



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  
2<sup>nd</sup> January 2024

# Table of Contents

<b>Table of Contents</b>	<b>ii</b>
<b>List of Figures</b>	<b>iv</b>
<b>List of Tables</b>	<b>v</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Methodologies</b>	<b>2</b>
2.1 Architecture of the Project . . . . .	2
2.2 Data Collecting, Processing & Storing . . . . .	4
2.2.1 Overview of the first pipeline architecture . . . . .	4
2.2.2 Data Collecting . . . . .	4
2.2.2.1 Data Source . . . . .	4
2.2.2.2 Ingestion Script . . . . .	5
2.2.3 Data Processing . . . . .	5
2.2.3.1 Processing Script . . . . .	5
2.2.3.2 Leveraging Apache Spark . . . . .	5
2.2.4 Data Storing . . . . .	6
2.2.4.1 Database Choice . . . . .	6
2.2.4.2 Storing Script . . . . .	6
2.2.5 Pipeline Implementation on AWS . . . . .	7
2.3 Data Distributing . . . . .	8
2.3.1 Overview of the second pipeline architecture . . . . .	8
2.3.2 Auto Scaling Group Configuration . . . . .	10
2.3.2.1 AMI Configuration . . . . .	11
2.3.2.2 Launch Template Configuration . . . . .	12
2.3.2.3 Auto Scaling Group Configuration . . . . .	13
2.3.2.4 Target Group Configuration . . . . .	14
2.3.2.5 Load Balancer Configuration . . . . .	15
<b>3 Results &amp; Discussion</b>	<b>16</b>
3.1 Results . . . . .	16
3.1.1 Accessing the Data - Grafana Dashboard . . . . .	16
3.1.2 Load Test . . . . .	21
3.1.2.1 Artillery Load Test . . . . .	21
3.1.2.2 Auto Scaling Group Behaviour Monitoring . . . . .	21

3.2	Data Security and Sovereignty Considerations . . . . .	23
3.2.1	Data Security . . . . .	23
3.2.2	Data Sovereignty . . . . .	23
<b>4</b>	<b>Conclusion</b>	<b>24</b>
<b>References</b>		<b>25</b>
<b>A</b>	<b>Documentation</b>	<b>26</b>
A.A	Project tree . . . . .	26
A.B	Getting Started . . . . .	26
A.C	Detailed Features of Functions . . . . .	27
<b>B</b>	<b>Source Codes</b>	<b>28</b>
B.A	Ingestion, Processing & Storing Pipeline Source Code . . . . .	28
B.B	Data Collecting Source Code . . . . .	30
B.C	Data Processing Source Code . . . . .	31
B.D	Data Storing Source Code . . . . .	34
B.E	Scripts & Services Source Codes . . . . .	38
B.E.1	Scripts . . . . .	38
B.E.2	Services . . . . .	39
B.E.2.1	Get IAM Credentials Service . . . . .	39
B.E.2.2	Spark Python Job Service . . . . .	39
B.E.2.3	Grafana Server Service . . . . .	40
B.F	Load Test Source Codes . . . . .	41
B.F.1	Artillery Load Test Configuration File . . . . .	41
B.F.2	Instances Monitoring Script . . . . .	42

# List of Figures

2.1	Project Architecture Diagram . . . . .	3
2.2	Data Collecting, Processing & Storing Pipeline Diagram . . . . .	4
2.3	EC2 Instance Configuration Screenshot . . . . .	7
2.4	Data Distributing Pipeline Diagram . . . . .	9
2.5	Auto Scaling Group Pipeline Diagram . . . . .	10
2.6	AMI Settings Screenshot . . . . .	11
2.7	Launch Template Settings Screenshot . . . . .	12
2.8	Auto Scaling Group Settings Screenshot . . . . .	13
2.9	Scaling Policy Settings Screenshot . . . . .	14
2.10	Target Group Settings Screenshot . . . . .	14
2.11	Load Balancer Settings Screenshot . . . . .	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 . . . . .	22

