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

In an increasingly connected world, cloud computing and the Internet of Things (IoT) are revolutionising many fields, including environmental monitoring. This technological development offers unprecedented possibilities for managing and analysing air quality, a major public health issue. This report, drawn up as part of my Master's degree in Cloud and Embedded Systems Science and Technology (CSTE), focuses on the use of these technologies to collect, process and distribute environmental data.

The main aim of this assignment is to store and make accessible the latest air quality data, captured by a network of small environmental IoT sensors. The project aims to provide a reliable platform for real-time consultation of environmental data, a crucial tool for researchers, decision-makers and the general public.

We face a number of technical challenges in achieving this objective. Firstly, managing the large quantities of data generated by IoT sensors requires a robust and adaptable cloud infrastructure. Secondly, calculating the Air Quality Index (AQI) from this data in real time requires considerable processing power and accuracy. Finally, the need to keep the system adaptable and responsive to varying workloads presents an additional challenge.

To address these challenges, our approach is to use a database located in the cloud, specifically designed to manage and process large volumes of IoT data. This database will be regularly updated with new data, while allowing quick and easy access for end users. In addition, we will be implementing advanced algorithms for calculating the AQI, guaranteeing the accuracy and reliability of the information provided.

The importance of this system is not limited to environmental monitoring; it also has a significant impact on public health, urban planning and environmental awareness. By providing accurate and up-to-date data, we contribute to a better understanding and management of air quality.

In conclusion, this report will detail our methodology, the architecture of the system, the challenges encountered and the solutions adopted. We will also discuss the implications of our work, not only in technical terms but also in terms of its practical applications and impact on different stakeholders.

Chapter 2

Methodologies

2.1 Data Collecting, Processing & Storing

2.1.1 Overview of the Pipeline Architecture

The initial pipeline in this project consists of three primary components: **Data Collecting**, **Data Processing**, and **Data Storing**.

During the **Data Collecting** phase, the most recent version of the dataset is acquired from its source. This is followed by the **Data Processing** phase, where the data is formatted, and the Air Quality Index (AQI) is calculated for each particulate matter sensor. Lastly, in the **Data Storing** phase, the data from each sensor is methodically stored in a time-series database.

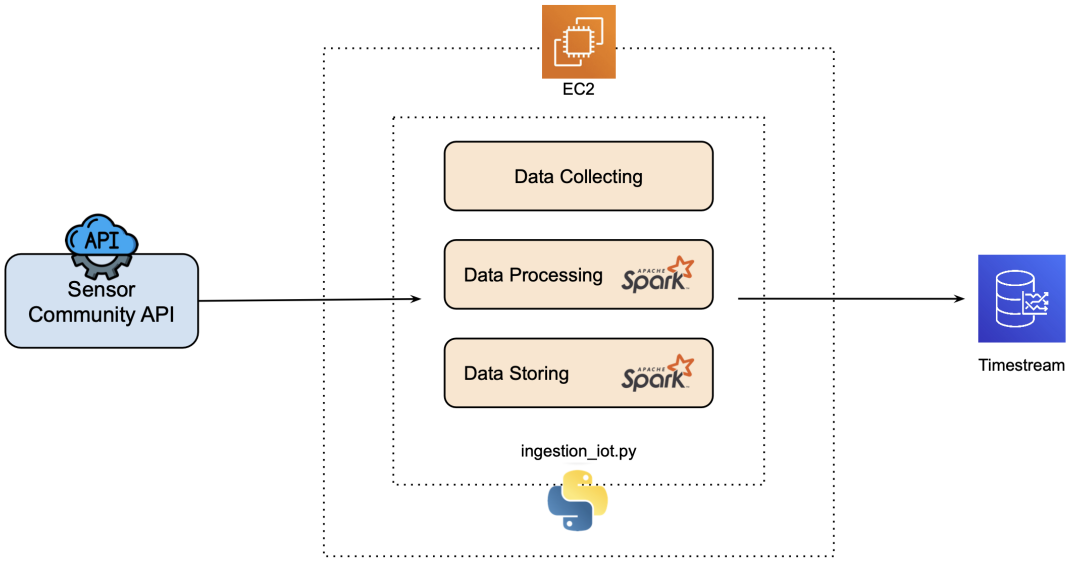


Figure 2.1: Data Collecting, Processing & Storing Pipeline Diagram

2.1.2 Data Collecting

2.1.2.1 Data Source

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

1. **5-Minute Averaged Data API:** This API provides data averaged over the last 5 minutes for each sensor. This is useful for near real-time analysis or immediate air quality assessments, particularly in test or active monitoring contexts.

2. **24 Hour Averaged Data API:** This API provides data averaged over the last 24 hours for each sensor. It is particularly suited to analysing daily trends and understanding environmental changes over a longer period.

2.1.2.2 Ingestion Script

The data acquisition process in this project involved leveraging both APIs from the Sensor Community. To facilitate this, the Python `requests` library was employed, enabling efficient data retrieval. Once collected, the data was temporarily stored in a local cache, formatted as a Spark DataFrame for optimized 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). Notably, this function is programmed to run at a consistent interval of every 10 seconds, ensuring a steady and up-to-date flow of data from the sensors. This systematic approach not only streamlines the data collection process but also enhances the reliability and timeliness of the gathered information.

2.1.3 Data Processing

2.1.3.1 Processing Script

Once the data has been ingested, the `computeAQI` function is called to compute the Air Quality Index (AQI) values using user-defined functions (UDFs) that transform measured PM2.5 and PM10 concentrations into the corresponding AQI values, based on the thresholds as defined in Table 2.1. This operation, located within the `processing.py` script (see Appendix B.C for detailed code), utilises UDFs applied to a Spark DataFrame, a distributed collection of data organised into named columns, which facilitates data processing operations. The `explode` operation on the DataFrame transforms the sensor data values into individual rows, enabling the UDFs to be applied to each concentration measurement. The process is optimised by caching the DataFrame, thereby reducing processing time by minimising read and write operations to disk.

This procedure uses Apache Spark, a distributed data processing system, to calculate the AQI from data collected in real time by sensors. Spark's distributed approach is especially adept at managing the large volumes of data generated by sensors, enabling the AQI to be calculated quickly and efficiently.

Finally, the AQI is calculated by grouping the data by sensor ID and timestamp, followed by aggregation to determine the maximum AQI value between PM2.5 and PM10 particles. This ensures that the AQI reflects the highest concentration of pollutants, which is essential for an accurate assessment of exposure to air pollutants. This distributed calculation, performed in parallel across multiple processing nodes, illustrates the power of Spark in processing environmental IoT data, providing up-to-date and accurate air quality information that is essential for public health decision-making.

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

2.1.4 Data Storing

2.1.4.1 Database Choice

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

Faced with these challenges, time series databases (TSDBs) are emerging as an optimal solution. Unlike traditional relational databases, TSDBs are specifically designed to manage time-series data, such as that from IoT sensors. They offer better performance in tracking, monitoring and aggregating data over time. In addition, TSDBs such as Amazon Timestream stand out for their ability to ingest data quickly and process large volumes efficiently, while ensuring regular updates and in-depth analysis, essential aspects for applications such as Air Quality Index analysis.

Amazon Timestream is a good choice for this project. As a cloud-native time-series database, it offers superior time-series management capabilities tailored to IoT sensor data. What sets Amazon Timestream apart is the speed with which it ingests data and its efficiency in processing large volumes of data, enabling regular updates and in-depth analysis. Its ability to adapt to changing workloads ensures consistent performance, a

major advantage for real-time data management. What's more, its advanced security features meet strict standards of confidentiality and data sovereignty, an essential

2.1.4.2 Storing Script

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

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

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

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

2.1.5 Pipeline Implementation on AWS

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

1. **EC2 Instance Configuration:** An EC2 t4g.small instance has been configured to host the data pipeline. A t4g.small instance was chosen for its balance between performance and energy efficiency. These instances, based on ARM architecture, are ideal for light to moderate workloads, which correspond to the requirements of continuous data processing. In addition, Apache Spark has been installed on this instance to process the data collected. Spark was chosen for its ability to process data in memory and in parallel, optimising performance for data processing workloads.

