", err)

```

```

136
137
138 def writeToTimestream(database_name, table_name, partitionned_df):
139     """
140     Writes the data to Timestream
141
142     Args:
143         database_name (string): The name of the database
144         table_name (string): The name of the table
145         partitionned_df (DataFrame): The DataFrame containing the data to be stored
146     """
147
148     # Initialize the boto3 client for each partition
149     session = boto3.Session()
150     write_client = session.client(
151         "timestream-write",
152         config=Config(
153             read_timeout=20, max_pool_connections=5000, retries={"max_attempts": 10}
154         ),
155     )
156
157     # Create a list of records
158     records = []
159     for row in partitionned_df:
160         try:
161             # Skip rows that are not of type Row
162             if not isinstance(row, Row):
163                 continue
164
165             # Convert timestamp to Unix epoch time in milliseconds
166             timestamp_datetime = dt.datetime.strptime(
167                 row.timestamp, "%Y-%m-%d %H:%M:%S"
168             )
169             row_timestamp = str(int(timestamp_datetime.timestamp() * 1000))
170
171             # altitude
172             altitude = row.altitude if row.altitude != "" else 0
173
174             # Create dimensions list
175             dimensions = [
176                 {"Name": "country", "Value": str(row.country)},
177                 {"Name": "latitude", "Value": str(row.latitude)},
178                 {"Name": "longitude", "Value": str(row.longitude)},
179                 {"Name": "altitude", "Value": str(altitude)},
180                 {"Name": "location_id", "Value": str(row.location_id)},
181                 {"Name": "sensor_id", "Value": str(row.sensor_id)},
182                 {"Name": "sensor_pin", "Value": str(row.sensor_pin)},
183             {
184                 "Name": "sensor_type_manufacturer",
185                 "Value": str(row.sensor_type_manufacturer),
186             },
187             {"Name": "sensor_type_name", "Value": str(row.sensor_type_name)},
188             {"Name": "sensor_type_id", "Value": str(row.sensor_type_id)},
189         ]
190
191         # Create a record for each measurement
192         measuresValues = []
193         for measure in row.sensordatavalues:
194             measureValue = {
195                 "Name": measure.value_type,
196                 "Value": str(measure.value),
197                 "Type": "DOUBLE",
198             }
199             measuresValues.append(measureValue)
200
201         if measure.value_type == "P2" and row.aqi is not None:
202             aqi_measureValue = {
203                 "Name": "aqi",
204                 "Value": str(row.aqi),
205             }

```

```

205             "Type": "BIGINT",
206         }
207         measuresValues.append(aqi_measureValue)
208
209     # Create a record for each sensor
210     record = {
211         "Dimensions": dimensions,
212         "Time": row_timestamp,
213         "TimeUnit": "MILLISECONDS",
214         "MeasureName": "air_quality",
215         "MeasureValueType": "MULTI",
216         "MeasureValues": measuresValues,
217     }
218     records.append(record)
219
220     # Write records to Timestream if there are 98 records
221     if len(records) >= 98:
222         write_records(
223             database_name, table_name, write_client, records
224         ) # Write records to Timestream
225         records = [] # Reset the records list
226         time.sleep(1) # Sleep for 1 second
227
228     except Exception as e:
229         print(f"Error processing row: {row}")
230         print(f"Exception: {e}")
231
232     # Write records to Timestream if there are any remaining records
233     if len(records) > 100:
234         while len(records) > 100:
235             write_records(
236                 database_name, table_name, write_client, records[:99]
237             ) # Write records to Timestream
238             records = records[99:] # Keep the remaining records
239             time.sleep(1) # Sleep for 1 second
240     elif len(records) > 0:
241         write_records(database_name, table_name, write_client, records)

```

Appendix B.E Scripts & Services Source Codes

B.E.1 Scripts

Script used by the `get_iam_credentials` service to retrieve the IAM credentials from the metadata server.

```
#!/bin/bash

# Get the authentication token from the EC2 metadata service
TOKEN=$(curl -X PUT "http://169.254.169.254/latest/api/token" -H "X-aws-ec2-
metadata-token-ttl-seconds: 21600" -s)

# Name of the IAM role to assume
ROLE_NAME="LabRole"

# Get temporary credentials using the IAM role
IAM_ROLE_CREDENTIALS=$(curl -H "X-aws-ec2-metadata-token: $TOKEN" -s http
://169.254.169.254/latest/meta-data/iam/security-credentials/$ROLE_NAME)

# Extract the credentials and session token
AWS_ACCESS_KEY_ID=$(echo $IAM_ROLE_CREDENTIALS | jq -r .AccessKeyId)
AWS_SECRET_ACCESS_KEY=$(echo $IAM_ROLE_CREDENTIALS | jq -r .SecretAccessKey)
AWS_SESSION_TOKEN=$(echo $IAM_ROLE_CREDENTIALS | jq -r .Token)
AWS_DEFAULT_REGION="us-east-1"

# Export the credentials and session token
export AWS_ACCESS_KEY_ID
export AWS_SECRET_ACCESS_KEY
export AWS_SESSION_TOKEN
export AWS_DEFAULT_REGION
```

Script used by the `spark_python_job` service to run the Python Spark job.