# **List of Tables**

2.1 UK air quality index, “Review of the UK Air Quality Index”, 2011 . . . . .	5
--	---

# **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 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.

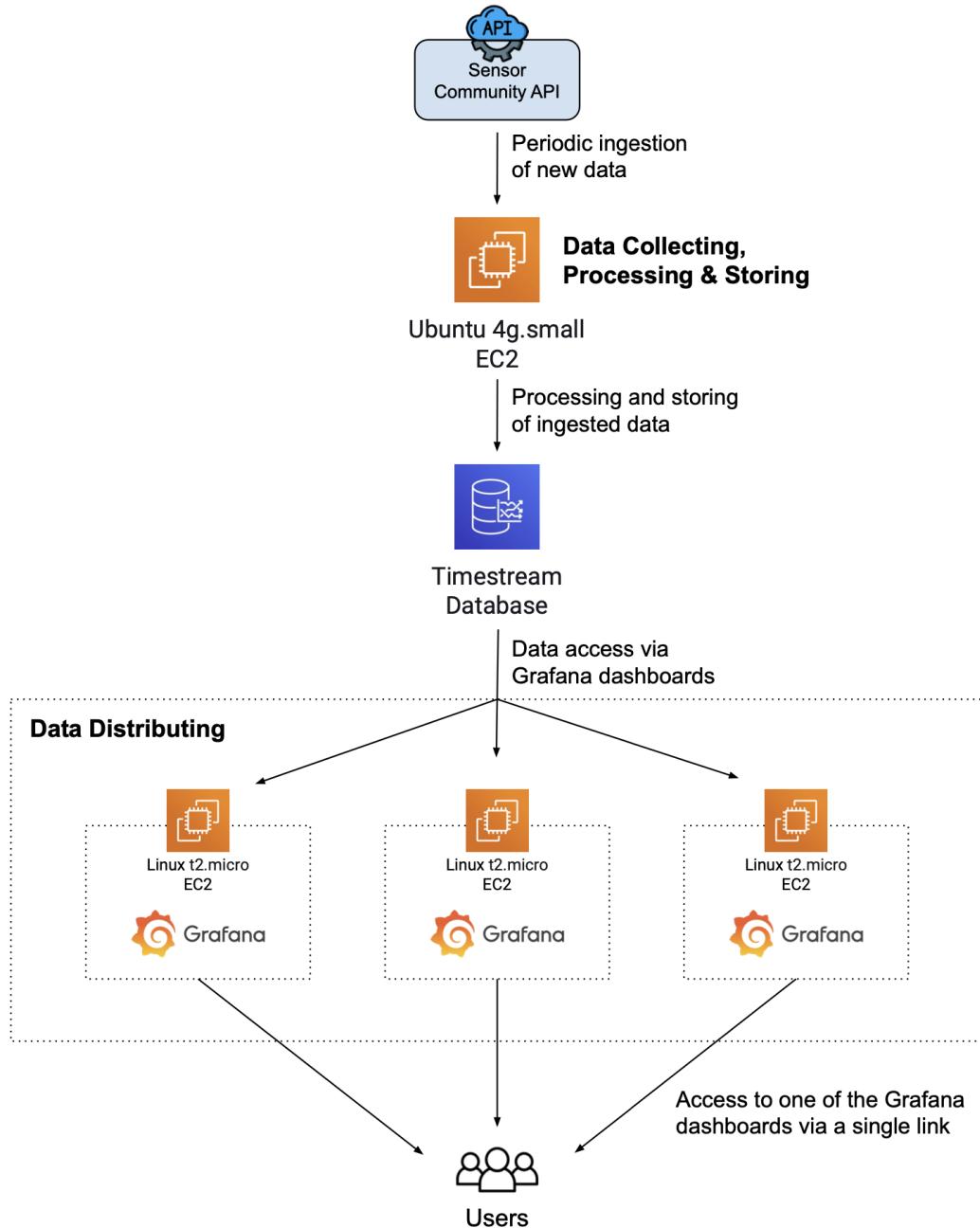


Figure 2.1: Project Architecture Diagram

## 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.

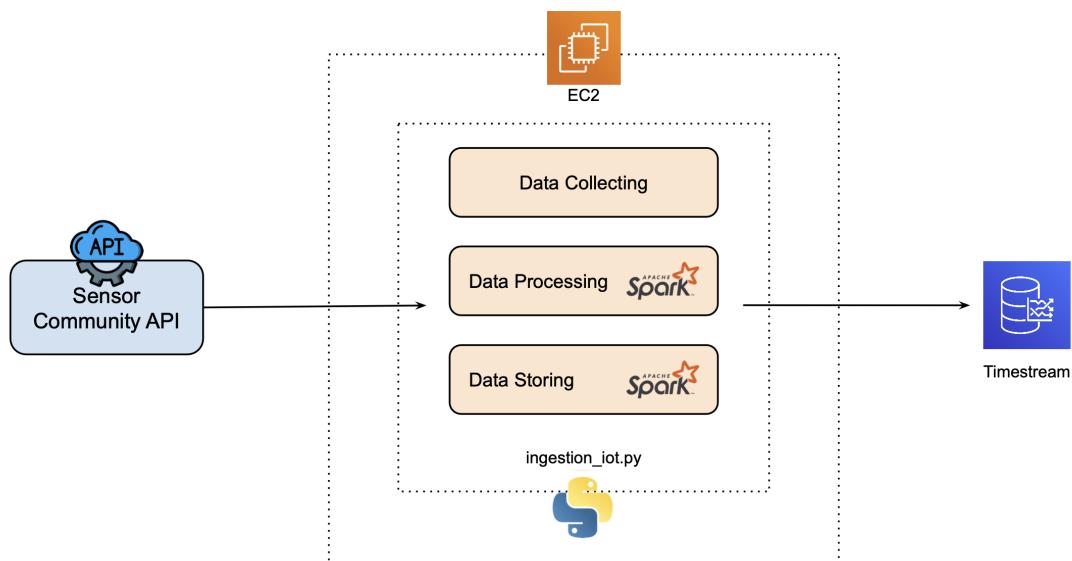


Figure 2.2: Data Collecting, Processing & Storing Pipeline Diagram

### 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

<b>Range</b>	<b>Air Quality Index</b>	<b>PM<sub>2.5</sub> Particles, 24 hour mean (<math>\mu\text{g}/\text{m}^3</math>)</b>	<b>PM<sub>10</sub> Particles, 24 hour mean (<math>\mu\text{g}/\text{m}^3</math>)</b>
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 [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.

## 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). 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), which starts the script responsible for the collection, processing and storage of IoT data (See Appendix B.A).