2. **Setting Up Automated Services:** Two systemd services were then set up to automate the essential tasks: The `get_iam_credentials.service` (see Appendix B.E.2.1) executes `get_iam_credentials.sh` (see Appendix B.E.1). This script retrieves the instance's IAM credentials, enabling secure integration with other AWS services. The `spark_python_job.service` (see Appendix B.E.2.2) launches `start_spark_job.sh` (see Appendix B.E.1), which starts the main data processing script (see Appendix B.A). This script orchestrates the collection, processing (AQI calculation) and storage of IoT sensor data. These services are activated each time the instance is started.

2.2 Data Distributing

2.2.1 Overview of the Pipeline Architecture

The second pipeline in this project is the one responsible for distributing the data collected. Here is a brief overview of the pipeline architecture:

1. **Internet Gateway**:
   Users initiate their interaction with the system through the secure Internet Gateway. Access to the Grafana dashboards is facilitated through a dedicated URL, which can be visited at: Air Quality Monitoring Dashboard.

2. **Load Balancer**:
   The Load Balancer evenly distributes incoming traffic to the EC2 instances, ensuring balanced workload distribution and optimal resource utilization.

3. **Grafana Dashboard Access**:
   A Grafana dashboard, hosted on one of the Load Balancers, presents the ingested data stored in the Timestream database, providing real-time visualization of the air quality metrics.

4. **CloudWatch Monitoring**:
   AWS CloudWatch continuously monitors the total CPU utilization across all EC2 instances within the Auto Scaling Group. Depending on the aggregated CPU usage, CloudWatch dynamically upscales or downscales the resources to maintain system efficiency and cost-effectiveness.