```
#!/bin/bash

# Run the spark job in the background and log output to output.log file
nohup python3 /home/ubuntu/iot_project/ingestion_iot.py >/home/ubuntu/iot_project/
output.log 2>&1 &
```

B.E.2 Services

B.E.2.1 Get IAM Credentials Service

Service used by the Ubuntu EC2 instance to retrieve the IAM credentials from the metadata server.

```
[Unit]
Description=Script to setup AWS cli thanks to the attached IAM Profile

[Service]
ExecStart=/usr/local/bin/get_iam_credentials.sh

[Install]
WantedBy=multi-user.target
```

B.E.2.2 Spark Python Job Service

Service used by the Ubuntu EC2 instance to run the Python Spark job (Data Collecting, Processing and Storing).

```
[Unit]
Description=Script to run the ingestion python script

[Service]
ExecStart=/usr/local/bin/start_spark_job.sh

[Install]
WantedBy=multi-user.target
```

B.E.2.3 Grafana Server Service

Service used by the Linux EC2 instances to run the Grafana server (Data Distributing).

```
[Unit]
Description=Grafana instance
Documentation=http://docs.grafana.org
Wants=network-online.target
After=network-online.target
After=postgresql.service mariadb.service mysqld.service influxdb.service

[Service]
EnvironmentFile=/etc/sysconfig/grafana-server
User=grafana
Group=grafana
Type=notify
Restart=on-failure
WorkingDirectory=/usr/share/grafana
RuntimeDirectory=grafana
RuntimeDirectoryMode=0750
ExecStart=/usr/share/grafana/bin/grafana server
    \
        --config=${CONF_FILE}
    \
        --pidfile=${PID_FILE_DIR}/grafana-server.pid
    \
        --packaging=rpm
    \
        cfg:default.paths.logs=${LOG_DIR}
    \
        cfg:default.paths.data=${DATA_DIR}
    \
        cfg:default.paths.plugins=${PLUGINS_DIR}
    \
        cfg:default.paths.provisioning=${PROVISIONING_CFG_DIR}

LimitNOFILE=10000
TimeoutStopSec=20
CapabilityBoundingSet=
DeviceAllow=
LockPersonality=true
MemoryDenyWriteExecute=false
NoNewPrivileges=true
PrivateDevices=true
PrivateTmp=true
ProtectClock=true
ProtectControlGroups=true
ProtectHome=true
ProtectHostname=true
ProtectKernelLogs=true
ProtectKernelModules=true
ProtectKernelTunables=true
ProtectProc=invisible
ProtectSystem=full
RemoveIPC=true
RestrictAddressFamilies=AF_INET AF_INET6 AF_UNIX
RestrictNamespaces=true
RestrictRealtime=true
RestrictSUIDSGID=true
SystemCallArchitectures=native
UMask=0027

[Install]
WantedBy=multi-user.target
```

Appendix B.F Load Test Source Codes

B.F.1 Artillery Load Test Configuration File

```

config:
  # This is a test server run by team Artillery
  # It's designed to be highly scalable
  target: "http://grafana-1777174802.us-east-1.elb.amazonaws.com"
  phases:
    - duration: 50
      arrivalRate: 1
      name: "Stage 1"
    - pause: 30
    - duration: 50
      arrivalRate: 1
      name: "Stage 2"
    - pause: 60
    - duration: 60
      arrivalRate: 1
      name: "Stage 3"
    - pause: 30
    - duration: 60
      arrivalRate: 1
      name: "Stage 4"
    - pause: 60
    - duration: 50
      arrivalRate: 1
      name: "Stage 5"
    - pause: 60
    - duration: 50
      arrivalRate: 1
      name: "Stage 6"

  # Load & configure a couple of useful plugins
  # https://docs.artillery.io/reference/extensions
  plugins:
    ensure: {}
    apdex: {}
    metrics-by-endpoint: {}

  apdex:
    threshold: 100
  ensure:
    thresholds:
      - http.response_time.p99: 100
      - http.response_time.p95: 75
  scenarios:
    - flow:
        - loop:
            - get:
                url: "/dashboards"
            - get:
                url: "/d/f8742187-f440-4ee8-96cc-bad5af8edef1/air-quality-monitoring?orgId=1&var-measure_name=aqi&var-location_id=64294"
            count: 100

```

B.F.2 Instances Monitoring Script