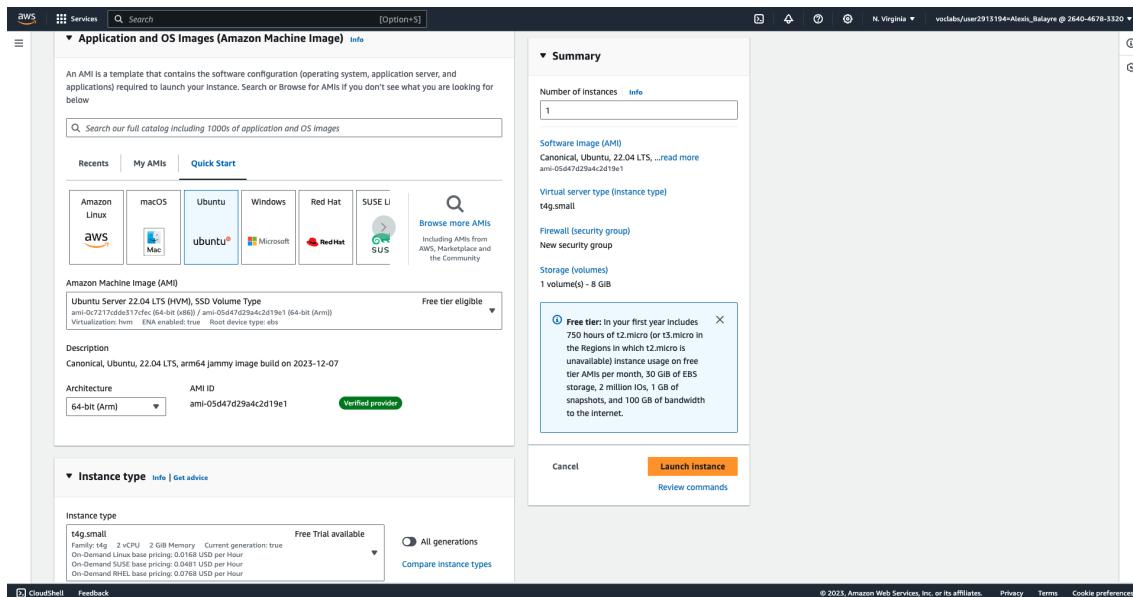


Figure 2.3: EC2 Instance Configuration Screenshot

## 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.

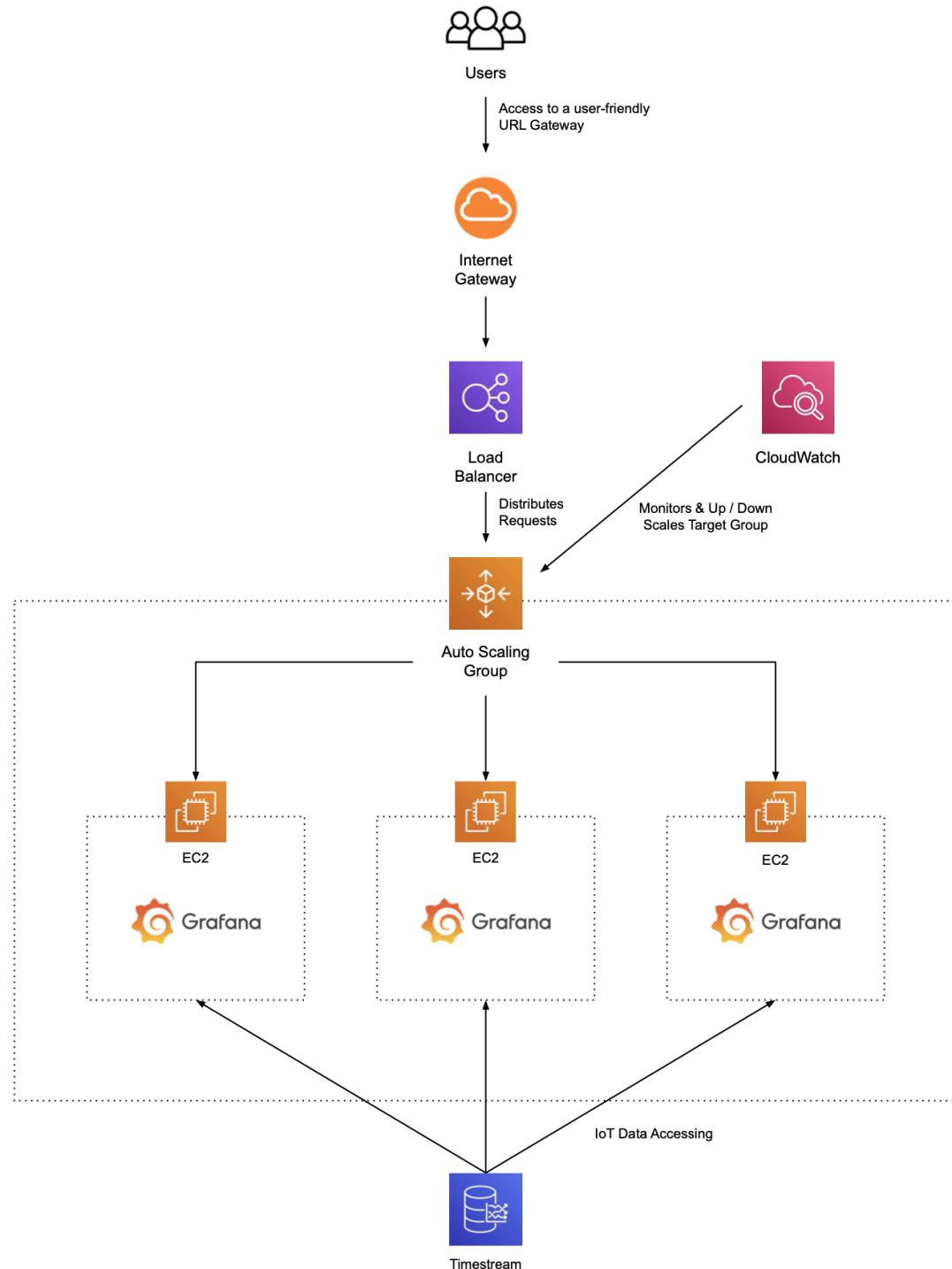


Figure 2.4: Data Distributing Pipeline Diagram

### 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.

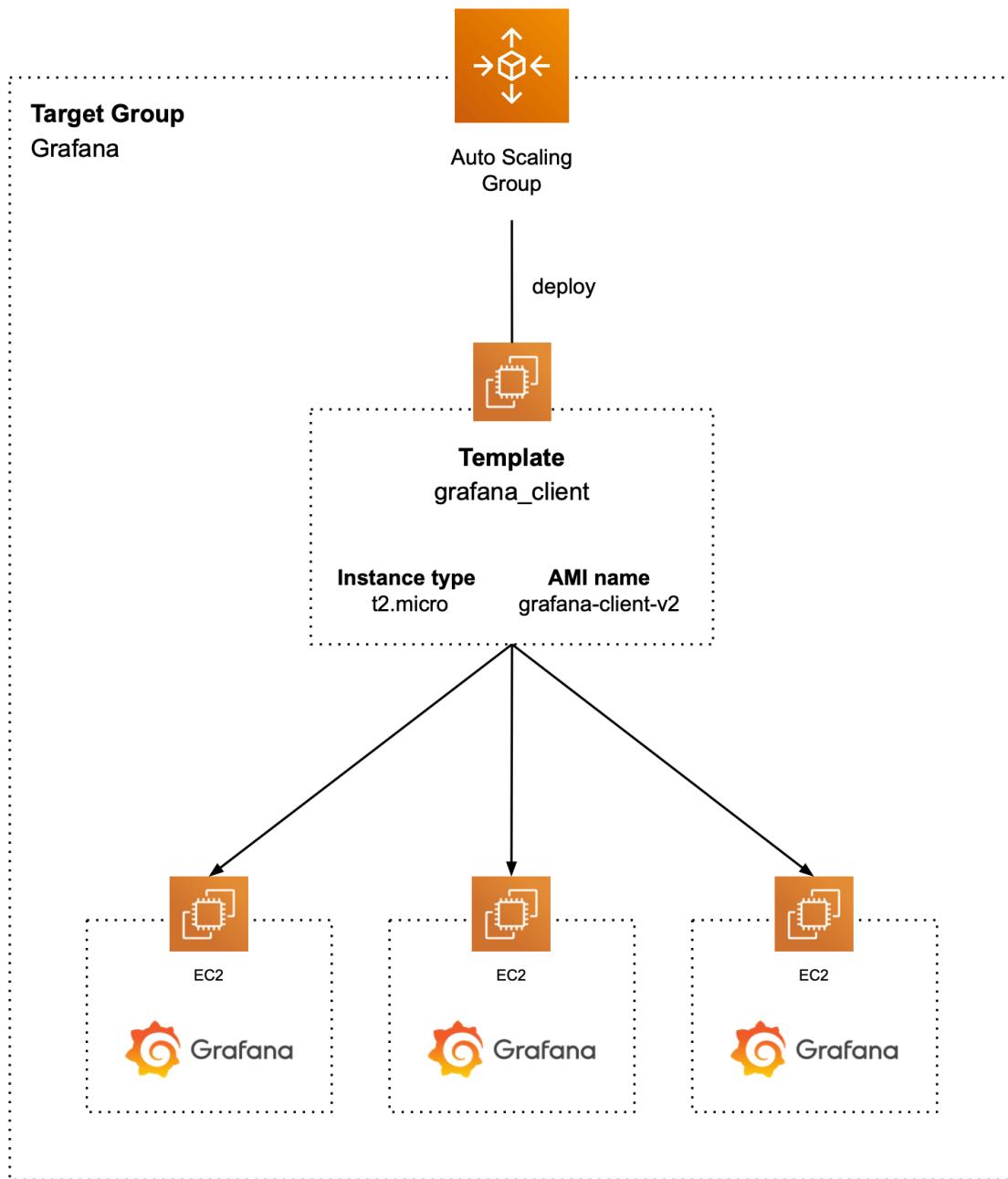


Figure 2.5: Auto Scaling Group Pipeline Diagram

### 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, specifically tailored to host the Grafana server for efficient data visualisation.