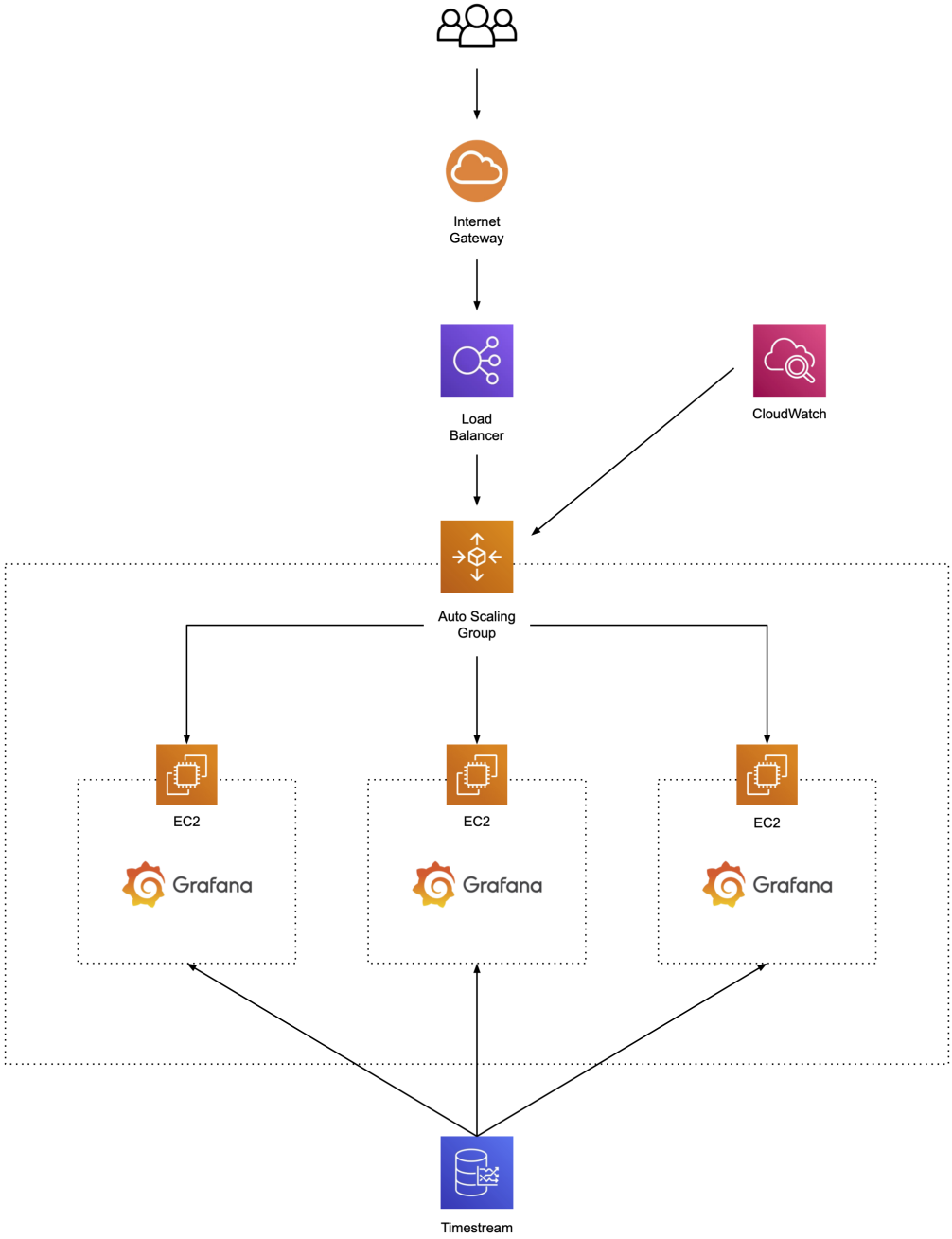


Figure 2.2: Data Distributing Pipeline Diagram

2.2.2 Auto Scaling Group Configuration

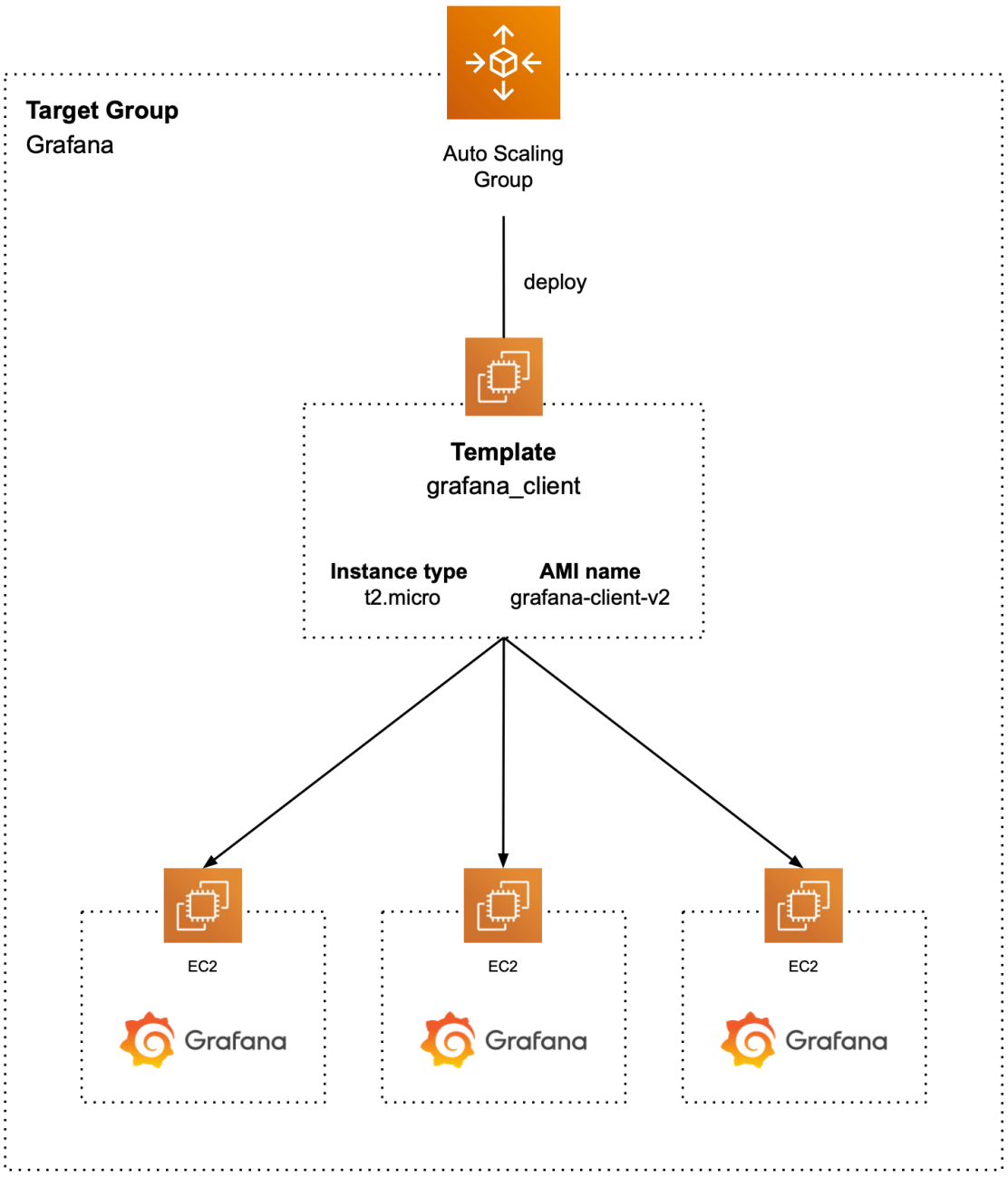


Figure 2.3: Auto Scaling Group Pipeline Diagram

2.2.2.1 AMI Configuration

An Amazon Machine Image (AMI) is a template that contains the software configuration (operating system, application server, and applications) required to launch an EC2 instance. The AMI used in this project is based on the Amazon Linux 2 AMI, which is a stable, secure and high-performance Linux distribution. The AMI is configured with the following software:

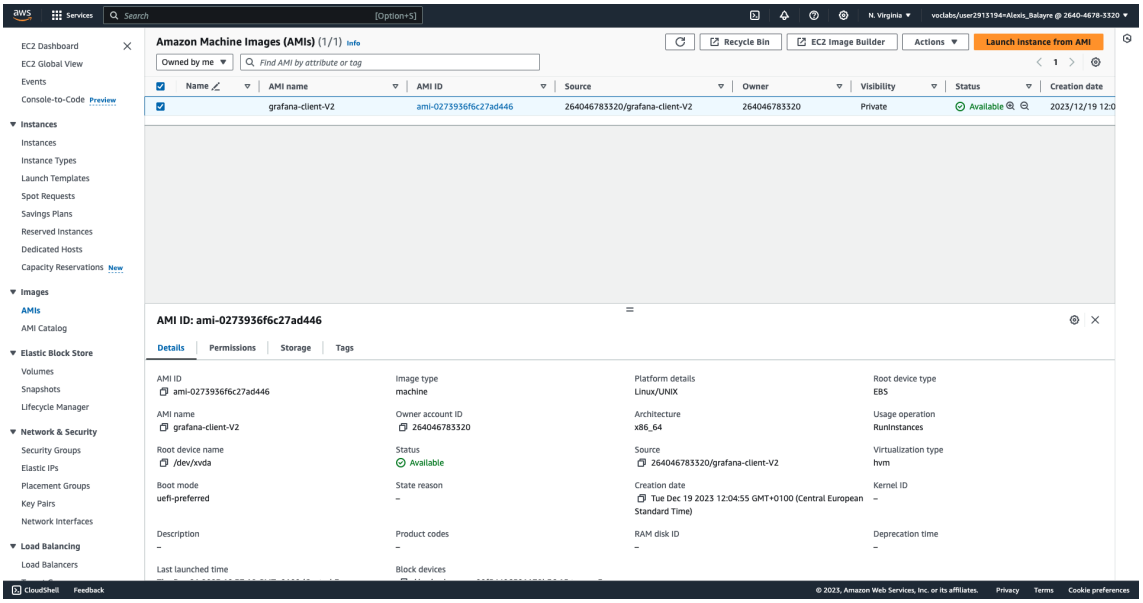


Figure 2.4: AMI Settings Screenshot

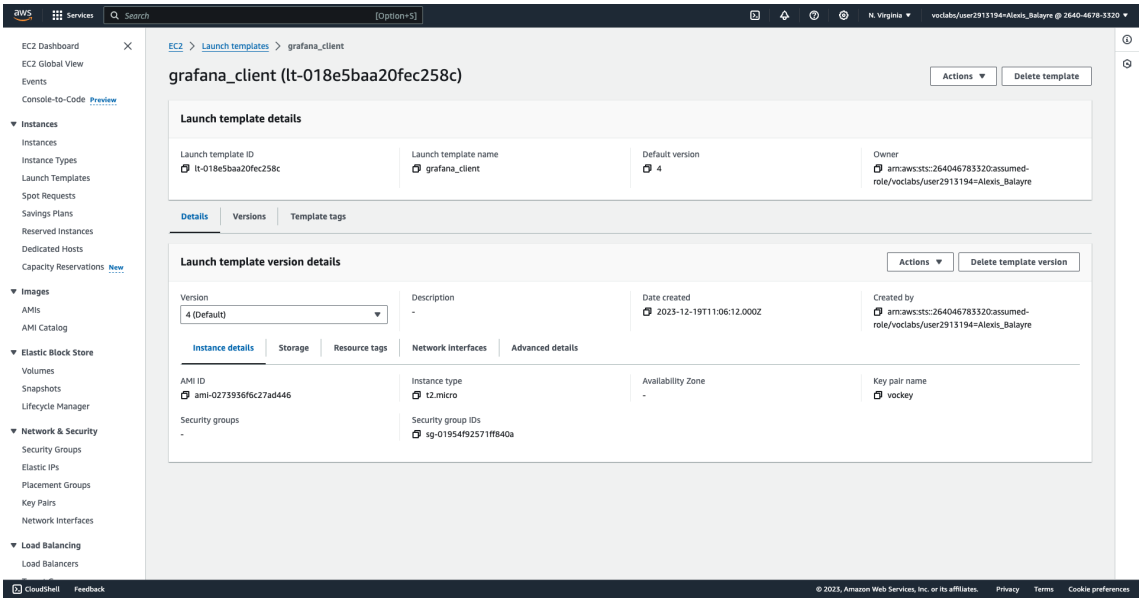


Figure 2.5: Launch Template Settings Screenshot

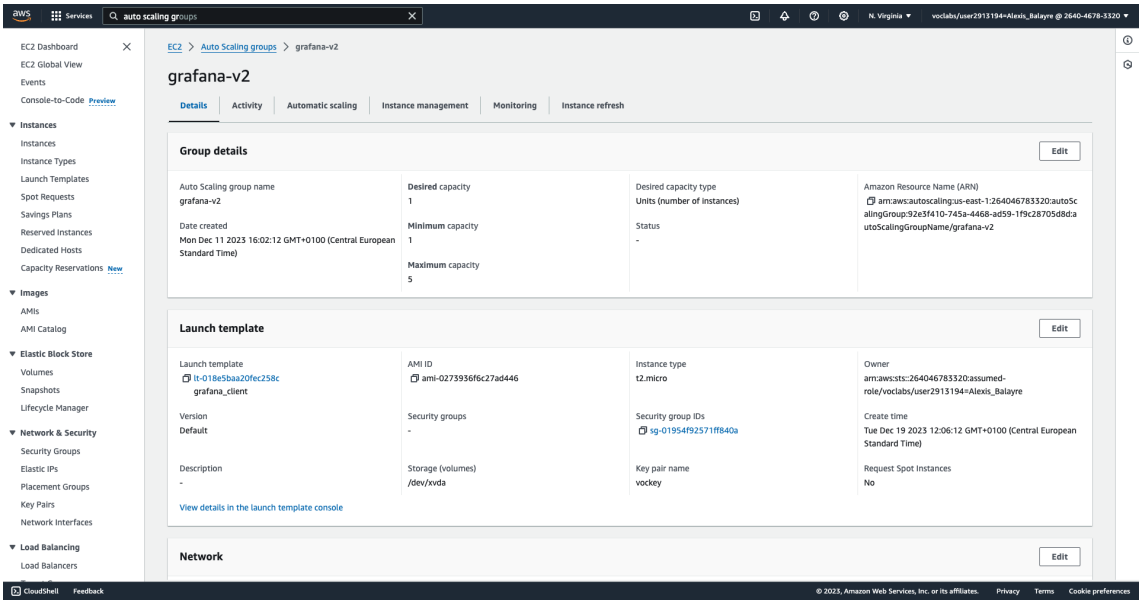


Figure 2.6: Auto Scaling Group Settings Screenshot

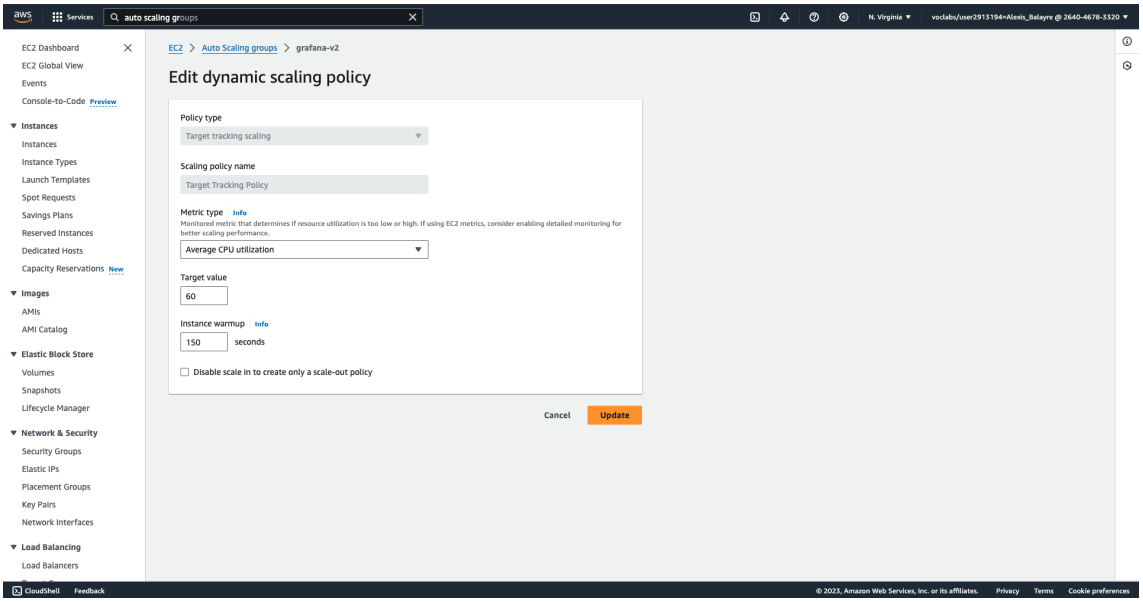


Figure 2.7: Scaling Policy Settings Screenshot

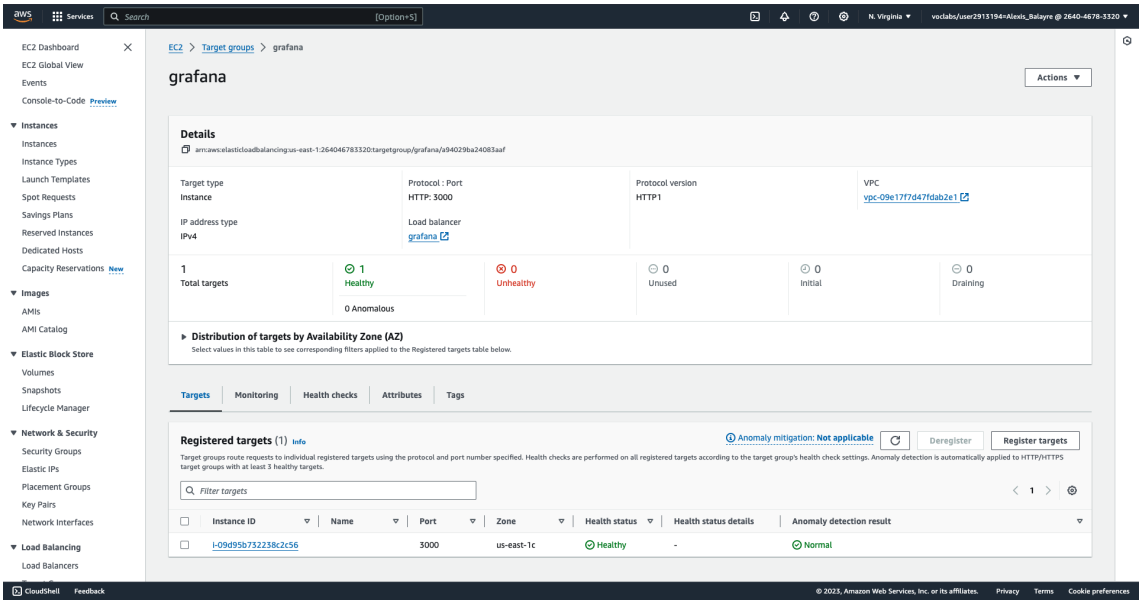


Figure 2.8: Target Group Settings Screenshot

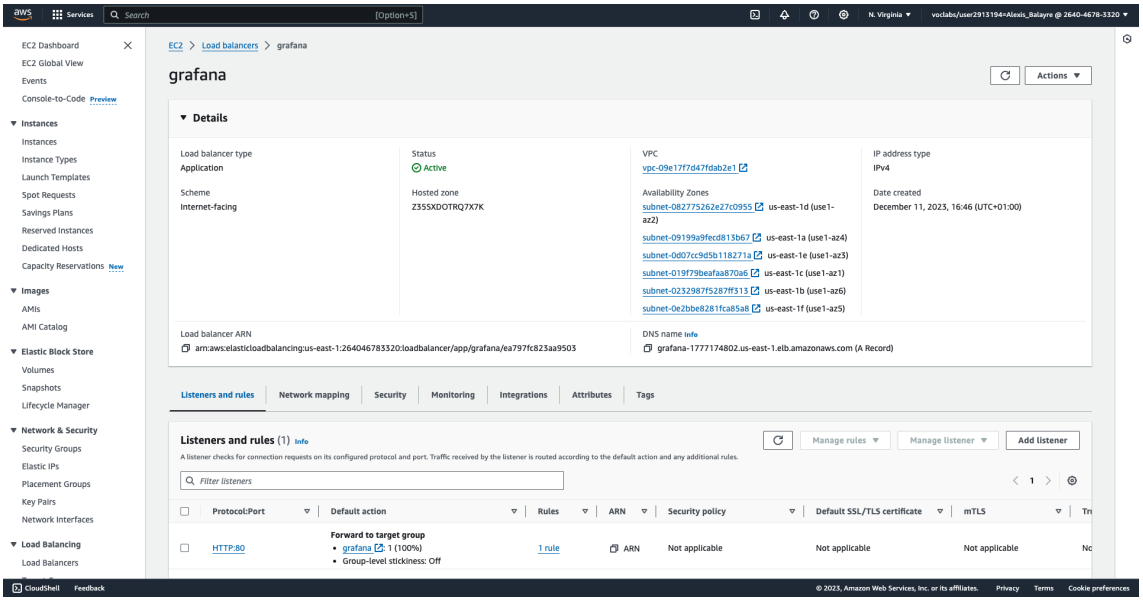


Figure 2.9: Load Balancer Settings Screenshot

2.3 Pipeline of the Project

2.3.1 Pipeline Overview

The pipeline of this project is composed of four main components: **data ingestion**, **query 1**, **query 2** and **query 3**.