```

1 import boto3
2 import csv
3 import time
4 from datetime import datetime, timedelta
5
6 # AWS Config
7 ec2_client = boto3.client("ec2") # EC2 Client
8 autoscaling_client = boto3.client("autoscaling") # Auto Scaling Group Client
9 cloudwatch_client = boto3.client("cloudwatch") # CloudWatch Client
10 asg_name = "grafana-v2" # Auto Scaling Group Name
11
12 # Store the previous states of the instances
13 previous_states = {}
14
15
16 def get_instance_states(ec2_client, instance_ids):
17     """
18         Retrieves the state of the instances
19
20     Args:
21         ec2_client (boto3.client): EC2 Client
22         instance_ids (list): List of instances IDs
23
24     Returns:
25         states (dict): Dictionary of instances IDs and their states
26     """
27     states = {}
28     response = ec2_client.describe_instances(InstanceIds=instance_ids)
29     for reservation in response["Reservations"]:
30         for instance in reservation["Instances"]:
31             states[instance["InstanceId"]] = instance["State"]["Name"]
32     return states
33
34
35 def get_instance_ids(asg_client, asg_name):
36     """
37         Retrieves the instance IDs of the instances in the Auto Scaling Group
38
39     Args:
40         asg_client (boto3.client): Auto Scaling Group Client
41         asg_name (str): Auto Scaling Group Name
42
43     Returns:
44         instance_ids (list): List of instances IDs
45     """
46     asg = asg_client.describe_auto_scaling_groups(AutoScalingGroupNames=[asg_name])
47     return [
48         instance["InstanceId"] for instance in asg["AutoScalingGroups"][0]["Instances"]
49     ]
50
51
52 def get_ram_usage(cloudwatch_client, instance_id):
53     """
54         Retrieves the average RAM usage of the instance
55
56     Args:
57         cloudwatch_client (boto3.client): CloudWatch Client
58         instance_id (str): Instance ID
59
60     Returns:
61         ram_usage (float): Average RAM usage
62     """
63     end_time = datetime.utcnow()
64     start_time = end_time - timedelta(minutes=3) # Period of 3 minutes
65     metric_data = cloudwatch_client.get_metric_statistics(
66         Namespace="CWAgent",

```

```

67         MetricName="mem_used_percent",
68         Dimensions=[{"Name": "InstanceId", "Value": instance_id}],
69         StartTime=start_time,
70         EndTime=end_time,
71         Period=180,
72         Statistics=["Average"],
73     )
74     if metric_data["Datapoints"]:
75         return metric_data["Datapoints"][0]["Average"]
76     return None
77
78
79 def get_cpu_usage(cloudwatch_client, instance_id):
80     """
81     Retrieves the average CPU usage of the instance
82
83     Args:
84         cloudwatch_client (boto3.client): CloudWatch Client
85         instance_id (str): Instance ID
86
87     Returns:
88         cpu_usage (float): Average CPU usage
89     """
90     end_time = datetime.utcnow()
91     start_time = end_time - timedelta(minutes=3) # Period of 3 minutes
92     metric_data = cloudwatch_client.get_metric_statistics(
93         Namespace="CWAgent",
94         MetricName="cpu_usage_user",
95         Dimensions=[{"Name": "InstanceId", "Value": instance_id}],
96         StartTime=start_time,
97         EndTime=end_time,
98         Period=180,
99         Statistics=["Average"],
100    )
101    if metric_data["Datapoints"]:
102        return metric_data["Datapoints"][0]["Average"]
103    return None
104
105
106 # Main loop
107 while True:
108     # Gets the instances IDs and their states in the Auto Scaling Group
109     instance_ids = get_instance_ids(
110         autoscaling_client, asg_name
111     ) # Gets the instances IDs
112     current_states = get_instance_states(
113         ec2_client, instance_ids
114     ) # Gets the instances states
115     # Write the metrics to the CSV file
116     with open("test/metrics.csv", "a", newline="") as file:
117         writer = csv.writer(file)
118         for instance_id in instance_ids:
119             print("Instance ID: ", instance_id)
120             ram_usage = get_ram_usage(
121                 cloudwatch_client, instance_id
122             ) # Gets the RAM usage
123             print("RAM usage: ", ram_usage)
124             cpu_usage = get_cpu_usage(
125                 cloudwatch_client, instance_id
126             ) # Gets the CPU usage
127             print("CPU usage: ", cpu_usage)
128             # Check if the state of the instance has changed
129             state_changed = (
130                 instance_id in previous_states
131                 and previous_states[instance_id] != current_states[instance_id]
132             )
133             # Update the previous state
134             previous_states[instance_id] = current_states[instance_id]
135             print("Current state: ", current_states[instance_id])
136             # Handle pending state

```

```
137         if instance_id not in previous_states:
138             current_states[instance_id] = "pending"
139             ram_usage = None
140             cpu_usage = None
141             # Write the metrics to the CSV file
142             if (
143                 current_states[instance_id] == "pending"
144                 or state_changed
145                 or (
146                     current_states[instance_id] == "running"
147                     and ram_usage is not None
148                     and cpu_usage is not None
149                 )
150             ):
151                 writer.writerow(
152                     [
153                         datetime.now().strftime("%Y-%m-%d %H:%M:%S"),
154                         instance_id,
155                         current_states[instance_id],
156                         ram_usage,
157                         cpu_usage,
158                     ]
159                 ) # Write the metrics to the CSV file
160                 file.flush() # Flush the buffer
161             # Wait 1 second
162             time.sleep(1)
```