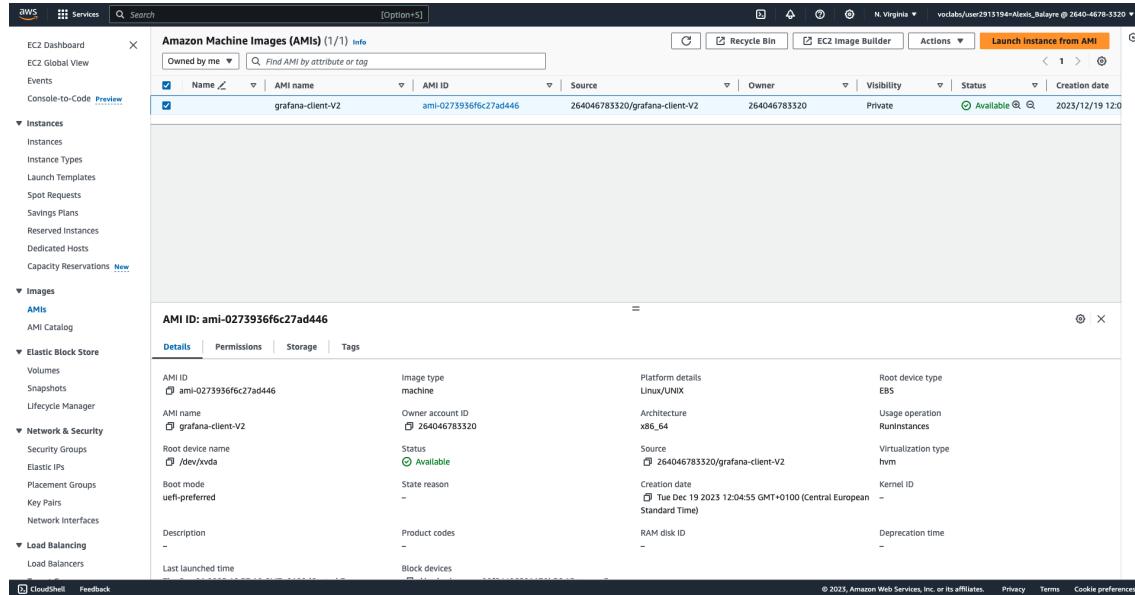


Figure 2.6: AMI Settings Screenshot

### 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.

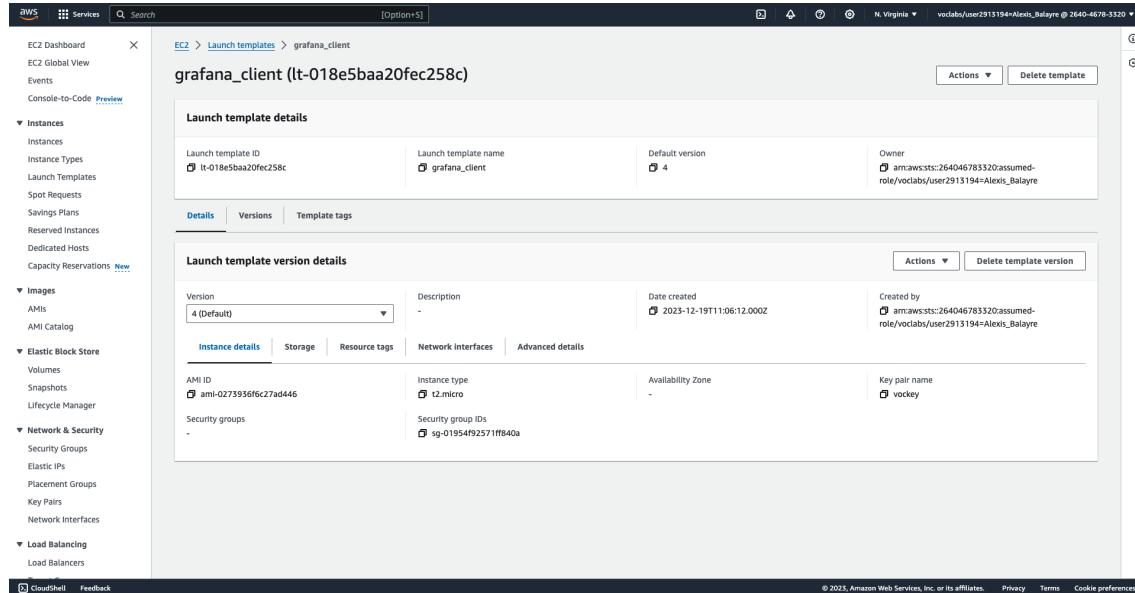


Figure 2.7: Launch Template Settings Screenshot

### 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. The group is also linked to the specified target group and load balancer, facilitating effective traffic distribution.