The **ingestion** task retrieves the last version of the data set from the source and stores it in the data lake (CSV file stored in local storage). Then, the **queries 1, 2 & 3** tasks retrieves the data from the data lake and performs the queries on the data.

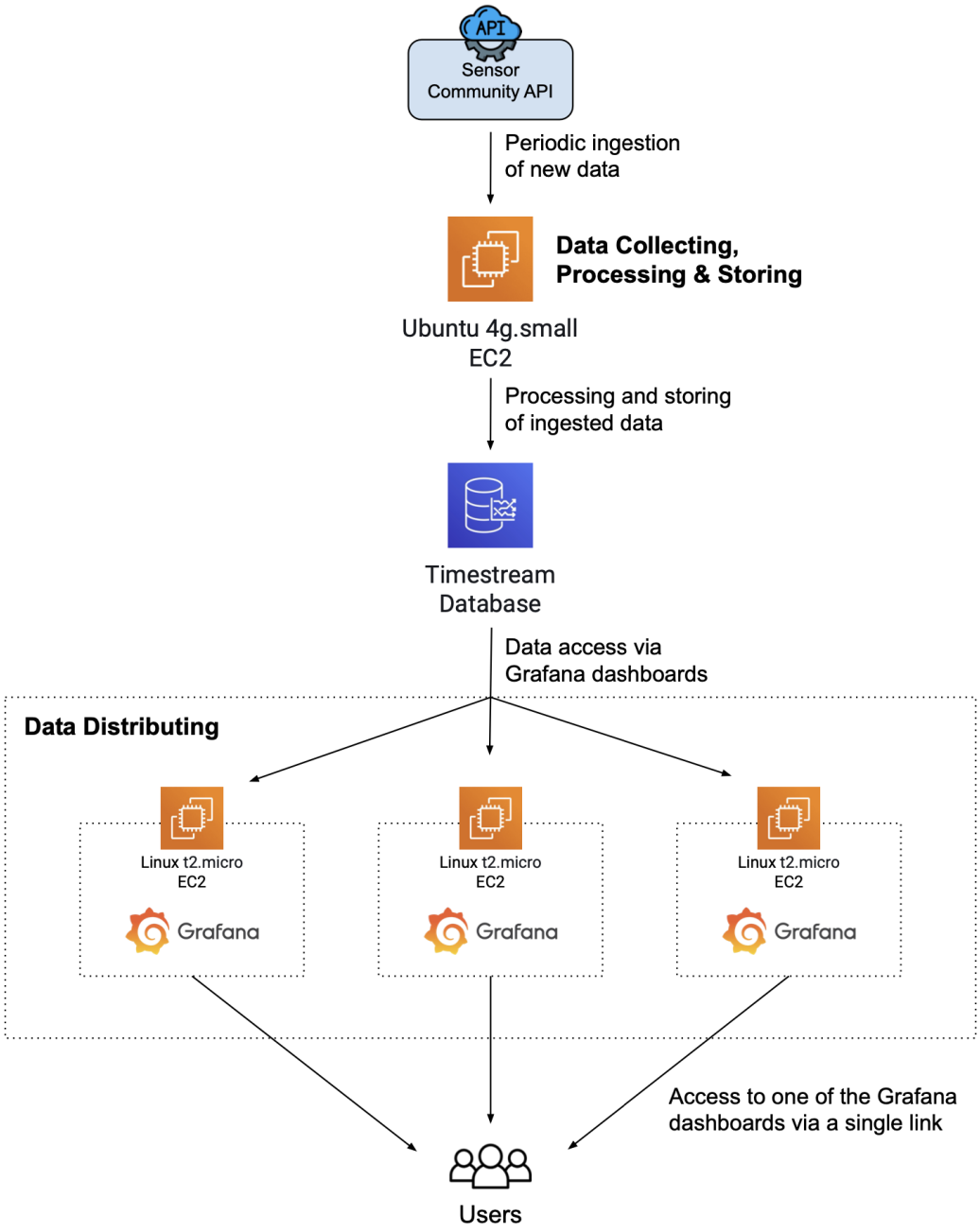


Figure 2.10: Data Distributing Pipeline Diagram

2.3.2 Pipeline Orchestration

In order to orchestrate and automate the pipeline, a scheduled task must be run every day to retrieve the latest version of the dataset and run the tasks when a new daily row is added at 23:59 UTC to the dataset.

To perform this task, a DAG (Directed Acyclic Graph) was created using Apache Airflow. The DAG is scheduled to run every day at 00:00 UTC and is composed of four tasks: **ingestion**, **query 1**, **query 2** and **query 3**. The screenshot below **??** shows the DAG graph of the pipeline in the web interface of Apache Airflow.

The benefits of using a workflow platform such as Apache Airflow are its ability to schedule and automate the pipeline, as well as its ability to monitor the pipeline and send alerts if a task fails.

Chapter 3

Results & Discussion

3.1 Results

The programme was last run on 18 November 2023. The appendix **??** shows the output of the program on the terminal.

3.1.1 Accessing the Data - Grafana Dashboard

The first query takes around 3 seconds, and the table **??** shows a sample of the data calculated during the task. In order to evaluate performance, an equivalent script not using Spark was run. Execution time was 0.5 sec. The 3.2 section will cover this point. The results are consistent with what was expected (**?**). For example, the figure **??** shows that Brazil was heavily impacted by the pandemic, reaching an average peak in February 2022. The same is true for Korea in figure **??** and the United States in figure **??**, which were heavily impacted.

