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

List of Figures

List of Tables

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. It discusses the technical challenges encountered during the project and the innovative solutions developed to address them. Additionally, 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 Architecture of the Project

The project is divided into two main pipelines:

1. **Data Collecting, Processing & Storing Pipeline:** This pipeline is responsible for collecting, processing and storing the data from the IoT sensors in a time-series database.

2. **Data Distributing Pipeline:** This pipeline is responsible for distributing the data to the users through a Grafana dashboard and for monitoring the system.

The figure 2.1 below illustrates the project pipeline.

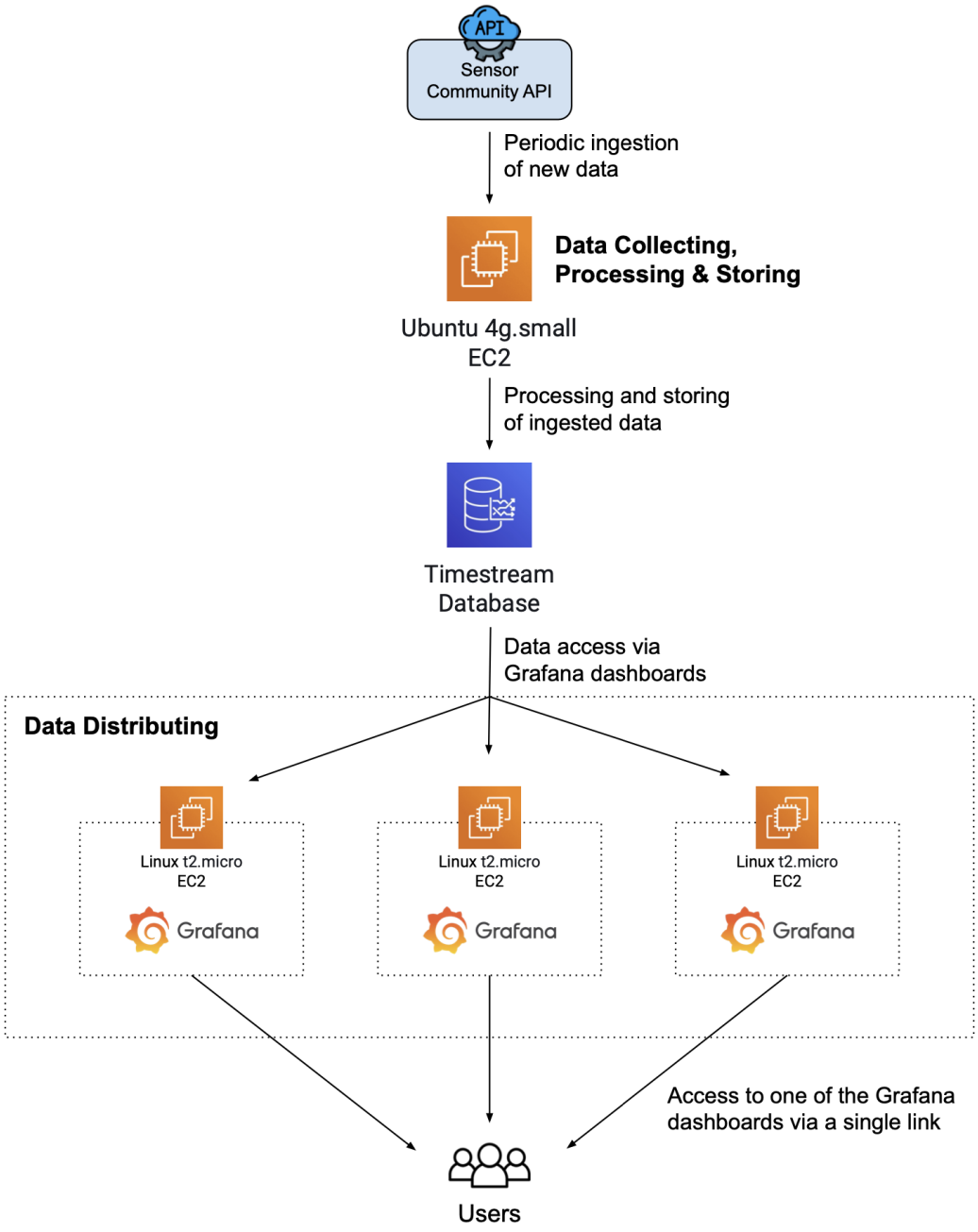


Figure 2.1: Project Architecture Diagram

2.2 Data Collecting, Processing & Storing

2.2.1 Overview of the first pipeline architecture

In the **Data Collecting** phase, the last set of data is obtained from its source, marking the start 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.

For a detailed visual representation of this pipeline, refer to Figure 2.2, which illustrates the Data Collecting, Processing, and Storing Pipeline.

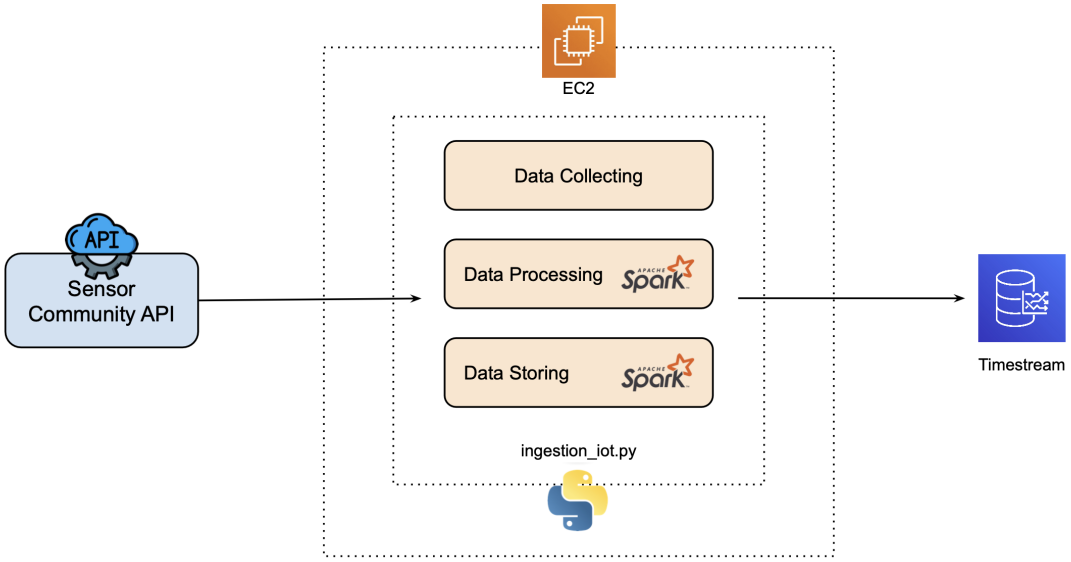


Figure 2.2: Data Collecting, Processing & Storing Pipeline Diagram

2.2.2 Data Collecting

2.2.2.1 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.

2. **24 Hour Averaged Data API:** This API provides data averaged over the last 24 hours for each sensor.

2.2.2.2 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 10 seconds, ensuring a regular and up-to-date flow of data from the sensors.

2.2.3 Data Processing

2.2.3.1 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 ($\mu g/m^3$)**	**PM_{10} Particles, 24 hour mean ($\mu g/m^3$)**
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

2.2.3.2 Leveraging Apache Spark

Apache Spark is a distributed computing framework that excels at managing and processing large-scale data sets. Using Apache Spark improves data processing capabilities in several ways:

1. **Scalability for Varying Data Volumes:** Apache Spark's scalable architecture is optimally designed to handle the varying volumes and complexity of IoT sensor data. This scalability is crucial for real-time data processing, ensuring fast and efficient AQI calculations.

2. **Parallel processing between nodes:** The distributed nature of Spark enables parallel processing of computational tasks across multiple nodes. This functionality is essential for the simultaneous processing of AQI calculations from a range of sensors, significantly increasing the speed and efficiency of data processing.

2.2.4 Data Storing

2.2.4.1 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.

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.

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.

2.2.4.2 Storing Script

The data storing process represents the final phase in the IoT data collecting and processing pipeline. Once the 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 B.D 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.

2.2.5 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:

1. **EC2 Instance Configuration:** A t4g.small EC2 instance with Ubuntu 2.3 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. In addition, Apache Spark has been installed on this instance to process the data collected.

2. **Setting Up Automated Services:** Two systemd services, which are activated each time the instance is started, were then set up to automate the following tasks:

    - **get_iam_credentials.service:** This service (See Appendix B.E.2.1) executes `get_iam_credentials.sh` (See Appendix B.E.1.1). This script retrieves the instance's IAM credentials, enabling secure integration with other AWS services.
    - **spark_python_job.service:** This service (See Appendix B.E.2.2) launches `start_spark_job.sh` (See Appendix B.E.1.2), which starts the script responsible for the collection, processing and storage of IoT data (See Appendix B.A).