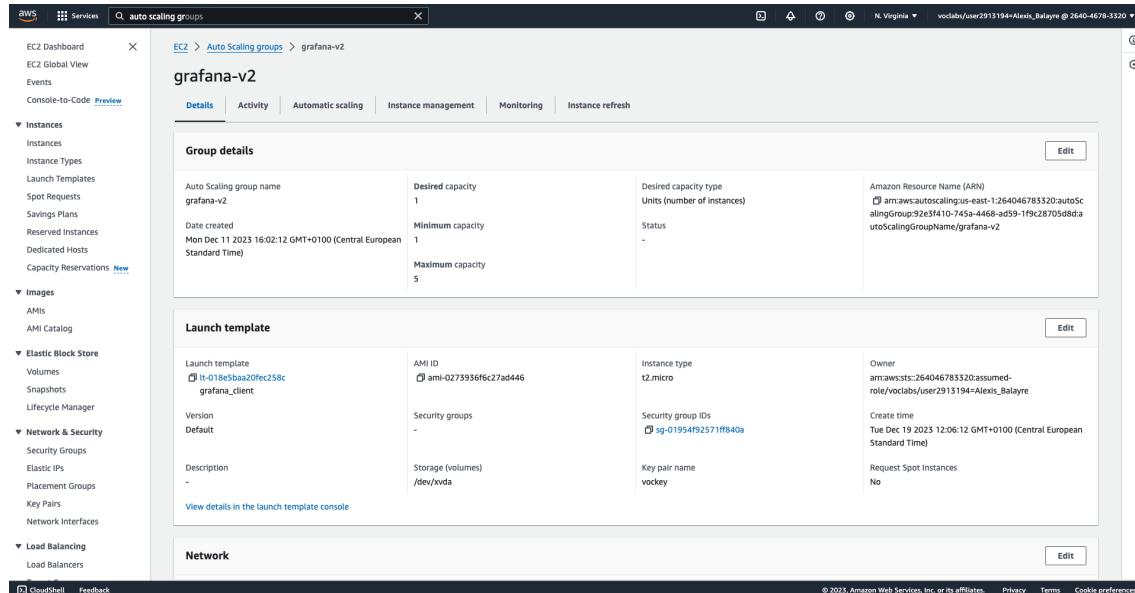


Figure 2.8: Auto Scaling Group Settings Screenshot

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.