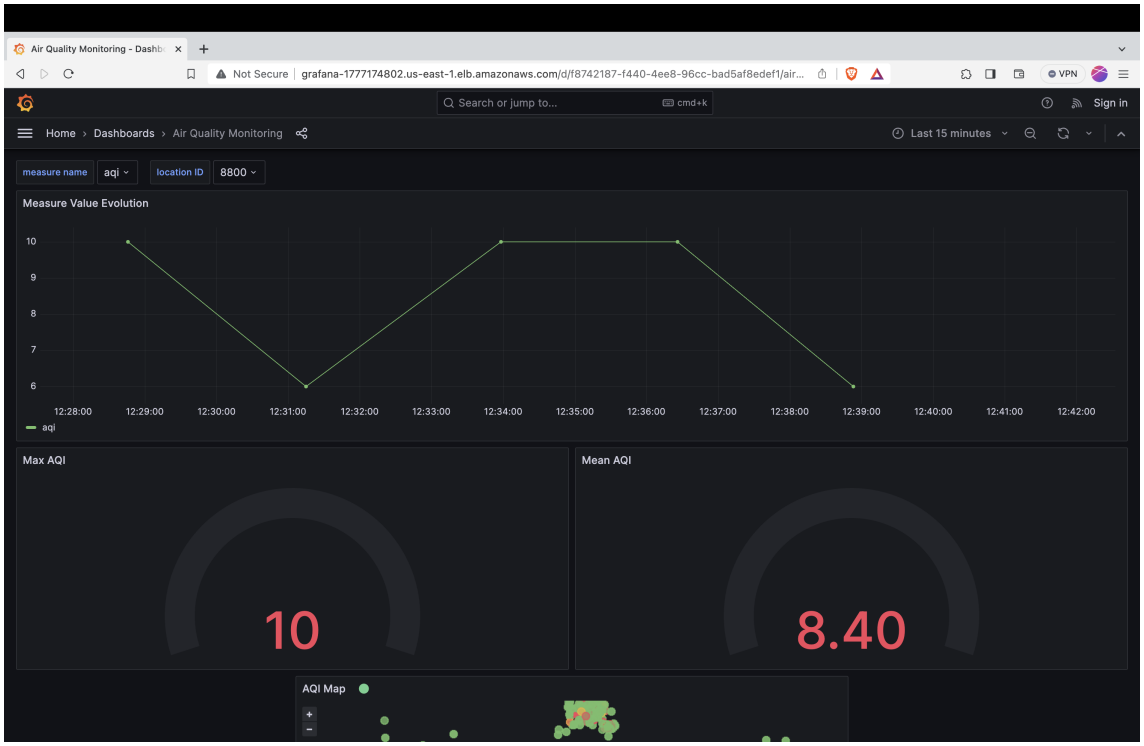
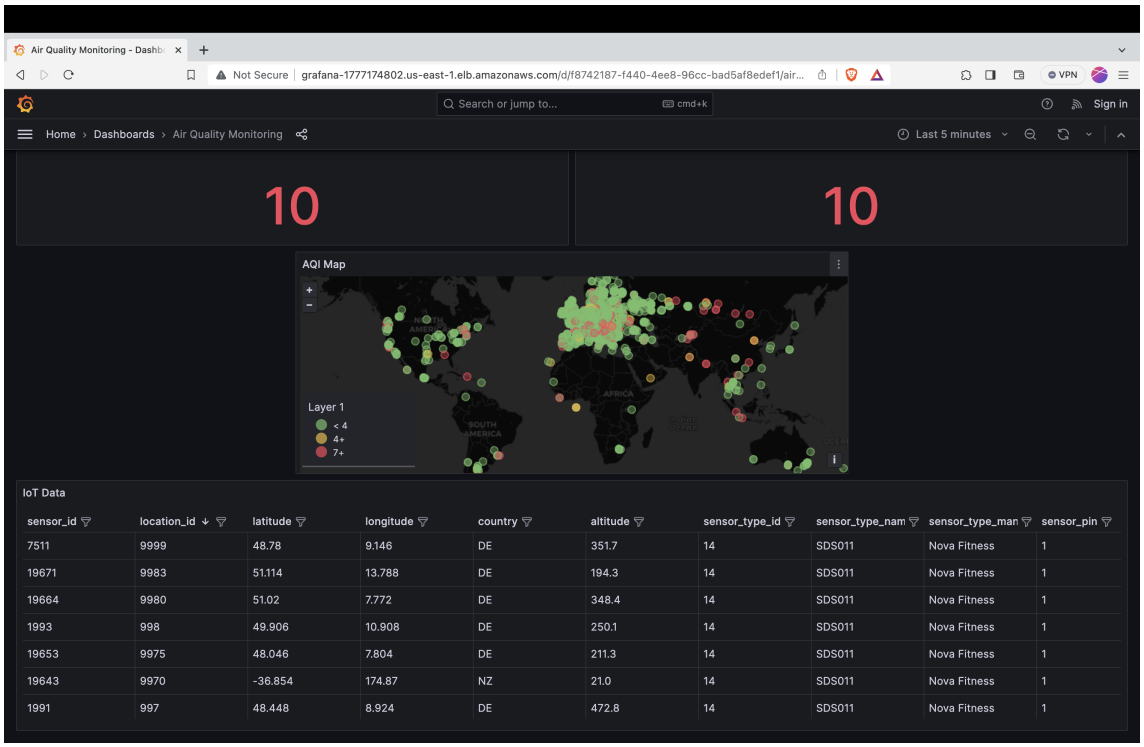