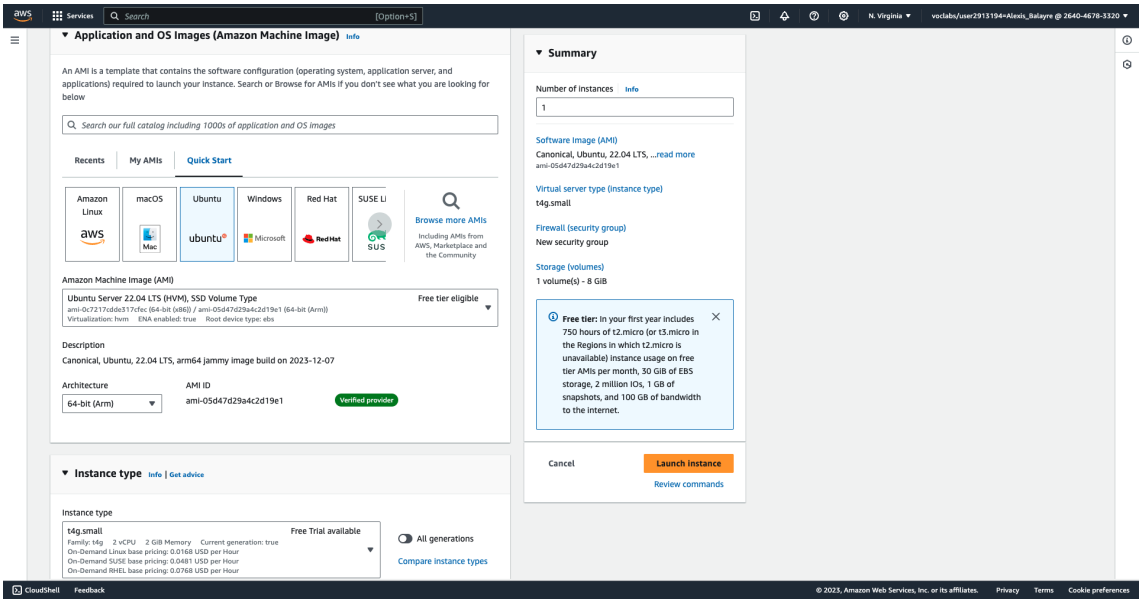


Figure 2.3: EC2 Instance Configuration Screenshot

2.3 Data Distributing

2.3.1 Overview of the second pipeline architecture

The second pipeline of this project focuses on the efficient distribution and monitoring of processed data, comprising several essential components:

1. **Internet Gateway**:
   This is the entry point for user interaction, directing them to the Grafana dashboards through specific URLs.

2. **Load Balancer**:
   Central to this pipeline, the Load Balancer evenly distributes the workload across EC2 instances.

3. **Grafana Dashboard Access**:
   Hosted on EC2 instances, the Grafana dashboard provides an interactive platform for data visualization.

4. **CloudWatch Monitoring**:
   AWS CloudWatch actively monitors system performance and dynamically adjusts resources to maintain system efficiency and stability.

For a comprehensive visual overview of this pipeline, please refer to Figure 2.4, which depicts the Data Distributing Pipeline.

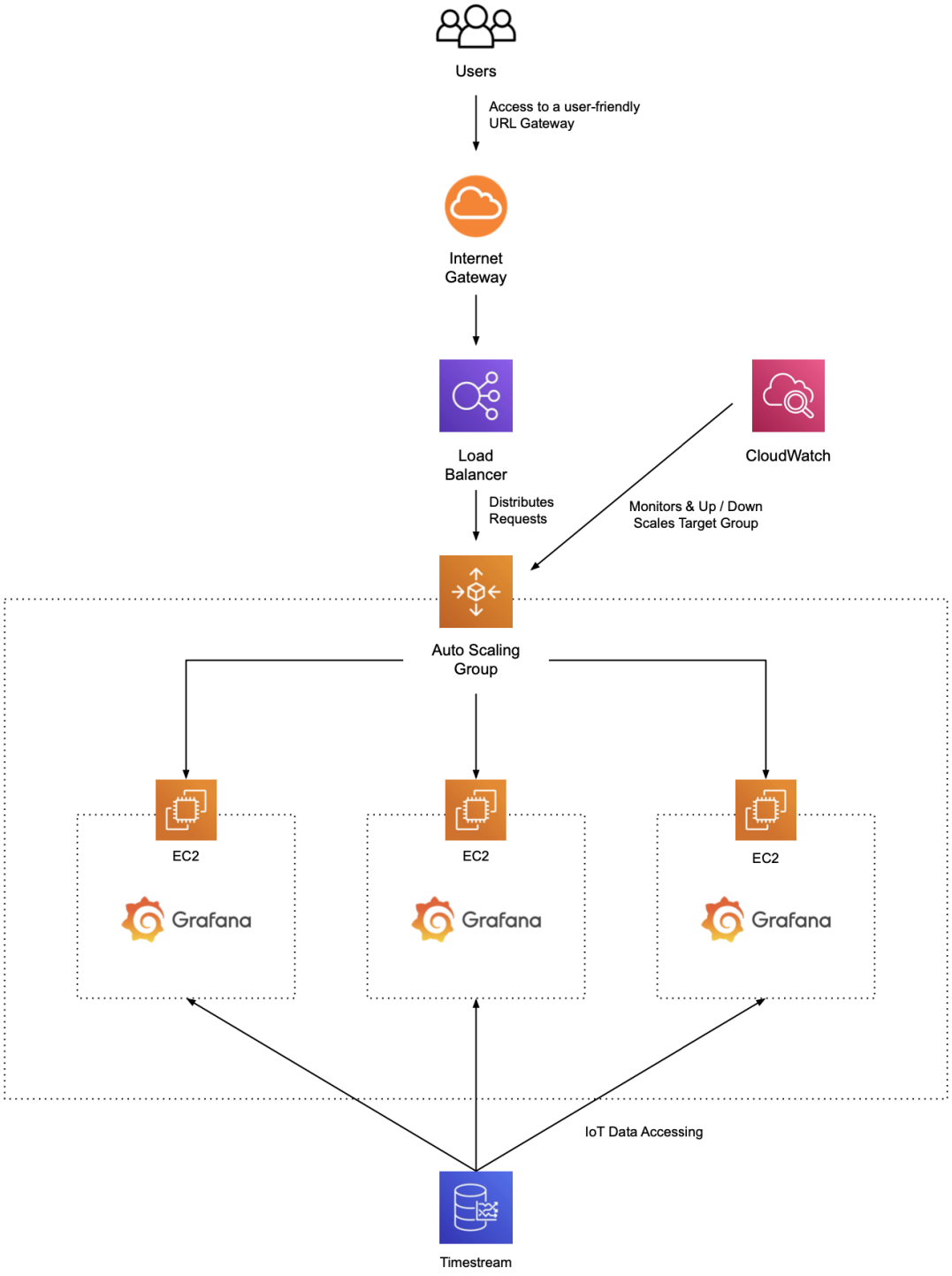


Figure 2.4: Data Distributing Pipeline Diagram

2.3.2 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 2.5 illustrates the Auto Scaling Group Pipeline.

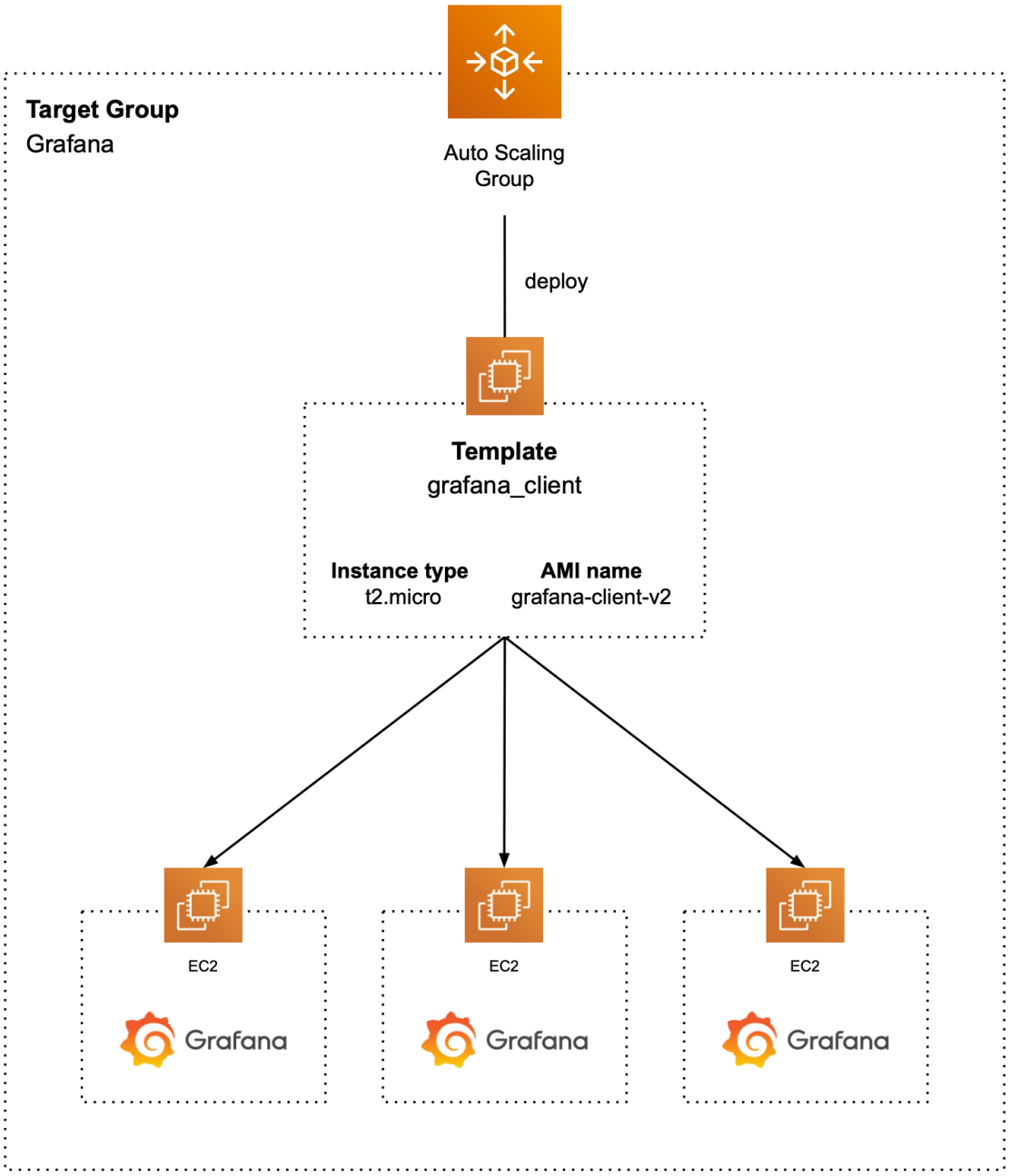


Figure 2.5: Auto Scaling Group Pipeline Diagram

2.3.2.1 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. A Grafana server was configured on the AMI to display the IoT data in a user-friendly format. The AMI was then stored in the AWS Elastic Block Store (EBS) for future use.