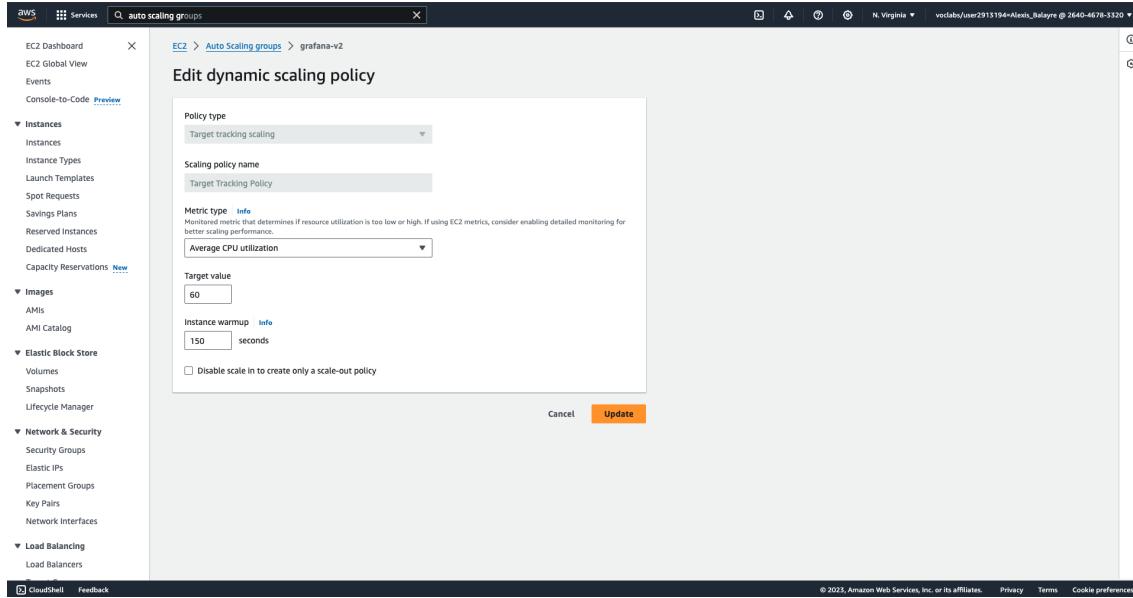


Figure 2.9: Scaling Policy Settings Screenshot

### 2.3.2.4 Target Group Configuration

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

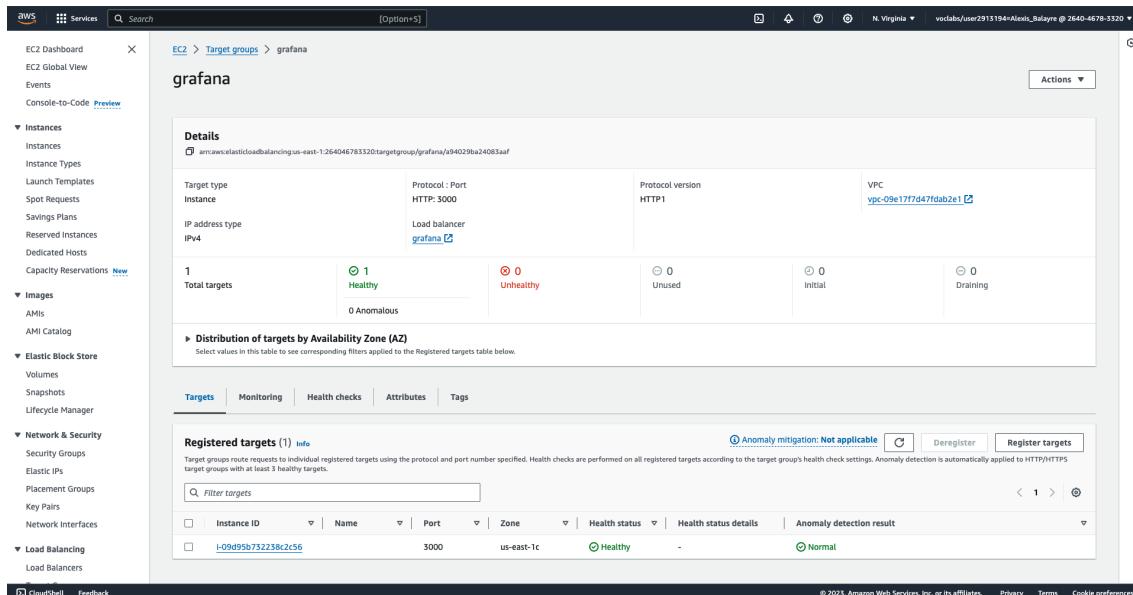


Figure 2.10: Target Group Settings Screenshot

### 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.

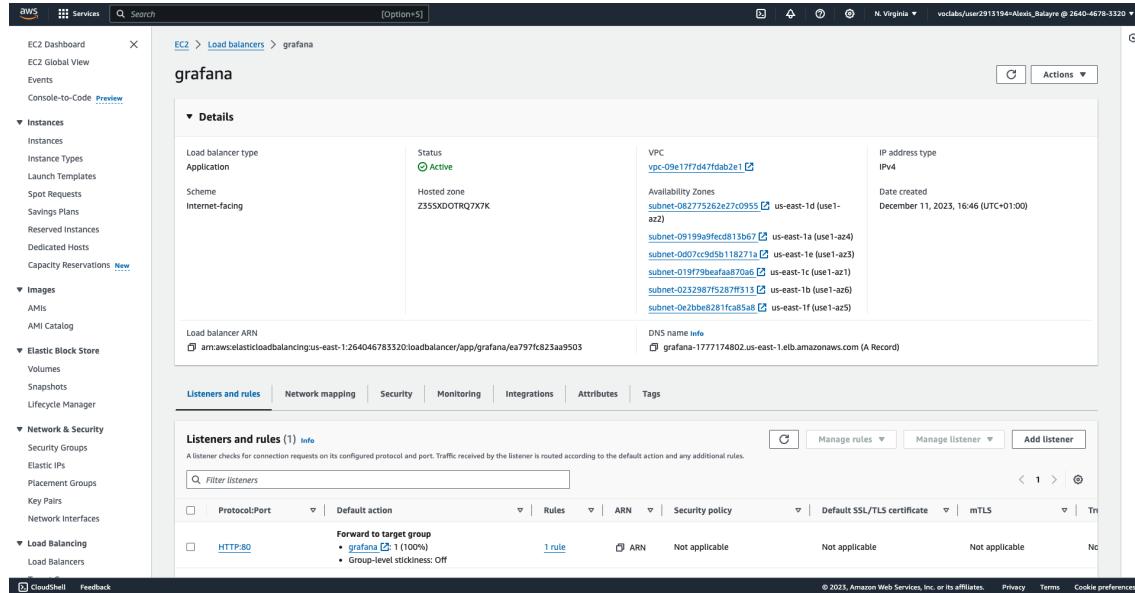


Figure 2.11: Load Balancer Settings Screenshot

# Chapter 3

## Results & Discussion

### 3.1 Results

#### 3.1.1 Accessing the Data - Grafana Dashboard

The Grafana dashboard, 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. 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.