Figure 3.1: Grafana Dashboard Main Panel



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 Main Panel

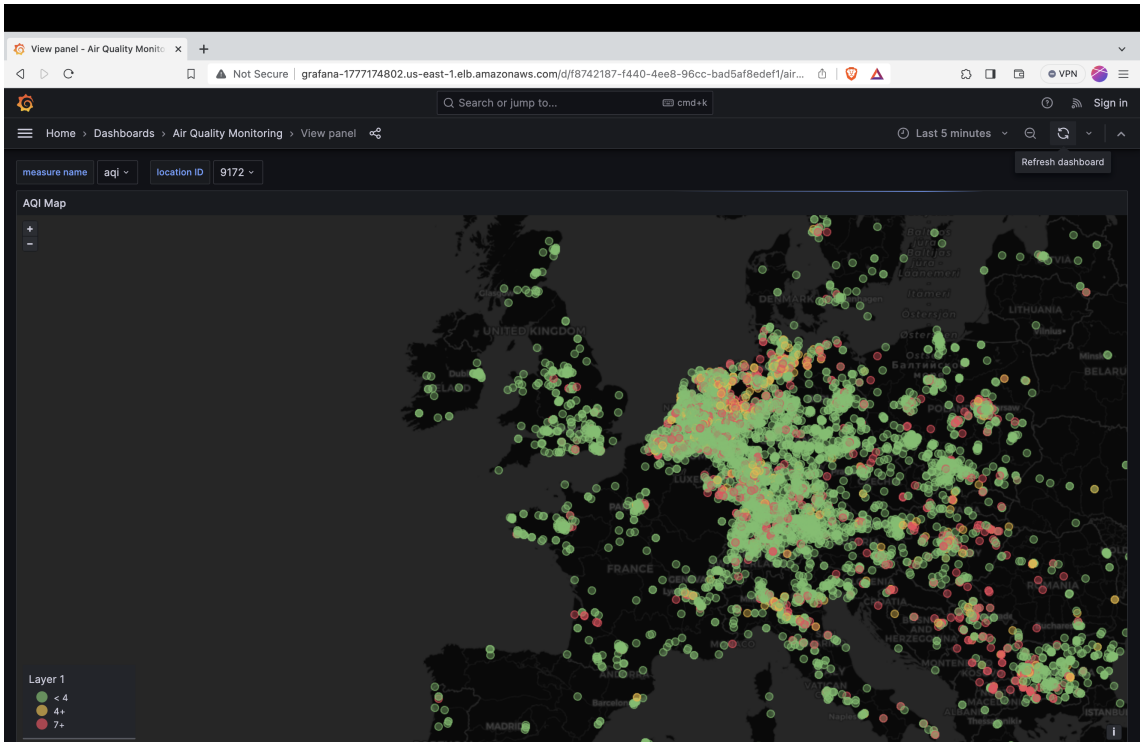


Figure 3.3: Grafana Dashboard AQI Map Panel

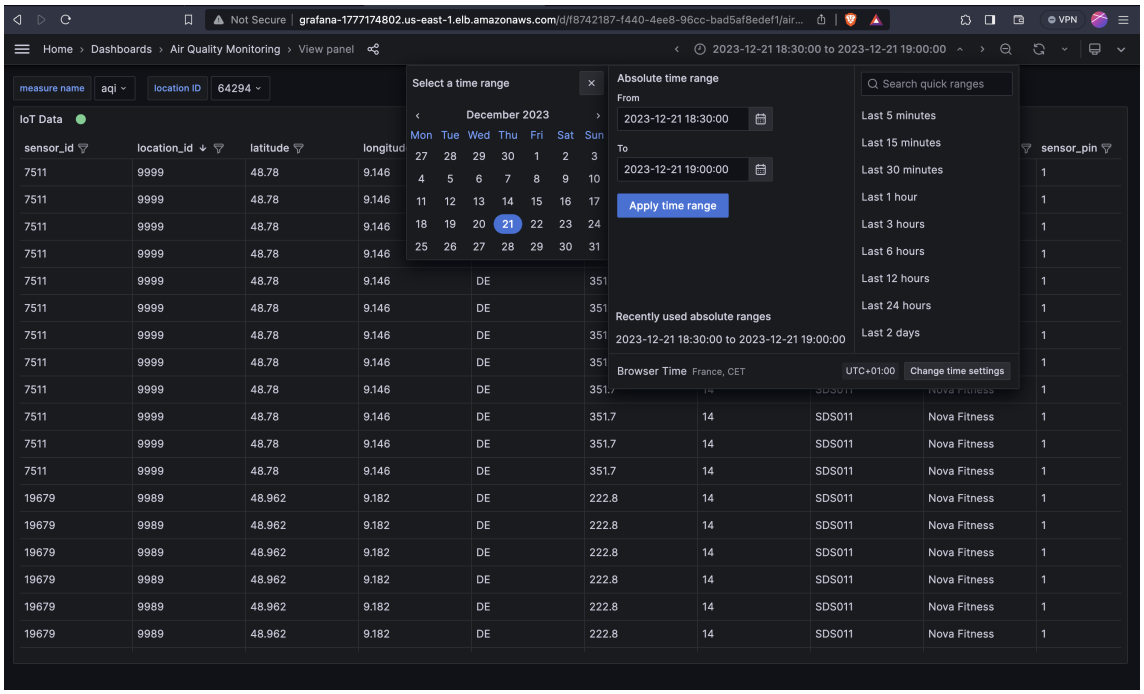


Figure 3.4: Grafana Dashboard IoT Data Panel - Time Range Choice

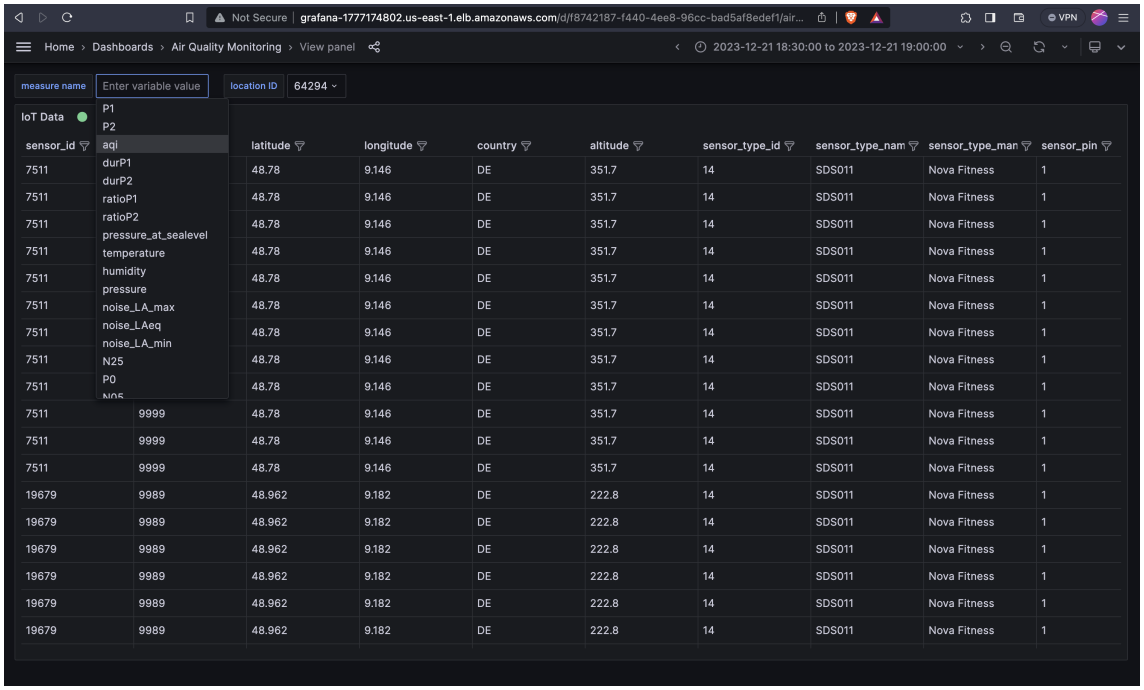


Figure 3.5: Grafana Dashboard IoT Data Panel - Measure Value Choice

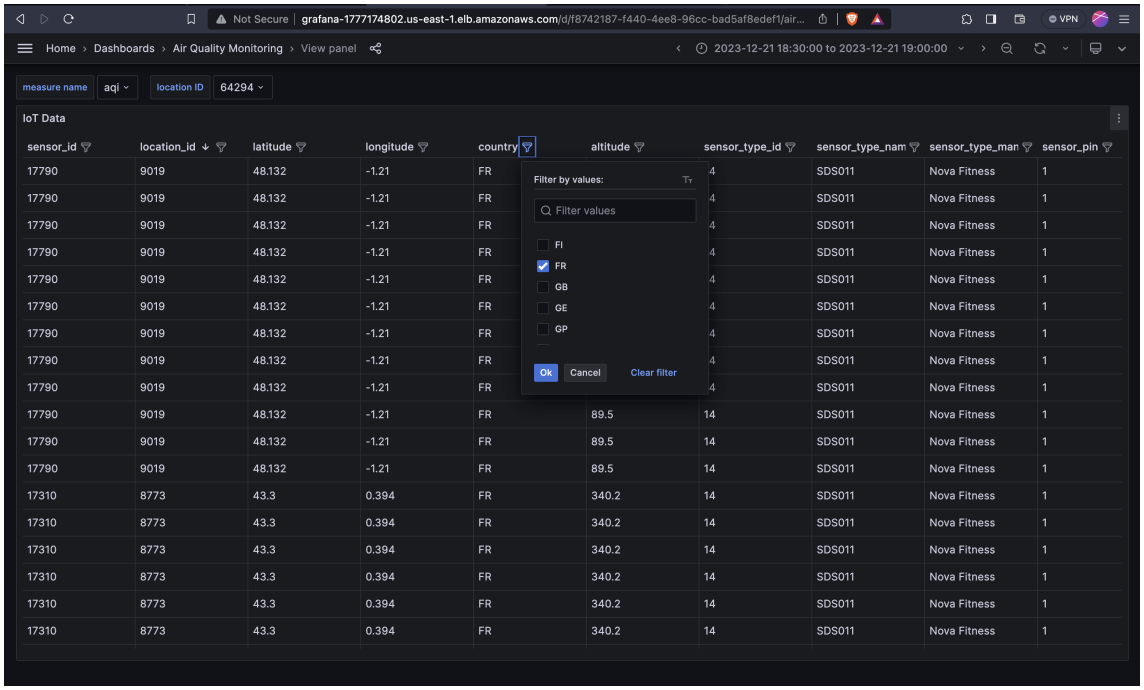


Figure 3.6: Grafana Dashboard IoT Data Panel - Apply a Filter

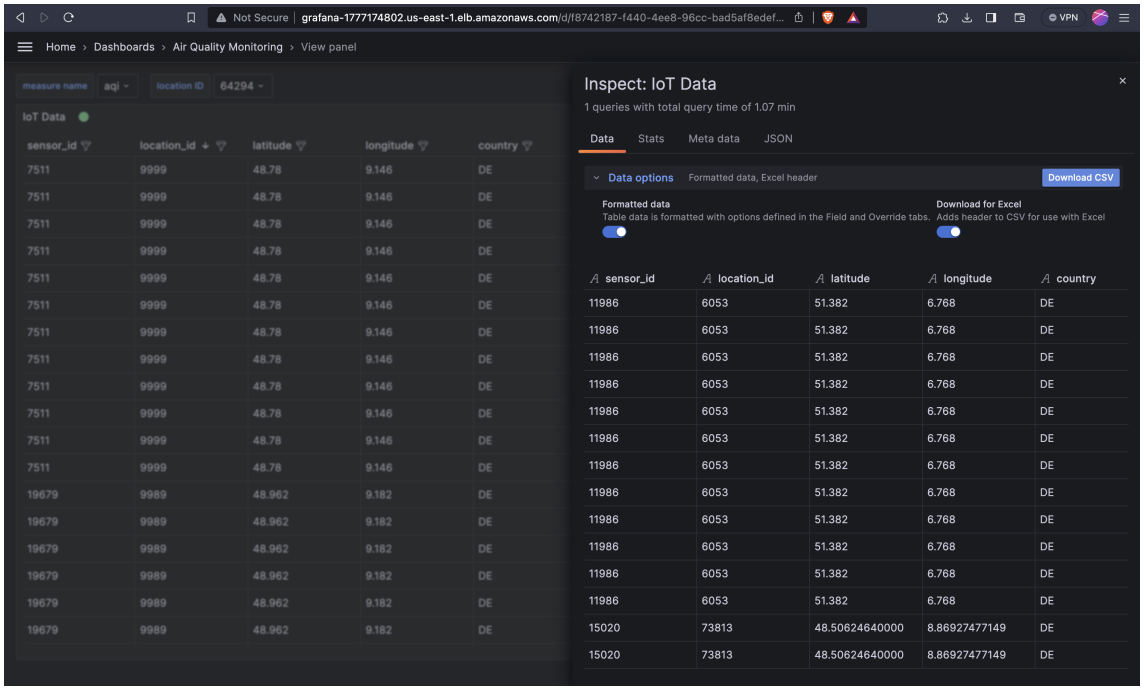


Figure 3.7: Grafana Dashboard IoT Data Panel - Export Values

3.1.2 Load Test

The second query takes around 15 seconds, and the table **??** shows a sample of the data calculated during the task. In order to evaluate performance, an equivalent script not using Spark was run. Execution time was 6 sec. The 3.2 section will cover this point. Locations used to compute the statistics are shown on the map of figure 3.8. The area of the circles is proportional to how the location has been affected by the pandemic.

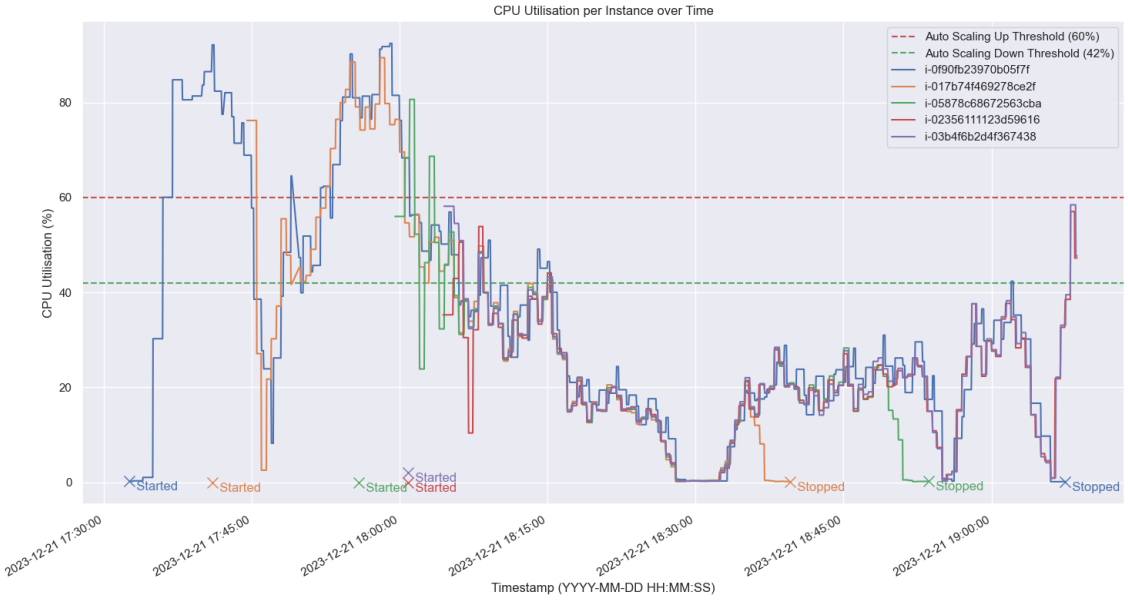


Figure 3.8: Load Test Result

The figures below **??** and **??** show the mean and standard deviation of the number of confirmed cases by week and continent. The results are consistent with expectations: the continents most affected are America and Europe (**?**).

The figures **??** and **??** show the maximum and minimum of the number of confirmed cases by week and continent.

The figures below **??** and **??** show the clusters of the top 50 locations most affected by the pandemic in March 2020. The clusters are represented by different colours.

3.2 Discussion of Results

As the previous results show, it seems that scripts not using Spark are faster. While these differences in execution time may seem surprising at first glance, they can in fact be attributed to several factors:

1. **Data size and structure:** The DataFrame, with its 289 entries and 1147 columns occupying 2.5 MB of memory, is relatively modest in size. Pandas is particularly efficient at processing such large amounts of data in memory on a single node. Spark, on the other hand, is designed for distributed processing of large datasets. In this case, the overheads of distributing the data and managing the Spark environment may outweigh the benefits of using it for smaller datasets.

2. **Operational efficiency:** Pandas performs vectorised operations that are optimised for speed, especially with datasets that fit easily into memory on a single computer. Spark, while powerful for processing large volumes of data, introduces an initial overhead for distributing data and configuring the distributed environment, which can slow down processing for smaller datasets.

3. **Complexity of the environment:** Complexity of the environment: Running operations in a Spark environment involves initializing a cluster (even in local mode), distributing tasks, and managing distributed memory, which adds extra processing time compared to running in-memory Pandas directly.