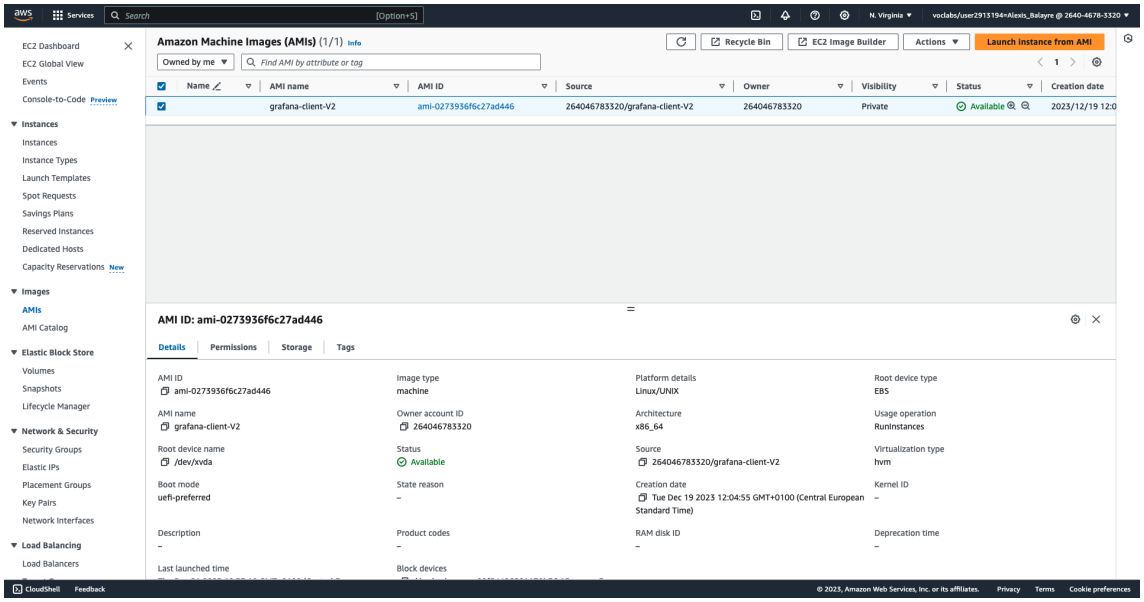


Figure 2.6: AMI Settings Screenshot

2.3.2.2 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.

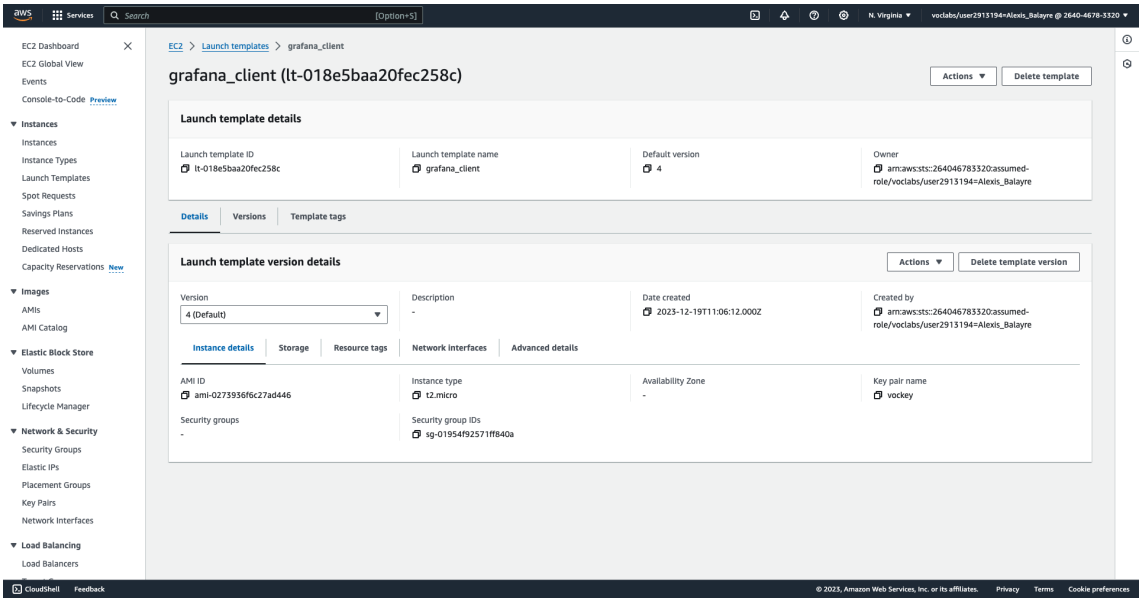


Figure 2.7: Launch Template Settings Screenshot

2.3.2.3 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.

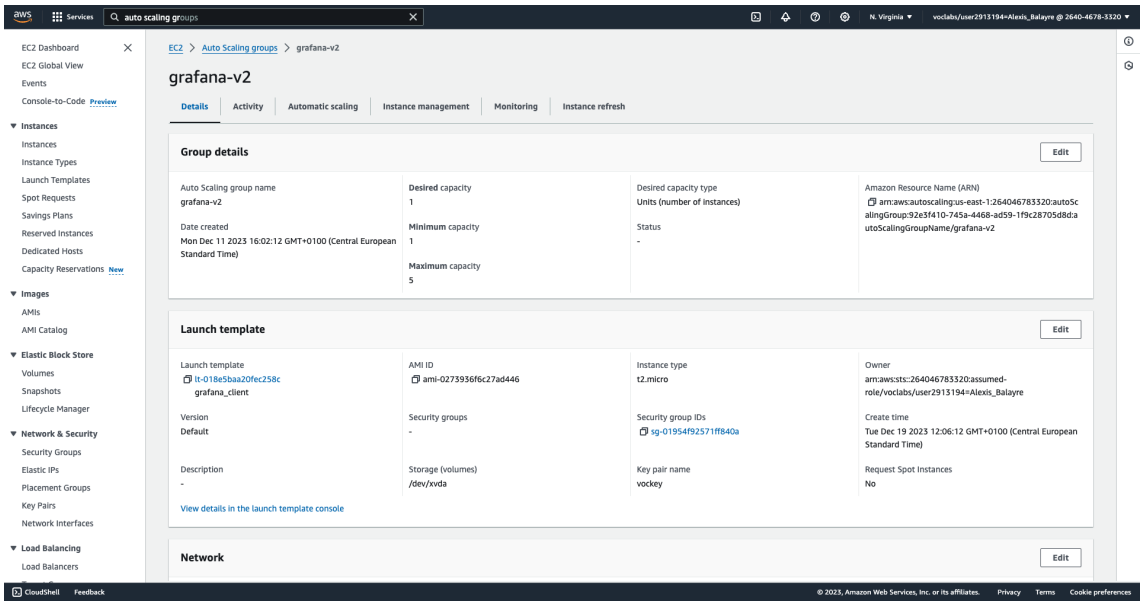


Figure 2.8: Auto Scaling Group Settings Screenshot

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.
- **Cooldown Period:** Set at 150 seconds to maintain stability by preventing frequent scaling.