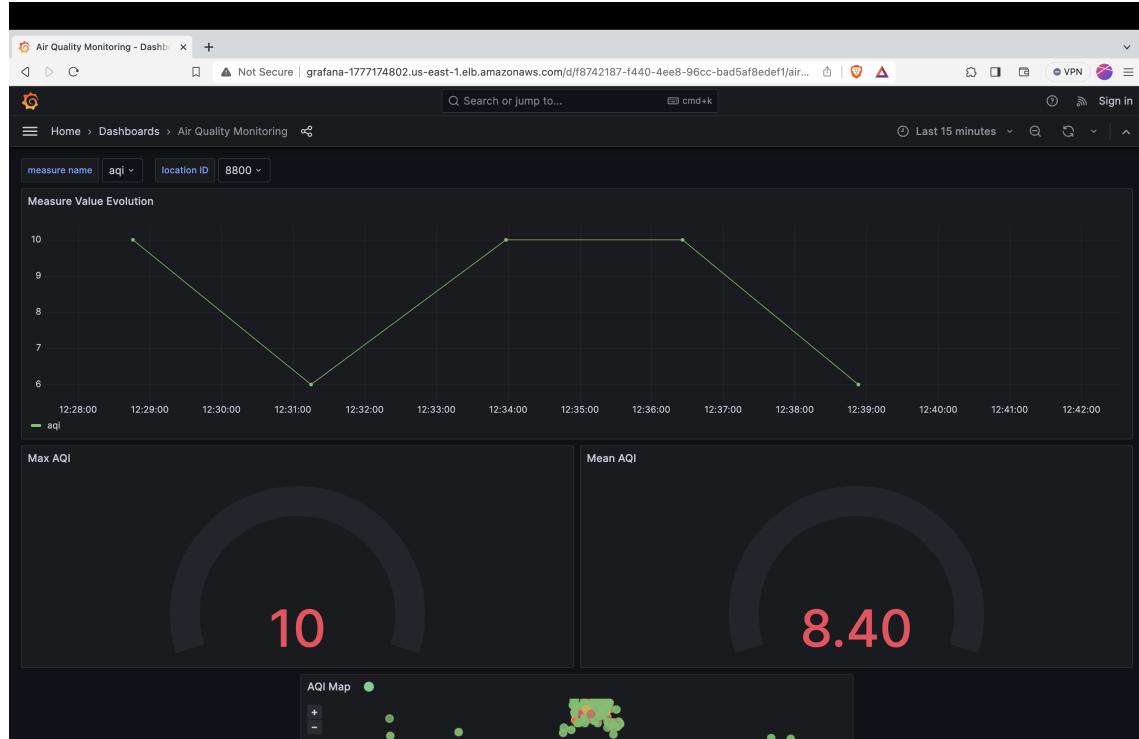


Figure 3.1: Grafana Dashboard: Measure Value Evolution, Max &amp; Mean AQI

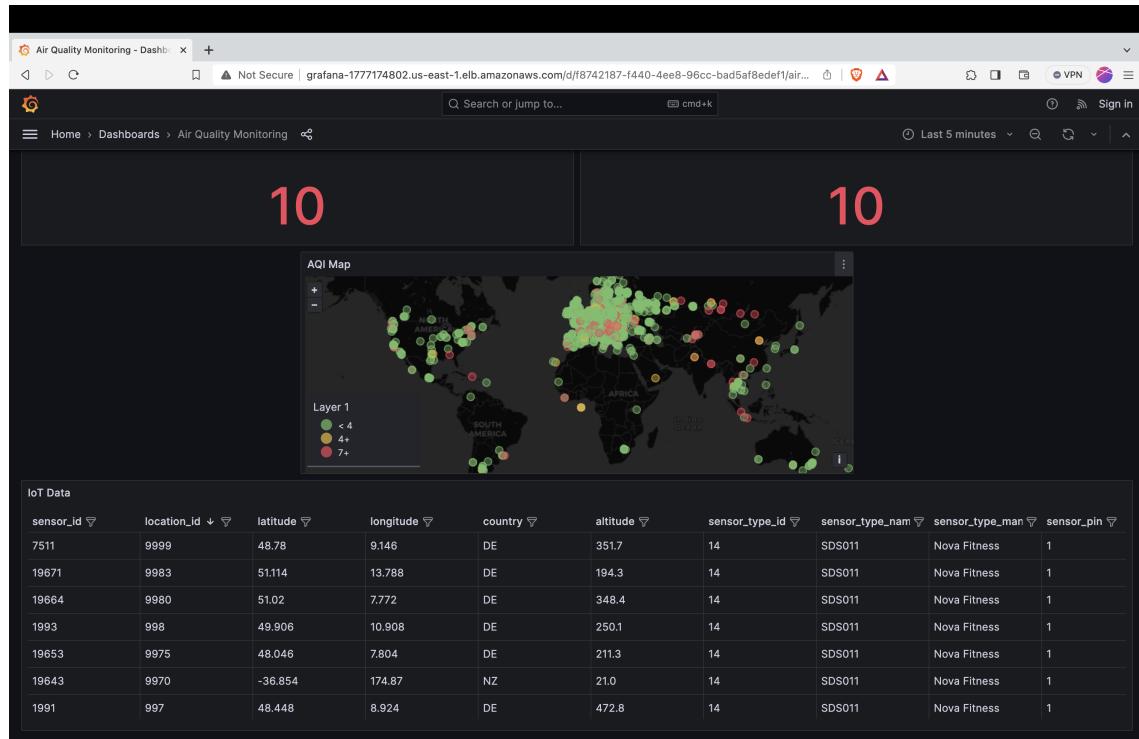


Figure 3.2: Grafana Dashboard: AQI Map &amp; IoT Data

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.