Thus in this project run locally, Pandas is faster than Spark, due to its efficient management of in-memory operations on a single node. Spark's distributed processing overhead makes it less efficient for such tasks.

3.3 Ethical Considerations and Challenges

Chapter 4

Conclusion

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).

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
ingestion_iot_data_flatten.py
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

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

```
#!/bin/bash

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

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

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

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

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

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

```
#!/bin/bash

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

B.E.2 Services

B.E.2.1 Get IAM Credentials Service

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

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

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

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

B.E.2.2 Spark Python Job Service

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

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

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

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

B.E.2.3 Grafana Server Service

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

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

[Service]
EnvironmentFile=/etc/sysconfig/grafana-server
User=grafana
Group=grafana
Type=notify
Restart=on-failure
WorkingDirectory=/usr/share/grafana
RuntimeDirectory=grafana
RuntimeDirectoryMode=0750
ExecStart=/usr/share/grafana/bin/grafana server
    \
                           --config=${CONF_FILE}
    \
                           --pidfile=${PID_FILE_DIR}/grafana-server.pid
    \
                           --packaging=rpm
    \
                           cfg:default.paths.logs=${LOG_DIR}
    \
                           cfg:default.paths.data=${DATA_DIR}
    \
                           cfg:default.paths.plugins=${PLUGINS_DIR}
    \
                           cfg:default.paths.provisioning=${PROVISIONING_CFG_DIR}
LimitNOFILE=10000
TimeoutStopSec=20
CapabilityBoundingSet=
DeviceAllow=
LockPersonality=true
MemoryDenyWriteExecute=false
NoNewPrivileges=true
PrivateDevices=true
PrivateTmp=true
ProtectClock=true
ProtectControlGroups=true
ProtectHome=true
ProtectHostname=true
ProtectKernelLogs=true
ProtectKernelModules=true
ProtectKernelTunables=true
ProtectProc=invisible
ProtectSystem=full
RemoveIPC=true
RestrictAddressFamilies=AF_INET AF_INET6 AF_UNIX
RestrictNamespaces=true
RestrictRealtime=true
RestrictSUIDSGID=true
SystemCallArchitectures=native
UMask=0027

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

Appendix B.F Load Test Source Codes

B.F.1 Artillery Load Test Configuration File

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

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

```
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)
```