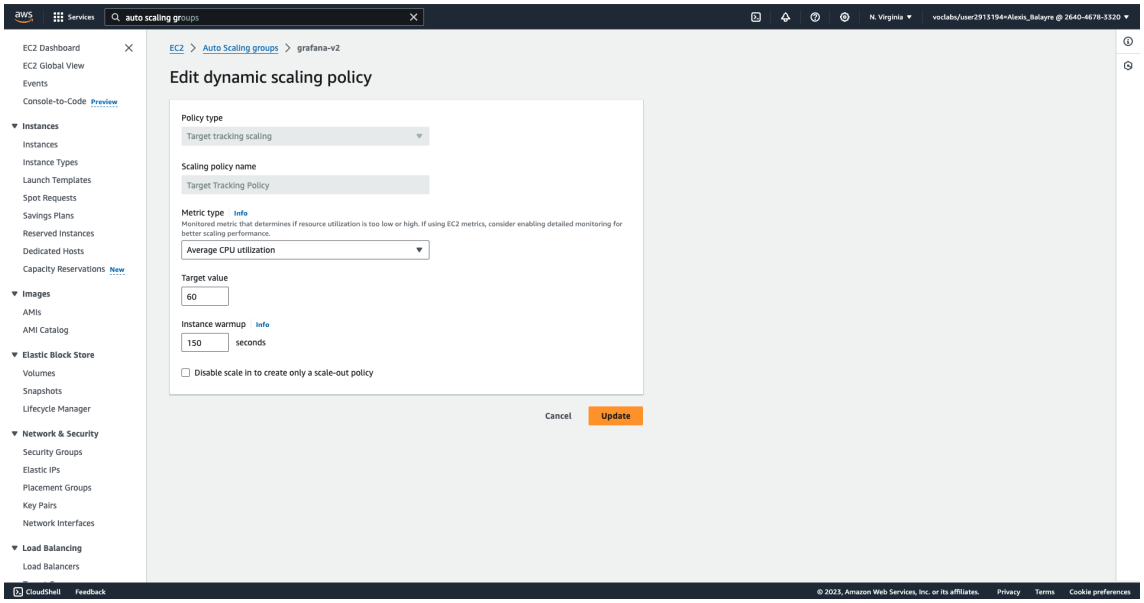


Figure 2.9: Scaling Policy Settings Screenshot

2.3.2.4 Target Group Configuration

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

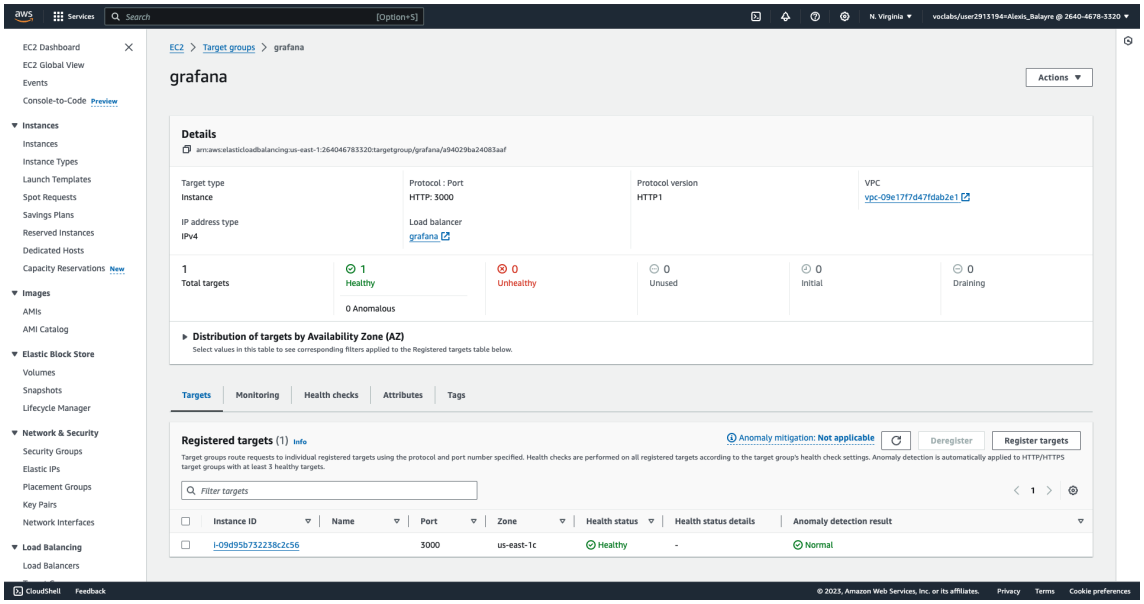


Figure 2.10: Target Group Settings Screenshot

2.3.2.5 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.

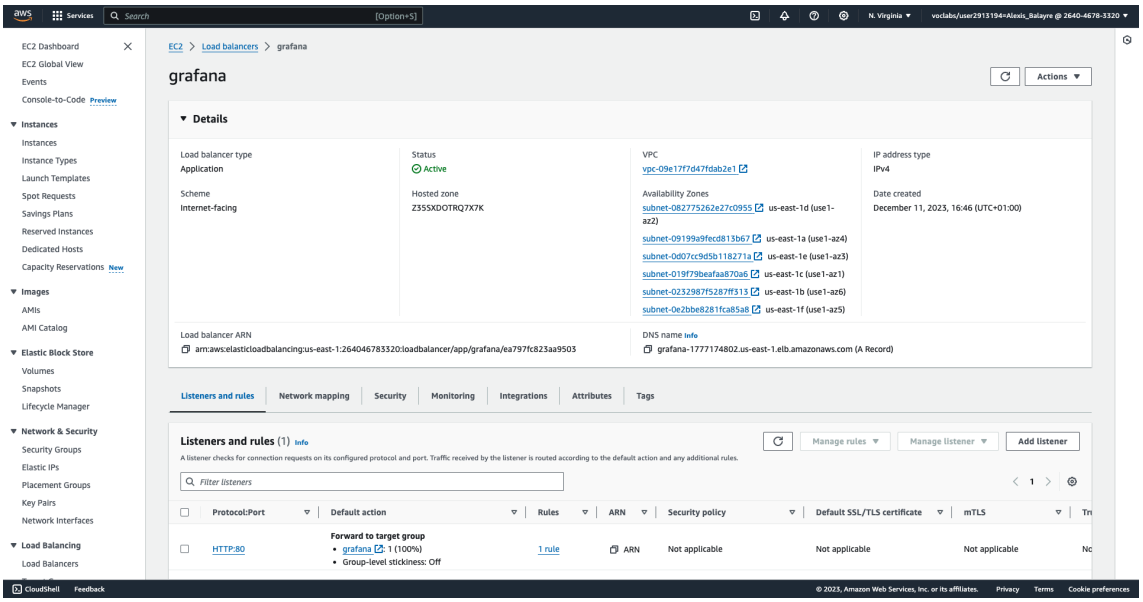


Figure 2.11: Load Balancer Settings Screenshot

Chapter 3

Results & Discussion

3.1 Results

3.1.1 Accessing the Data - Grafana Dashboard

The Grafana dashboard provides an interactive interface enabling users to analyse air quality data. It is accessible at the following URL: Air Quality Monitoring Dashboard. 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. This panel can be seen in Figure 3.1.

2. **Max AQI:** This panel shows the maximum AQI value in the selected location over the selected time range. This panel can be seen in Figure 3.1.

3. **Mean AQI:** This panel shows the mean AQI value in the selected location over the selected time range. This panel can be seen in Figure 3.1.

4. **AQI Map:** This panel shows the AQI value of each location on a map. This panel can be seen in Figure 3.2.

5. **IoT Data:** This panel shows all the data collected by the IoT sensors related to the selected measure value and time range. This panel can be seen in Figure 3.2.

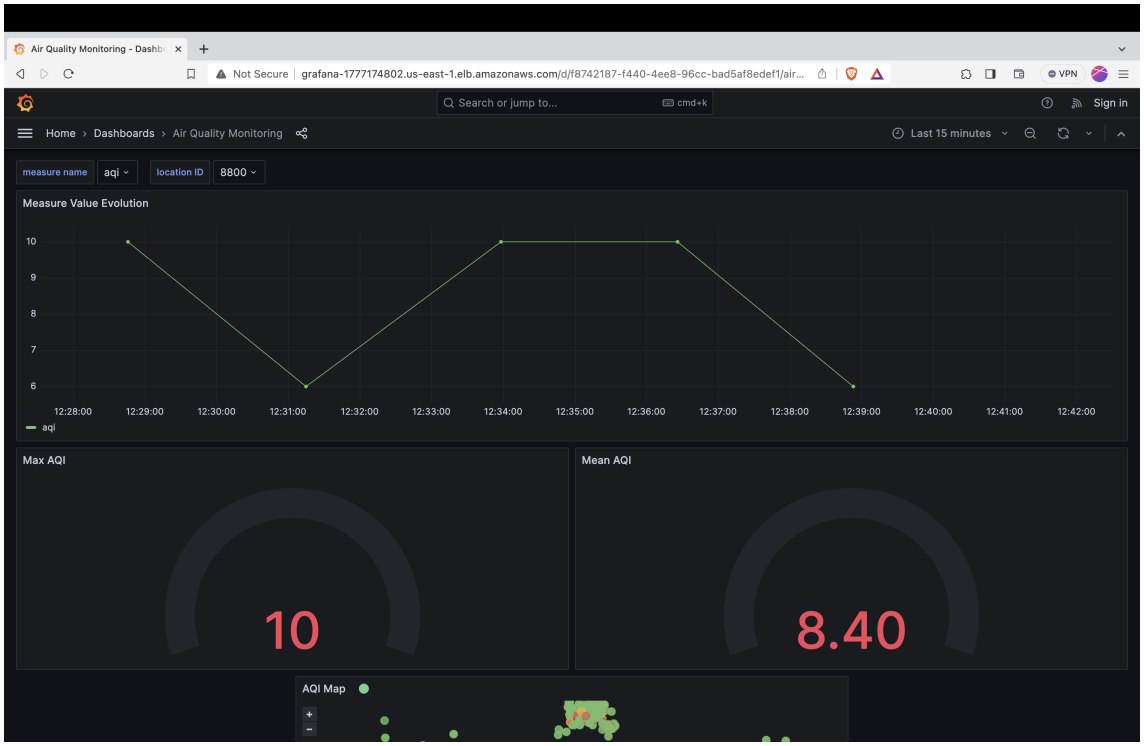
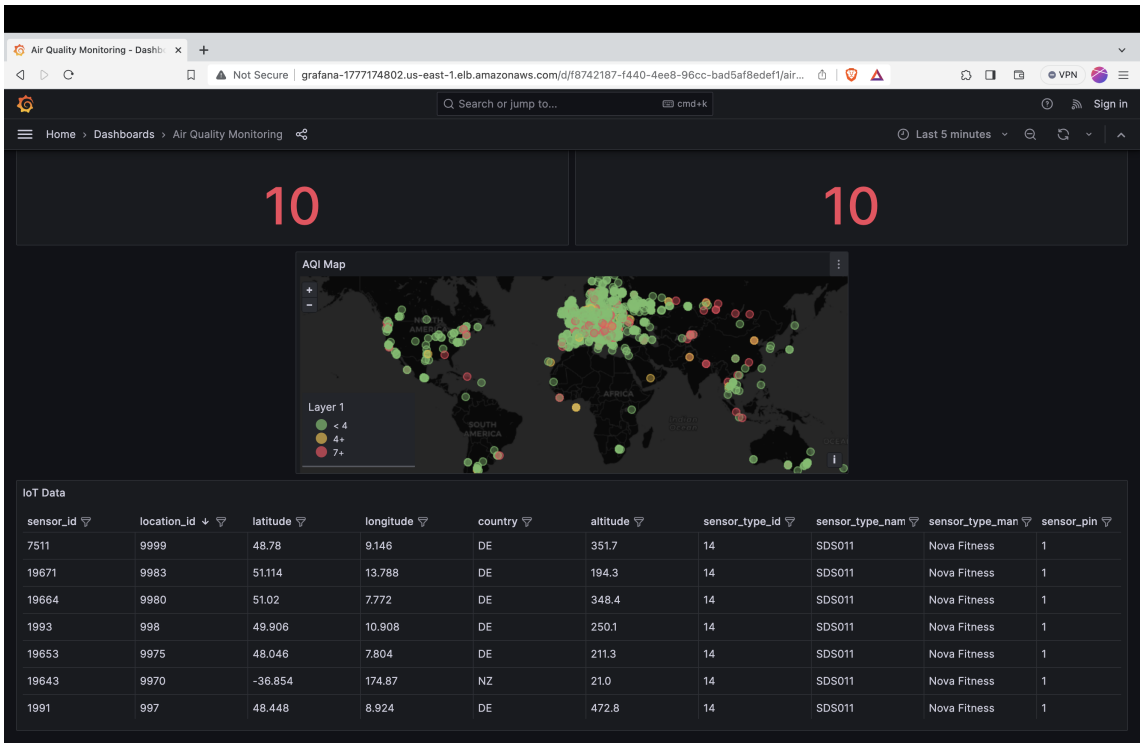