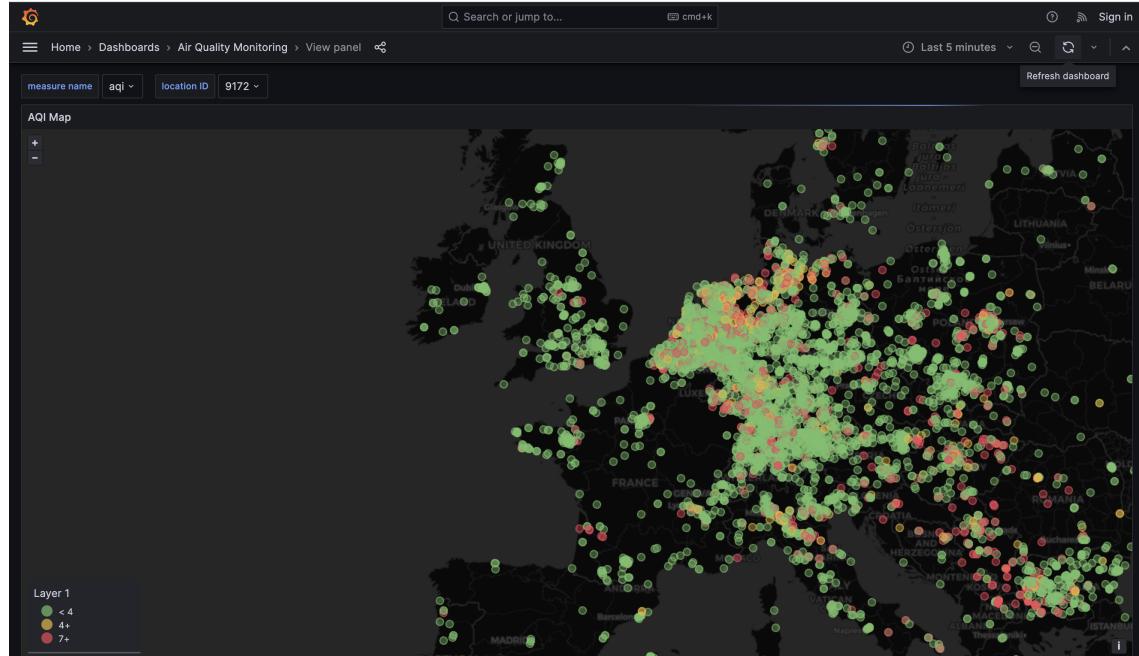


Figure 3.3: Grafana Dashboard AQI Map Panel

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

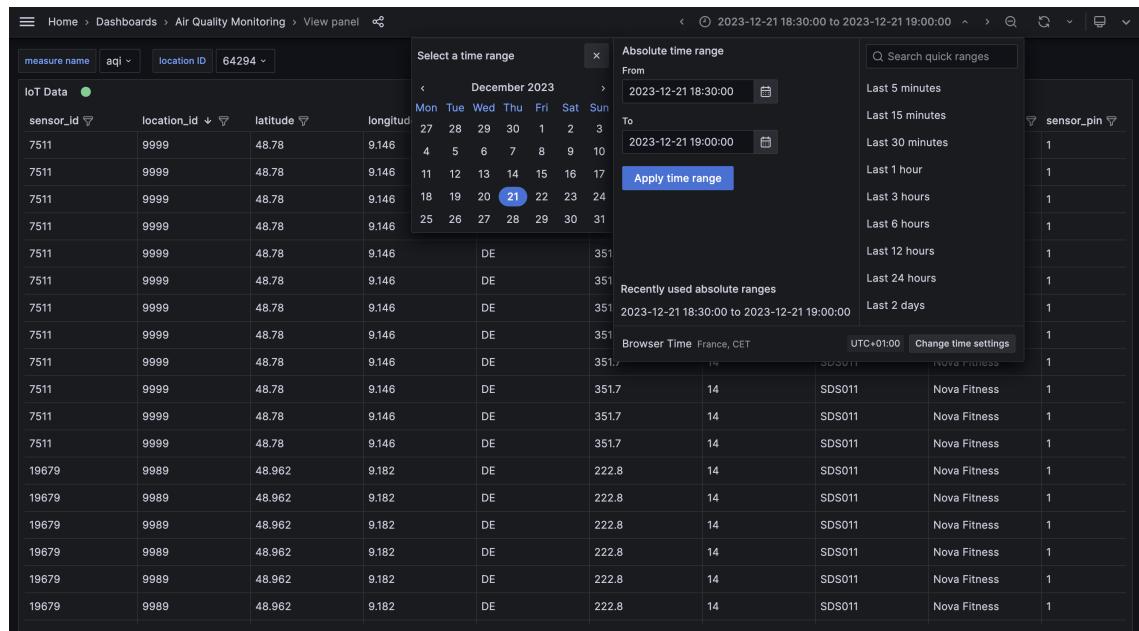


Figure 3.4: Grafana Dashboard IoT Data Panel - Time Range Choice

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

measure name	Enter variable value	location ID	64294						
IoT Data	P1								
sensor_id	aqi								
7511	durP1	48.78	9.146	DE	351.7	14	SDS011	Nova Fitness 1	
7511	durP2	48.78	9.146	DE	351.7	14	SDS011	Nova Fitness 1	
7511	ratioP1	48.78	9.146	DE	351.7	14	SDS011	Nova Fitness 1	
7511	ratioP2	48.78	9.146	DE	351.7	14	SDS011	Nova Fitness 1	
7511	pressure_at_sealevel	48.78	9.146	DE	351.7	14	SDS011	Nova Fitness 1	
7511	temperature	48.78	9.146	DE	351.7	14	SDS011	Nova Fitness 1	
7511	humidity	48.78	9.146	DE	351.7	14	SDS011	Nova Fitness 1	
7511	pressure	48.78	9.146	DE	351.7	14	SDS011	Nova Fitness 1	
7511	noise_LA_max	48.78	9.146	DE	351.7	14	SDS011	Nova Fitness 1	
7511	noise_LAeq	48.78	9.146	DE	351.7	14	SDS011	Nova Fitness 1	
7511	noise_LA_min	48.78	9.146	DE	351.7	14	SDS011	Nova Fitness 1	
7511	N25	48.78	9.146	DE	351.7	14	SDS011	Nova Fitness 1	
7511	P0	48.78	9.146	DE	351.7	14	SDS011	Nova Fitness 1	
7511	nmc	48.78	9.146	DE	351.7	14	SDS011	Nova Fitness 1	
7511	9999	48.78	9.146	DE	351.7	14	SDS011	Nova Fitness 1	
7511	9999	48.78	9.146	DE	351.7	14	SDS011	Nova Fitness 1	
7511	9999	48.78	9.146	DE	351.7	14	SDS011	Nova Fitness 1	
19679	9989	48.962	9.182	DE	222.8	14	SDS011	Nova Fitness 1	
19679	9989	48.962	9.182	DE	222.8	14	SDS011	Nova Fitness 1	
19679	9989	48.962	9.182	DE	222.8	14	SDS011	Nova Fitness 1	
19679	9989	48.962	9.182	DE	222.8	14	SDS011	Nova Fitness 1	
19679	9989	48.962	9.182	DE	222.8	14	SDS011	Nova Fitness 1	
19679	9989	48.962	9.182	DE	222.8	14	SDS011	Nova Fitness 1	
19679	9989	48.962	9.182	DE	222.8	14	SDS011	Nova Fitness 1	

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

Moreover, 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.

measure name	aqi	location ID	64294							
IoT Data										
sensor_id	location_id	48.132	-1.21	FR						
17790	9019	48.132	-1.21	FR						
17790	9019	48.132	-1.21	FR						
17790	9019	48.132	-1.21	FR						
17790	9019	48.132	-1.21	FR						
17790	9019	48.132	-1.21	FR						
17790	9019	48.132	-1.21	FR						
17790	9019	48.132	-1.21	FR						
17790	9019	48.132	-1.21	FR						
17790	9019	48.132	-1.21	FR						
17790	9019	48.132	-1.21	FR						
17790	9019	48.132	-1.21	FR						
17790	9019	48.132	-1.21	FR						
17790	9019	48.132	-1.21	FR						
17790	9019	48.132	-1.21	FR						
17310	8773	43.3	0.394	FR						
17310	8773	43.3	0.394	FR						
17310	8773	43.3	0.394	FR						
17310	8773	43.3	0.394	FR						
17310	8773	43.3	0.394	FR						
17310	8773	43.3	0.394	FR						
17310	8773	43.3	0.394	FR						

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.

The screenshot shows the Grafana IoT Data panel. At the top, there are filters for 'measure name' (aqi), 'location ID' (64294), and a dropdown for 'IoT Data'. Below this is a table with columns: sensor\_id, location\_id, latitude, longitude, and country. The data in the table consists of multiple rows of sensor information. To the right of the table is a sidebar titled 'Inspect: IoT Data' which shows '1 queries with total query time of 1.07 min'. It has tabs for 'Data' (selected), 'Stats', 'Meta data', and 'JSON'. Under the 'Data' tab, there is a 'Data options' section with 'Formatted data' and 'Download for Excel' checkboxes. A 'Download CSV' button is located at the top right of this section. Below these options is a preview of the CSV-formatted data, showing columns A\_sensor\_id, A\_location\_id, A\_latitude, A\_longitude, and A\_country, followed by several rows of data.

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 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.

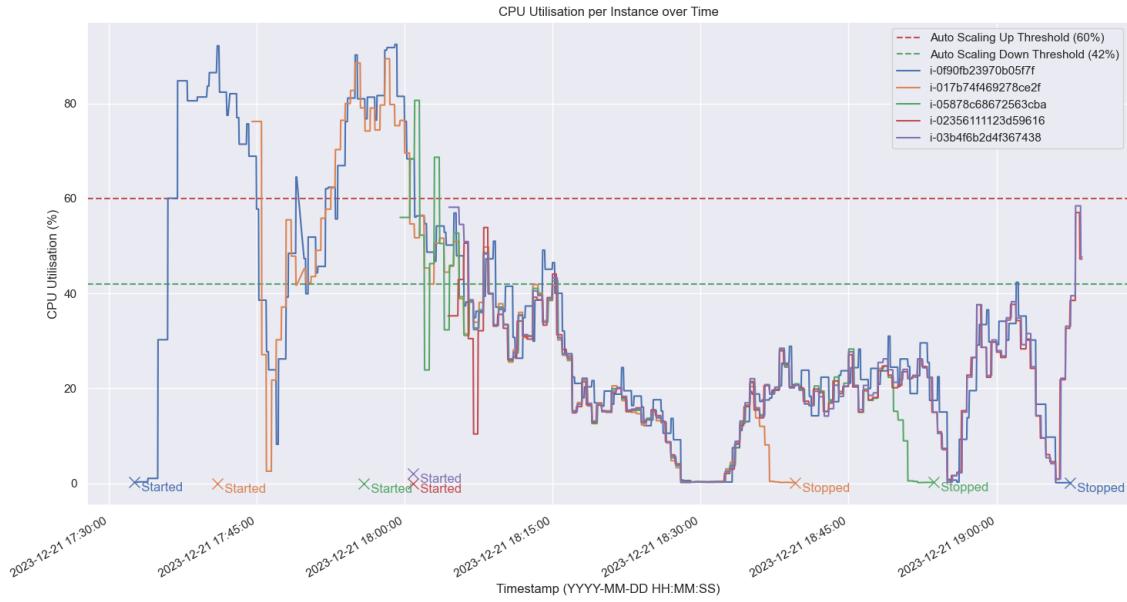


Figure 3.8: Load Test Result

## 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  
    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: 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.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)
```