Figure 3.1: Grafana Dashboard: Measure Value Evolution, Max & Mean AQI



sensor_id	location_id	latitude	longitude	country	altitude	sensor_type_id	sensor_type_nam	sensor_type_man	sensor_pin
7511	9999	48.78	9.146	DE	351.7	14	SDS011	Nova Fitness	1
19671	9983	51.114	13.788	DE	194.3	14	SDS011	Nova Fitness	1
19664	9980	51.02	7.772	DE	348.4	14	SDS011	Nova Fitness	1
1993	998	49.906	10.908	DE	250.1	14	SDS011	Nova Fitness	1
19653	9975	48.046	7.804	DE	211.3	14	SDS011	Nova Fitness	1
19643	9970	-36.854	174.87	NZ	21.0	14	SDS011	Nova Fitness	1
1991	997	48.448	8.924	DE	472.8	14	SDS011	Nova Fitness	1

Figure 3.2: Grafana Dashboard: AQI Map & IoT Data

As shown in Figure 3.3, the AQI Map panel displays the AQI value of each location on a map. The AQI value is represented by a colour, depending to the range it belongs to. The AQI value is also displayed when hovering over a location.

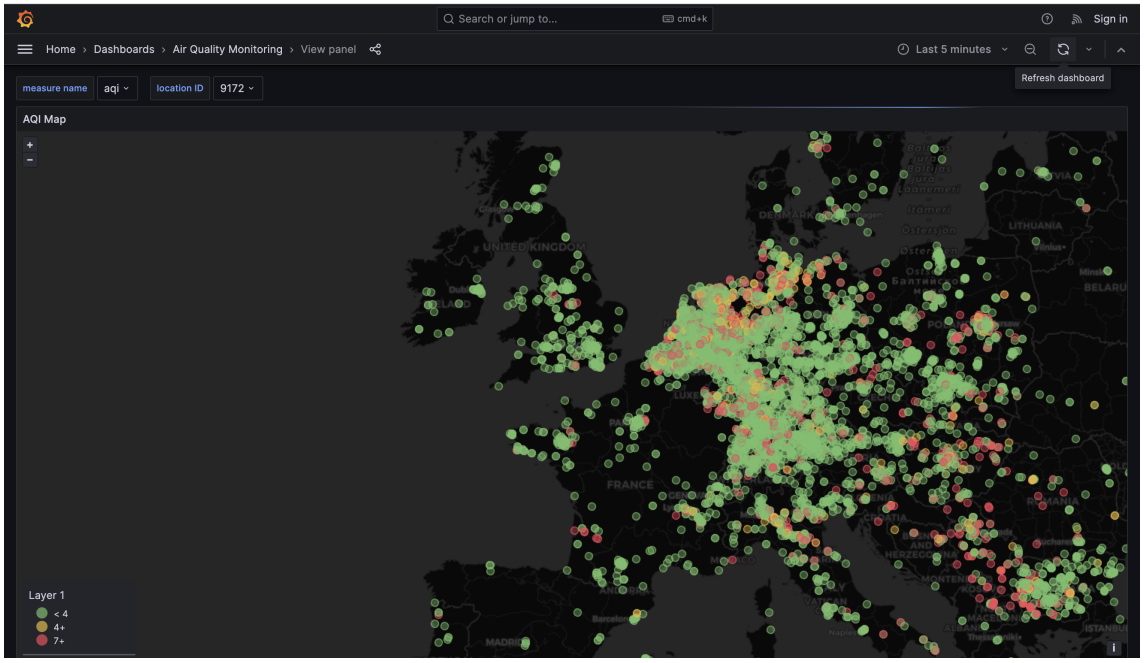


Figure 3.3: Grafana Dashboard AQI Map Panel

As required, the Grafana dashboard enables users to customise the time interval, as shown in Figure 3.4.

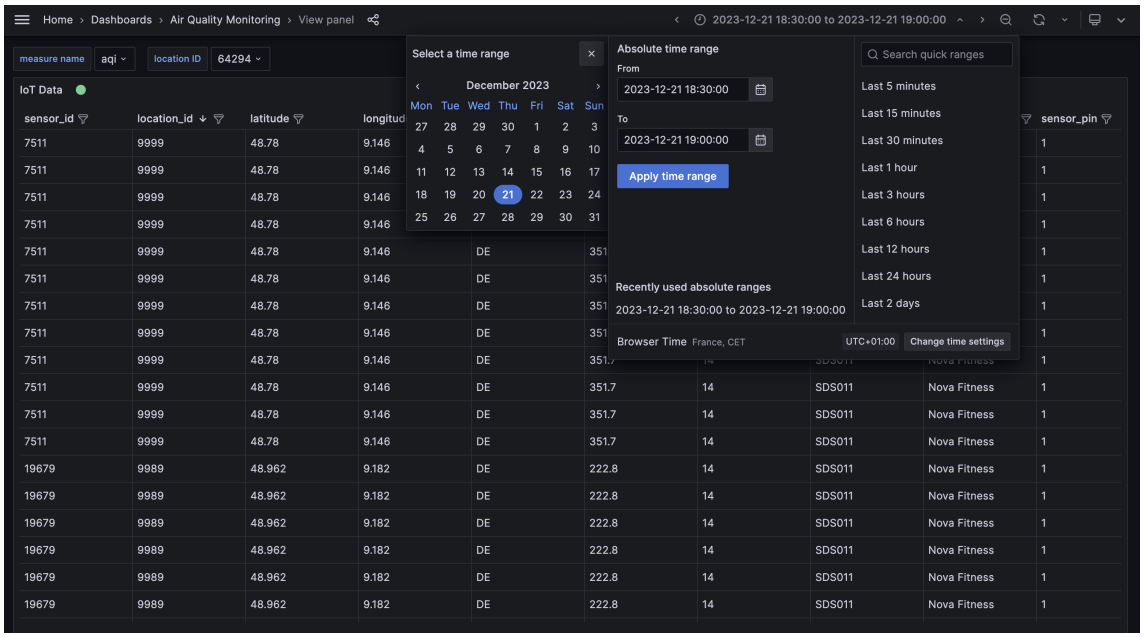


Figure 3.4: Grafana Dashboard IoT Data Panel - Time Range Choice

It is also possible to choose the measure value to display in the IoT Data panel, as shown in Figure 3.5.

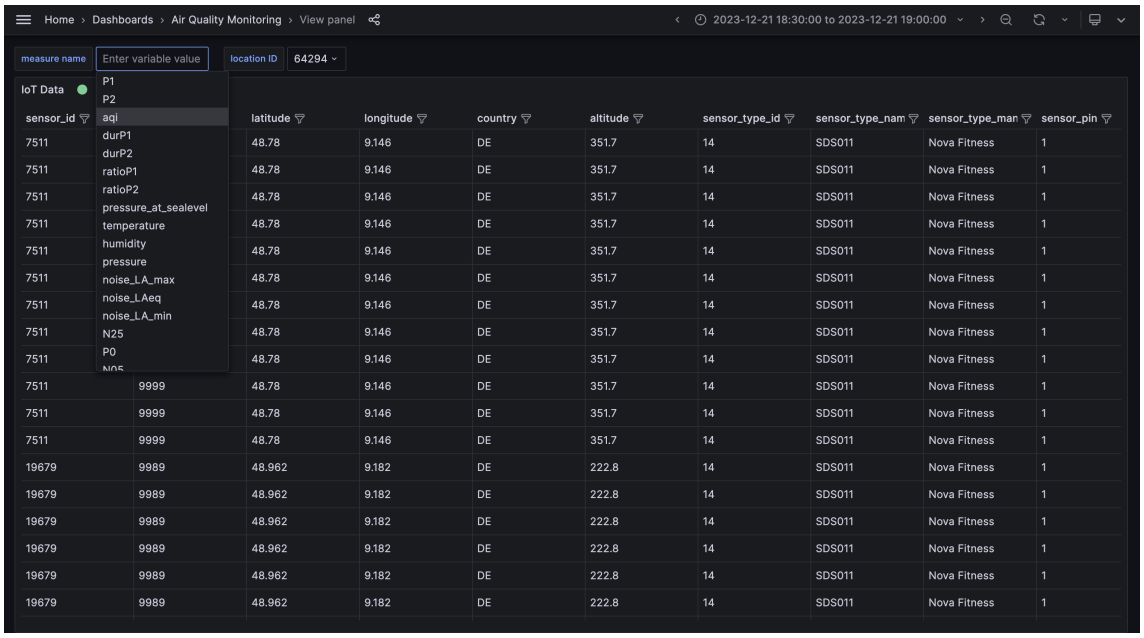


Figure 3.5: Grafana Dashboard IoT Data Panel - Measure Value Choice

Moreove, it is possible to apply a filter to the data displayed in the IoT Data panel. This can be done by clicking on the funnel icon, as shown in Figure 3.6.

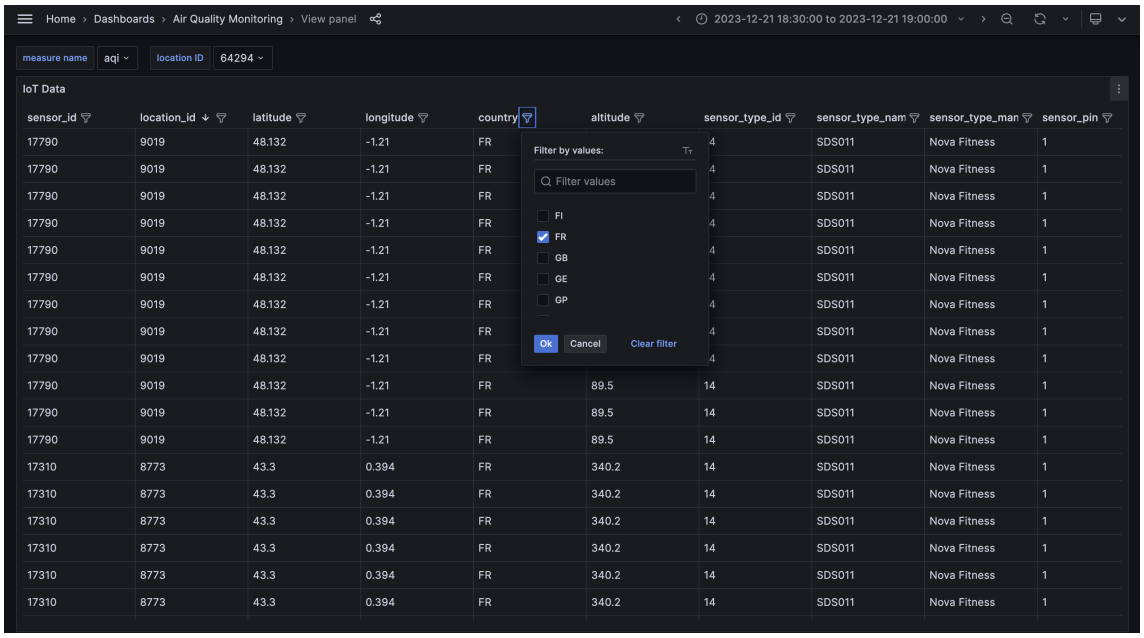


Figure 3.6: Grafana Dashboard IoT Data Panel - Apply a Filter

In addition, it is possible to export the data displayed in the IoT Data panel to a CSV file. This can be done by clicking on the `Download CSV` button, as shown in Figure 3.7.

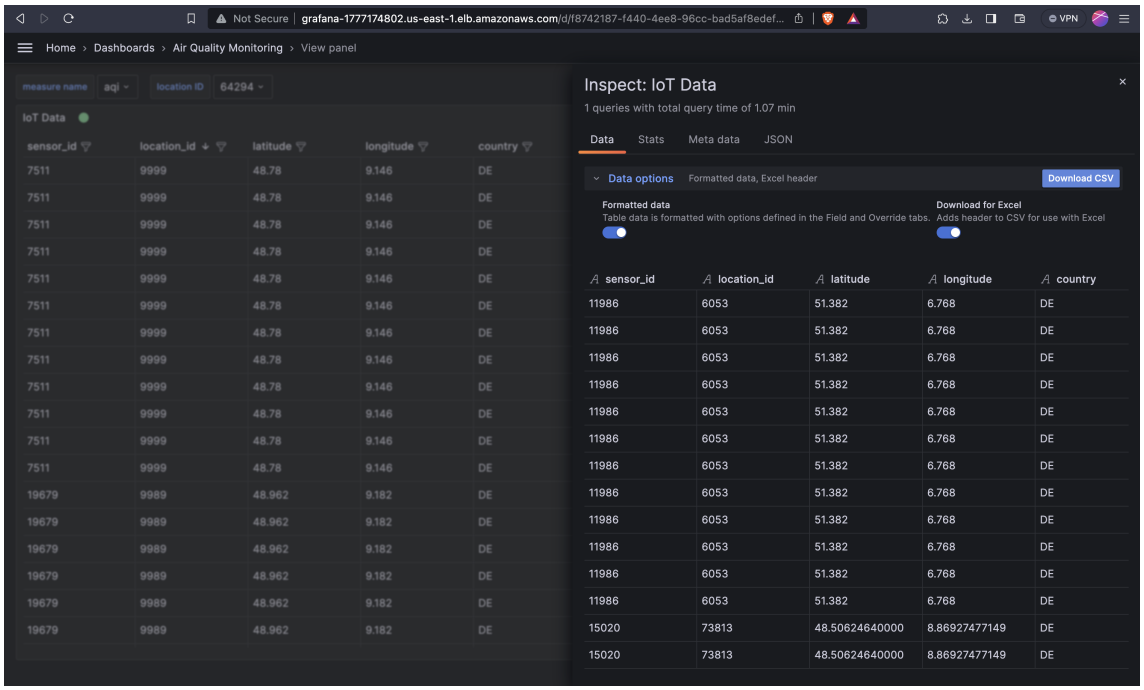


Figure 3.7: Grafana Dashboard IoT Data Panel - Export Values

3.1.2 Load Test

In order to test the scalability of the system, a load test was performed using the Artillery load testing tool. 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.

3.1.2.1 Artillery Load Test

The load test was performed using the Artillery load testing script available in Appendix B.F.1. Here are the steps of the load test:

1. **Stage 1:** 1 new user is added every second for 50 seconds.
2. **Pause:** No new users are added for 30 seconds.
3. **Stage 2:** 1 new user is added every second for 60 seconds.
4. **Pause:** No new users are added for 60 seconds.
5. **Stage 3:** 1 new user is added every second for 60 seconds.
6. **Pause:** No new users are added for 30 seconds.
7. **Stage 4:** 2 new users are added every second for 50 seconds.
8. **Pause:** No new users are added for 60 seconds.
9. **Stage 5:** 2 new users are added every second for 60 seconds.
10. **Pause:** No new users are added for 60 seconds.
11. **Stage 6:** 2 new users are added every second for 80 seconds.

Each user is configured to access 2 endpoints of the Grafana dashboard 100 times. Besides, this test was performed several times to ensure the reliability of the results.

3.1.2.2 Auto Scaling Group Behaviour Monitoring

The Auto Scaling Group behaviour was monitored during the load test thanks to a Python script available in Appendix B.F.2. The monitoring data is available in the `metrics.csv` file in the `test` folder. For enhanced data interpretation and visualisation, a Jupyter Notebook was developed to provide a graphical representation of system performance during load testing, as shown in Figure 3.8, which illustrates the dynamic scaling actions triggered by fluctuating demand on system resources.

At the start of the load test, only one instance was active. The data shows that this instance quickly reached its processing capacity. In response, the auto-scaling group launched additional instances, as evidenced by the subsequent reductions in CPU usage after each new instance was integrated into the service architecture. Once demand calmed and CPU utilisation fell below the 42% threshold, the Auto Scaling Group effectively deprovisioned the excess instances. This strategic downscaling is essential to

optimise resource utilisation and cost management. The figure highlights the elasticity of the system, with the Auto Scaling Group skilfully modulating the number of active instances according to the real-time workload, maintaining system stability and performance efficiency.

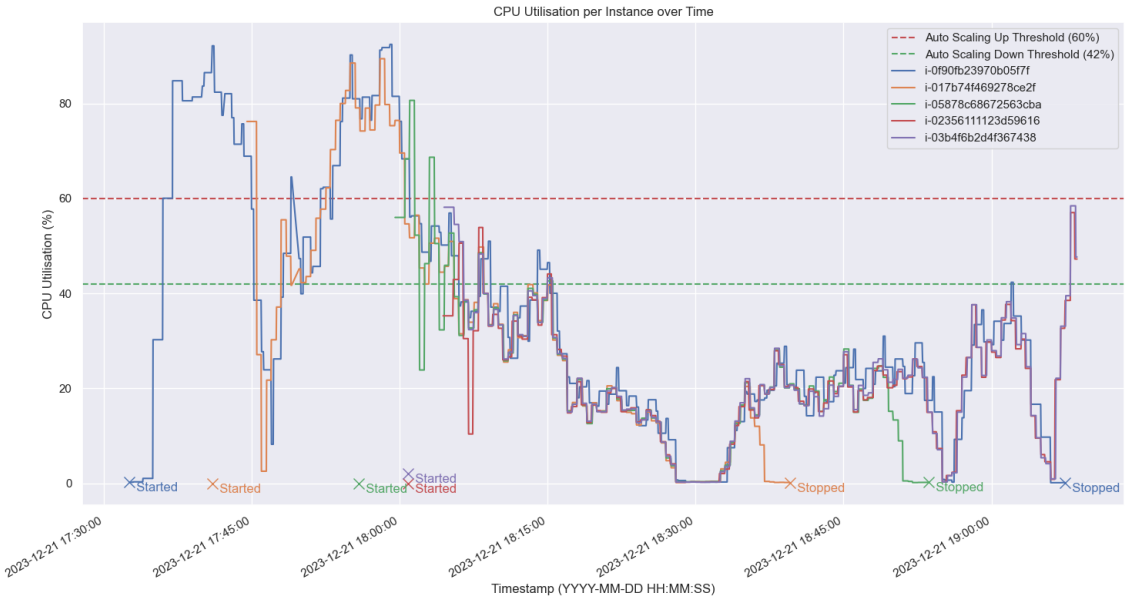


Figure 3.8: Load Test Result

3.2 Data Security and Sovereignty Considerations

Data security and sovereignty are critical elements in managing environmental sensor data, reflecting the importance of addressing evolving threats and protection mechanisms in IoT as highlighted in recent research.

3.2.1 Data Security

Data security for environmental sensors involves protecting sensitive data from unauthorised access, alteration, theft or destruction. The integration of IoT devices into wider networks introduces unique vulnerabilities. It is imperative to secure these devices against cyber threats, ensuring that the data they transmit and their operational integrity are protected. This includes protecting devices, their firmware and the networks to which they connect from unauthorised access and cyber attacks (2).

In addition to device and network security, managing the vast volumes of data generated by these sensors is crucial. The use of cloud-based solutions offers scalability and flexibility, but also raises significant security issues. The implementation of secure data transfer mechanisms, encrypted storage solutions and regular vulnerability assessments are essential to preserve data integrity and confidentiality. Compliance with international security standards and best practice ensures that data storage and processing meet the highest levels of security.

3.2.2 Data Sovereignty

Data sovereignty concerns the legal and regulatory aspects of data management, particularly with regard to the geographical location of data. The transnational nature of IoT deployments requires a thorough understanding of the various international and regional laws governing data privacy and protection. For example, the physical location of servers may subject data to the legal jurisdiction of that country, which affects the level of data protection and compliance obligations.

In addition, the transmission and processing of data across borders must comply with international data protection agreements, such as the General Data Protection Regulation (GDPR) in the European Union. This includes ensuring that data flows comply with the legal requirements of all countries involved.

Organisations need to navigate these complex legal landscapes while maintaining data accessibility and control. In cases where data crosses international borders, it is essential to understand who has jurisdiction over the data and under what conditions it can be accessed or shared. This is particularly important in situations involving judicial investigations or government requests for access to data.

Chapter 4

Conclusion

To conclude, this project successfully demonstrated the integration of cloud computing and IoT technologies to create a robust real-time environmental monitoring system, focusing on air quality. The implementation of a scalable data processing pipeline using Apache Spark and Amazon Timestream enabled the efficient management and analysis of large volumes of sensor data, ensuring accurate real-time AQI calculations. The development of a user-friendly Grafana dashboard facilitated data visualisation and accessibility, making the system invaluable to stakeholders in environmental monitoring, public health and urban planning. Load tests confirmed the system's scalability and responsiveness, underlining its ability to adapt dynamically to varying demands. Compliance with data security and sovereignty standards has ensured that the system complies with international regulations. This project not only demonstrates the practical application of combining cloud computing and IoT in environmental data management, but also lays the foundations for future advances in this vital area.

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
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, partionned_df)`: Function for writing the DataFrame to the Timestream database.

Appendix B

Source Codes

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

```python
import findspark

findspark.init()  # Initializing Spark

from pyspark.sql import SparkSession

import datetime as dt
import time

from lib.collecting import fetch_sensors_data
from lib.processing import computeAQI
from lib.storing import keepOnlyUpdatedRows, writeToTimestream

if __name__ == "__main__":
    # Define the Timestream database and table names
    DATABASE_NAME = "iot_project"
    TABLE_NAME = "iot_table"

    # Initializing Spark Session
    sparkSession = (
        SparkSession.builder.appName("Cloud Computing Project")
        .master("local[*]")
        .config("spark.sql.inMemoryColumnarStorage.compressed", "true")
        .config("spark.sql.inMemoryColumnarStorage.batchSize", "10000")
        .config("spark.serializer", "org.apache.spark.serializer.KryoSerializer")
        .config("spark.ui.enabled", "true")
        .config("spark.io.compression.codec", "snappy")
        .config("spark.rdd.compress", "true")
        .getOrCreate()
    )

    while True:
        try:
            print(
                dt.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
                + " Starting the pipeline..."
            )
            # Fetch the data from the sensors
            iotDfRaw = fetch_sensors_data(sparkSession)

            # Compute the AQI for each sensor
            iotDfFormatted = computeAQI(iotDfRaw)

            # Filter the data to keep only the updated rows
```

```
            dataFiltered = keepOnlyUpdatedRows(
                DATABASE_NAME, TABLE_NAME, iotDfFormatted
            )

            # Write the data to Timestream
            print(
                dt.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
                + " 4. Writing the data to Timestream..."
            )
            dataFiltered.foreachPartition(
                lambda partition: writeToTimestream(
                    DATABASE_NAME, TABLE_NAME, partition
                )
            )
            print(
                dt.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
                + " Done writing the data to Timestream.\n"
            )

            # Sleep for 10 seconds
            print(
                dt.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
                + " Done with the pipeline. Waiting for 10 seconds.\n"
            )
            time.sleep(10)
        except Exception as e:
            print(f"Exception: {e}")
```

Appendix B.B Data Collecting Source Code

```python
# collecting.py
# The first step of the pipeline

from requests import Session
import datetime as dt


def fetch_sensors_data(sparkSession):
    """
    Fetches the latest data from the sensors and returns it as a Spark DataFrame

    Args:
        sparkSession (SparkSession): The SparkSession instance

    Returns:
        df (DataFrame): The DataFrame containing the last data from the sensors
    """

    # Fetches the latest data from the data.sensor.community API
    url = "https://data.sensor.community/static/v2/data.24h.json"
    # Use a session to avoid creating a new connection for each request
    session = Session()
    try:
        print(
            dt.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
            + " 1. Fetching the latest data..."
        )
        response = session.get(url)
        # If the response was successful, no Exception will be raised
        if response.status_code == 200 and response.content:
            # Convert the response to a Spark DataFrame
            df = sparkSession.read.option("multiline", "true").json(
                sparkSession.sparkContext.parallelize([response.text])
            )
            print(
                dt.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
                + " Done fetching the latest data.\n"
            )
            return df
    except Exception as e:
        print(f"Request failed with exception {e}")
    finally:
        session.close()
    return None
```

Appendix B.C Data Processing Source Code

```python
# collecting.py
# The second step of the pipeline

from pyspark.sql.types import FloatType, IntegerType
import pyspark.sql.functions as F
import datetime as dt


# Defining a UDF to compute the AQI value for PM2.5
@F.udf(returnType=IntegerType())
def get_aqi_value_p25(value):
    """
    Computes the AQI value for PM2.5

    Args:
        value (float): The value of PM2.5
    Returns:
        aqi (int): The AQI value
    """

    if value is None:
        return None
    if 0 <= value <= 11:
        return 1
    elif 12 <= value <= 23:
        return 2
    elif 24 <= value <= 35:
        return 3
    elif 36 <= value <= 41:
        return 4
    elif 42 <= value <= 47:
        return 5
    elif 48 <= value <= 53:
        return 6
    elif 54 <= value <= 58:
        return 7
    elif 59 <= value <= 64:
        return 8
    elif 65 <= value <= 70:
        return 9
    return 10


# Defining a UDF to compute the AQI value for PM10
@F.udf(returnType=IntegerType())
def get_aqi_value_p10(value):
    """
    Computes the AQI value for PM10

    Args:
        value (float): The value of PM10

    Returns:
        aqi (int): The AQI value
    """

    if value is None:
        return None
    if 0 <= value <= 16:
        return 1
    elif 17 <= value <= 33:
        return 2
    elif 34 <= value <= 50:
        return 3
    elif 51 <= value <= 58:
        return 4
    elif 59 <= value <= 66:
```

```
        return 5
    elif 67 <= value <= 75:
        return 6
    elif 76 <= value <= 83:
        return 7
    elif 84 <= value <= 91:
        return 8
    elif 92 <= value <= 99:
        return 9
    return 10


def computeAQI(df):
    """
    Computes the AQI for each particulate matter sensor

    Args:
        df (DataFrame): The DataFrame containing the data from the sensors

    Returns:
        df_grouped (DataFrame): The DataFrame containing the AQI for each sensor
    """

    print(dt.datetime.now().strftime("%Y-%m-%d %H:%M:%S") + " 2. Computing the AQI ...")
    df_exploded = df.withColumn(
        "sensordatavalue",
        F.explode("sensordatavalues"),  # Explode the sensordatavalues column
    ).withColumn(
        "aqi",
        F.when(
            F.col("sensordatavalue.value_type") == "P1",
            get_aqi_value_p25(
                F.col("sensordatavalue.value").cast(FloatType())
            ),  # Cast the value to float and compute the AQI of PM2.5
        ).when(
            F.col("sensordatavalue.value_type") == "P2",
            get_aqi_value_p10(
                F.col("sensordatavalue.value").cast(FloatType())
            ),  # Cast the value to float and compute the AQI of PM10
        ),
    )
    df_exploded.cache()  # Cache the DataFrame to avoid recomputing it
    df_grouped = (
        df_exploded.groupBy("sensor.id", "timestamp")  # Group by sensor and timestamp
        .agg(
            F.first("id").alias("id"),
            F.first("location").alias("location"),
            F.first("sensor").alias("sensor"),
            F.max("aqi").alias("aqi"),  # Compute the maximum AQI between PM2.5 and PM10
            F.collect_list("sensordatavalue").alias("sensordatavalues"),
        )  # Aggregate the AQI and the sensordatavalues
        .selectExpr(
            "sensor.id as sensor_id",
            "sensor.pin as sensor_pin",
            "sensor.sensor_type.id as sensor_type_id",
            "sensor.sensor_type.manufacturer as sensor_type_manufacturer",
            "sensor.sensor_type.name as sensor_type_name",
            "location.country as country",
            "location.latitude as latitude",
            "location.longitude as longitude",
            "location.altitude as altitude",
            "location.id as location_id",
            "aqi",
            "sensordatavalues",
            "timestamp",
        )  # Select the columns to keep
    )
```

```
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
        .\n"
138     )
139     return df_grouped
```

Appendix B.D Data Storing Source Code

```python
# storing.py
# The last step of the pipeline

from pyspark.sql.types import BooleanType
import pyspark.sql.functions as F
from pyspark.sql import Row
from botocore.config import Config
import boto3
import time
import datetime as dt


def keepOnlyUpdatedRows(database_name, table_name, df):
    """
    Verifies if the data is already stored in Timestream and keeps only the updated values

    Args:
        database_name (string): The name of the database
        table_name (string): The name of the table
        df (DataFrame): The DataFrame containing the data to be stored

    Returns:
        df_updated (DataFrame): The DataFrame containing only the updated rows
    """

    print(
        dt.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
        + " 3. Filtering the data to keep only the updated rows..."
    )
    query = """
        SELECT sensor_id, MAX(time) as last_timestamp
        FROM {}.{}
        GROUP BY sensor_id
    """.format(
        database_name, table_name
    )

    # Initialize the boto3 client
    session = boto3.Session()  # Create a boto3 session
    query_client = session.client(
        "timestream-query", config=Config(region_name="us-east-1")
    )  # Create a boto3 client
    paginator = query_client.get_paginator("query")  # Create a paginator

    # Get the last timestamp for each sensor
    last_timestamps = (
        {}
    )  # Initialize a dictionary to store the last timestamp for each sensor
    response_iterator = paginator.paginate(QueryString=query)  # Paginate the query
    for response in response_iterator:
        for row in response["Rows"]:
            sensor_id = row["Data"][0]["ScalarValue"]
            last_timestamps[sensor_id] = row["Data"][1]["ScalarValue"]

    # If there is no data in Timestream, return the DataFrame as is
    if len(last_timestamps) == 0:
        print("No data in Timestream")
        return df

    # Define an UDF to check if the row is updated
    @F.udf(returnType=BooleanType())
    def isUpdated(sensor_id, timestamp):
        """
        Checks if the row is updated

        Args:
```

```
            sensor_id (string): The sensor ID
            timestamp (string): The timestamp of the row

        Returns:
            isUpdated (boolean): True if the row is updated, False otherwise
        """

        if str(sensor_id) not in last_timestamps:
            return True
        current_timestamp = dt.datetime.strptime(timestamp, "%Y-%m-%d %H:%M:%S")
        last_timestamp_micro = last_timestamps[str(sensor_id)][
            :26
        ]  # Keep only up to microseconds
        last_sensor_timestamp = dt.datetime.strptime(
            last_timestamp_micro, "%Y-%m-%d %H:%M:%S.%f"
        )
        return (
            current_timestamp > last_sensor_timestamp
        )  # Return True if the row is updated

    df_updated = df.filter(
        isUpdated("sensor_id", "timestamp")
    )  # Filter the DataFrame to keep only the updated rows
    print(
        dt.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
        + " Done filtering the data to keep only the updated rows.\n"
    )
    return df_updated


def _print_rejected_records_exceptions(err):
    """
    Prints the rejected records exceptions

    Args:
        err (RejectedRecordsException): The RejectedRecordsException
    """

    print("RejectedRecords: ", err)
    for rr in err.response["RejectedRecords"]:
        print("Rejected Index " + str(rr["RecordIndex"]) + ": " + rr["Reason"])
        if "ExistingVersion" in rr:
            print("Rejected record existing version: ", rr["ExistingVersion"])


def write_records(database_name, table_name, client, records):
    """
    Helper function to write records to Timestream

    Args:
        database_name (string): The name of the database
        table_name (string): The name of the table
        client (TimestreamWriteClient): The TimestreamWriteClient
        records (list): The list of records to write
    """
    try:
        result = client.write_records(
            DatabaseName=database_name,
            TableName=table_name,
            CommonAttributes={},
            Records=records,
        )
        print(
            "WriteRecords Status: [%s]" % result["ResponseMetadata"]["
    HTTPStatusCode"]
        )
    except client.exceptions.RejectedRecordsException as err:
        _print_rejected_records_exceptions(err)
    except Exception as err:
        print("Error:", err)
```

```
136
137
138 def writeToTimestream(database_name, table_name, partionned_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         partionned_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 partionned_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),
```

```
                        "Type": "BIGINT",
                    }
                    measuresValues.append(aqi_measureValue)

            # Create a record for each sensor
            record = {
                "Dimensions": dimensions,
                "Time": row_timestamp,
                "TimeUnit": "MILLISECONDS",
                "MeasureName": "air_quality",
                "MeasureValueType": "MULTI",
                "MeasureValues": measuresValues,
            }
            records.append(record)

            # Write records to Timestream if there are 98 records
            if len(records) >= 98:
                write_records(
                    database_name, table_name, write_client, records
                )  # Write records to Timestream
                records = []  # Reset the records list
                time.sleep(1)  # Sleep for 1 second

        except Exception as e:
            print(f"Error processing row: {row}")
            print(f"Exception: {e}")

    # Write records to Timestream if there are any remaining records
    if len(records) > 100:
        while len(records) > 100:
            write_records(
                database_name, table_name, write_client, records[:99]
            )  # Write records to Timestream
            records = records[99:]  # Keep the remaining records
            time.sleep(1)  # Sleep for 1 second
    elif len(records) > 0:
        write_records(database_name, table_name, write_client, records)
```

Appendix B.E Scripts & Services Source Codes

B.E.1 Scripts

B.E.1.1 Get IAM Credentials Script

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
```

B.E.1.2 Start Spark Job Script

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
```

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: 50
      arrivalRate: 2
      name: "Stage 4"
    - pause: 60
    - duration: 60
      arrivalRate: 2
      name: "Stage 5"
    - pause: 60
    - duration: 80
      arrivalRate: 2
      name: "Stage 6"

  # Load & configure a couple of useful plugins
  # https://docs.art/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

```python
import boto3
import csv
import time
from datetime import datetime, timedelta

# AWS Config
ec2_client = boto3.client("ec2")  # EC2 Client
autoscaling_client = boto3.client("autoscaling")  # Auto Scaling Group Client
cloudwatch_client = boto3.client("cloudwatch")  # CloudWatch Client
asg_name = "grafana-v2"  # Auto Scaling Group Name

# Store the previous states of the instances
previous_states = {}


def get_instance_states(ec2_client, instance_ids):
    """
    Retrives the state of the instances

    Args:
        ec2_client (boto3.client): EC2 Client
        instance_ids (list): List of instances IDs

    Returns:
        states (dict): Dictionary of instances IDs and their states
    """
    states = {}
    response = ec2_client.describe_instances(InstanceIds=instance_ids)
    for reservation in response["Reservations"]:
        for instance in reservation["Instances"]:
            states[instance["InstanceId"]] = instance["State"]["Name"]
    return states


def get_instance_ids(asg_client, asg_name):
    """
    Retrives the instance IDs of the instances in the Auto Scaling Group

    Args:
        asg_client (boto3.client): Auto Scaling Group Client
        asg_name (str): Auto Scaling Group Name

    Returns:
        instance_ids (list): List of instances IDs
    """
    asg = asg_client.describe_auto_scaling_groups(AutoScalingGroupNames=[asg_name])
    return [
        instance["InstanceId"] for instance in asg["AutoScalingGroups"][0]["
    Instances"]
    ]


def get_ram_usage(cloudwatch_client, instance_id):
    """
    Retrives the average RAM usage of the instance

    Args:
        cloudwatch_client (boto3.client): CloudWatch Client
        instance_id (str): Instance ID

    Returns:
        ram_usage (float): Average RAM usage
    """
    end_time = datetime.utcnow()
    start_time = end_time - timedelta(minutes=3)  # Period of 3 minutes
    metric_data = cloudwatch_client.get_metric_statistics(
        Namespace="CWAgent",
```

```
        MetricName="mem_used_percent",
        Dimensions=[{"Name": "InstanceId", "Value": instance_id}],
        StartTime=start_time,
        EndTime=end_time,
        Period=180,
        Statistics=["Average"],
    )
    if metric_data["Datapoints"]:
        return metric_data["Datapoints"][0]["Average"]
    return None


def get_cpu_usage(cloudwatch_client, instance_id):
    """
    Retrives the average CPU usage of the instance

    Args:
        cloudwatch_client (boto3.client): CloudWatch Client
        instance_id (str): Instance ID

    Returns:
        cpu_usage (float): Average CPU usage
    """
    end_time = datetime.utcnow()
    start_time = end_time - timedelta(minutes=3)  # Period of 3 minutes
    metric_data = cloudwatch_client.get_metric_statistics(
        Namespace="CWAgent",
        MetricName="cpu_usage_user",
        Dimensions=[{"Name": "InstanceId", "Value": instance_id}],
        StartTime=start_time,
        EndTime=end_time,
        Period=180,
        Statistics=["Average"],
    )
    if metric_data["Datapoints"]:
        return metric_data["Datapoints"][0]["Average"]
    return None


# Main loop
while True:
    # Gets the instances IDs and their states in the Auto Scaling Group
    instance_ids = get_instance_ids(
        autoscaling_client, asg_name
    )  # Gets the instances IDs
    current_states = get_instance_states(
        ec2_client, instance_ids
    )  # Gets the instances states
    # Write the metrics to the CSV file
    with open("test/metrics.csv", "a", newline="") as file:
        writer = csv.writer(file)
        for instance_id in instance_ids:
            print("Instance ID: ", instance_id)
            ram_usage = get_ram_usage(
                cloudwatch_client, instance_id
            )  # Gets the RAM usage
            print("RAM usage: ", ram_usage)
            cpu_usage = get_cpu_usage(
                cloudwatch_client, instance_id
            )  # Gets the CPU usage
            print("CPU usage: ", cpu_usage)
            # Check if the state of the instance has changed
            state_changed = (
                instance_id in previous_states
                and previous_states[instance_id] != current_states[instance_id]
            )
            # Update the previous state
            previous_states[instance_id] = current_states[instance_id]
            print("Current state: ", current_states[instance_id])
            # 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)
```