



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 Data Collecting, Processing & Storing . . . . .	2
2.1.1 Overview of the Pipeline Architecture . . . . .	2
2.1.2 Data Collecting . . . . .	3
2.1.2.1 Data Source . . . . .	3
2.1.3 Choice of Database . . . . .	3
2.1.4 Query 3 . . . . .	3
2.2 Queries Optimisation . . . . .	4
2.3 Pipeline of the Project . . . . .	5
2.3.1 Pipeline Overview . . . . .	5
2.3.2 Pipeline Orchestration . . . . .	6
<b>3 Results &amp; Discussion</b>	<b>9</b>
3.1 Queries Results . . . . .	9
3.1.1 Query 1 . . . . .	9
3.1.2 Query 2 . . . . .	11
3.1.3 Query 3 . . . . .	14
3.2 Discussion of Results . . . . .	16
3.3 Ethical Considerations and Challenges . . . . .	17
<b>4 Conclusion</b>	<b>18</b>
<b>References</b>	<b>19</b>
<b>A Documentation</b>	<b>20</b>
<b>B Ingestion, Processing &amp; Storing Pipeline Source Code</b>	<b>23</b>
<b>C Data Collecting Source Code</b>	<b>25</b>

<b>D</b>	<b>Data Processing Source Code</b>	<b>27</b>
<b>E</b>	<b>Data Storing Source Code</b>	<b>31</b>
<b>F</b>	<b>Scripts &amp; Services</b>	<b>37</b>
	F.A Scripts . . . . .	37
	F.B Services . . . . .	38

# List of Figures

2.1	Data Collecting, Processing & Storing Pipeline Diagram . . . . .	2
2.2	Data Distributing Pipeline Diagram . . . . .	7
2.3	Apache Airflow DAG Graph . . . . .	8
3.1	Mean Daily Confirmed Cases Per Month . . . . .	10
3.2	Top 100 Locations most affected by the pandemic . . . . .	11
3.3	Mean Confirmed Cases By Week and Continent . . . . .	12
3.4	Standard Deviation Confirmed Cases By Week and Continent . . . . .	12
3.5	Maximum Confirmed Cases By Week and Continent . . . . .	13
3.6	Minimum Confirmed Cases By Week and Continent . . . . .	13
3.7	Top 50 Locations most affected by the pandemic . . . . .	14
3.8	Custom KMeans Clustering on 03/2020 . . . . .	15
3.9	Spark MLlib KMeans Clustering on 03/2020 . . . . .	15

# List of Tables

3.1	Query 1 Results Sample . . . . .	9
3.2	Query 2 Results Sample . . . . .	11
3.3	Query 3 Results Sample - Custom KMeans Clustering . . . . .	14
3.4	Query 3 Results Sample - Spark MLlib KMeans Clustering . . . . .	14

# 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.

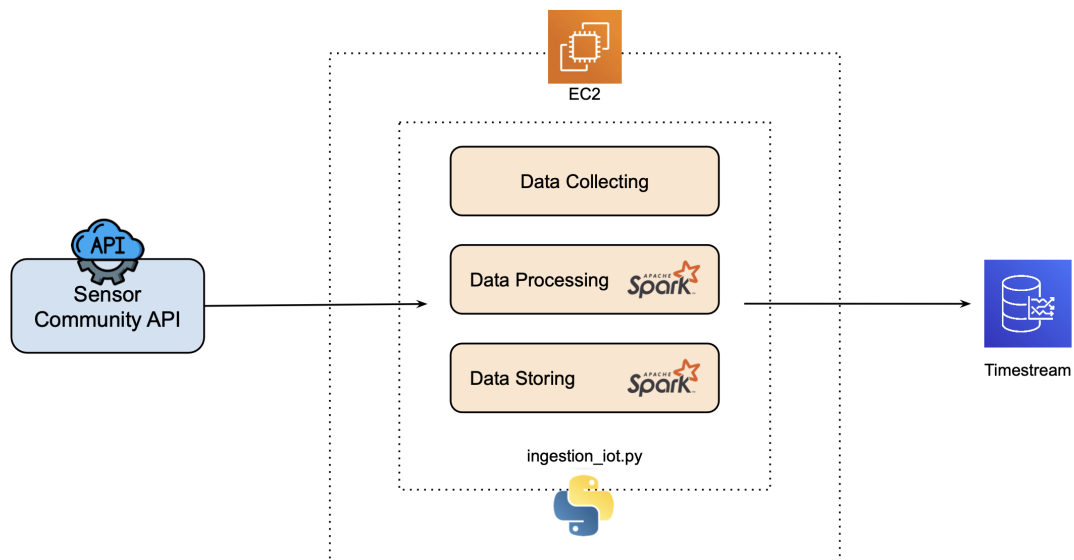


Figure 2.1: Data Collecting, Processing & Storing Pipeline Diagram

## 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

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.

24 Hour Averaged Data API: This API (<https://data.sensor.community/static/v2/data.24h.json>) 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.

5-Minute Averaged Data API: The API (<https://data.sensor.community/static/v2/data.json>) 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.

### 2.1.3 Choice of Database

This project uses Amazon Timestream, which is a cloud-native time series database. It is a wise choice because of its superior time series management capabilities, which are particularly well suited to data from IoT sensors. This database is distinguished by its speed of ingestion and efficiency in processing vast volumes of data, facilitating regular updates and in-depth analyses of the Air Quality Index. Its ability to adjust to variations in workload is a major asset, ensuring consistent performance. What's more, Timestream's advanced security features meet strict standards of confidentiality and data sovereignty, an essential criterion for the secure management of sensitive information.

### 2.1.4 Query 3



## **2.2 Queries Optimisation**

## 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.

### 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 2.3 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.

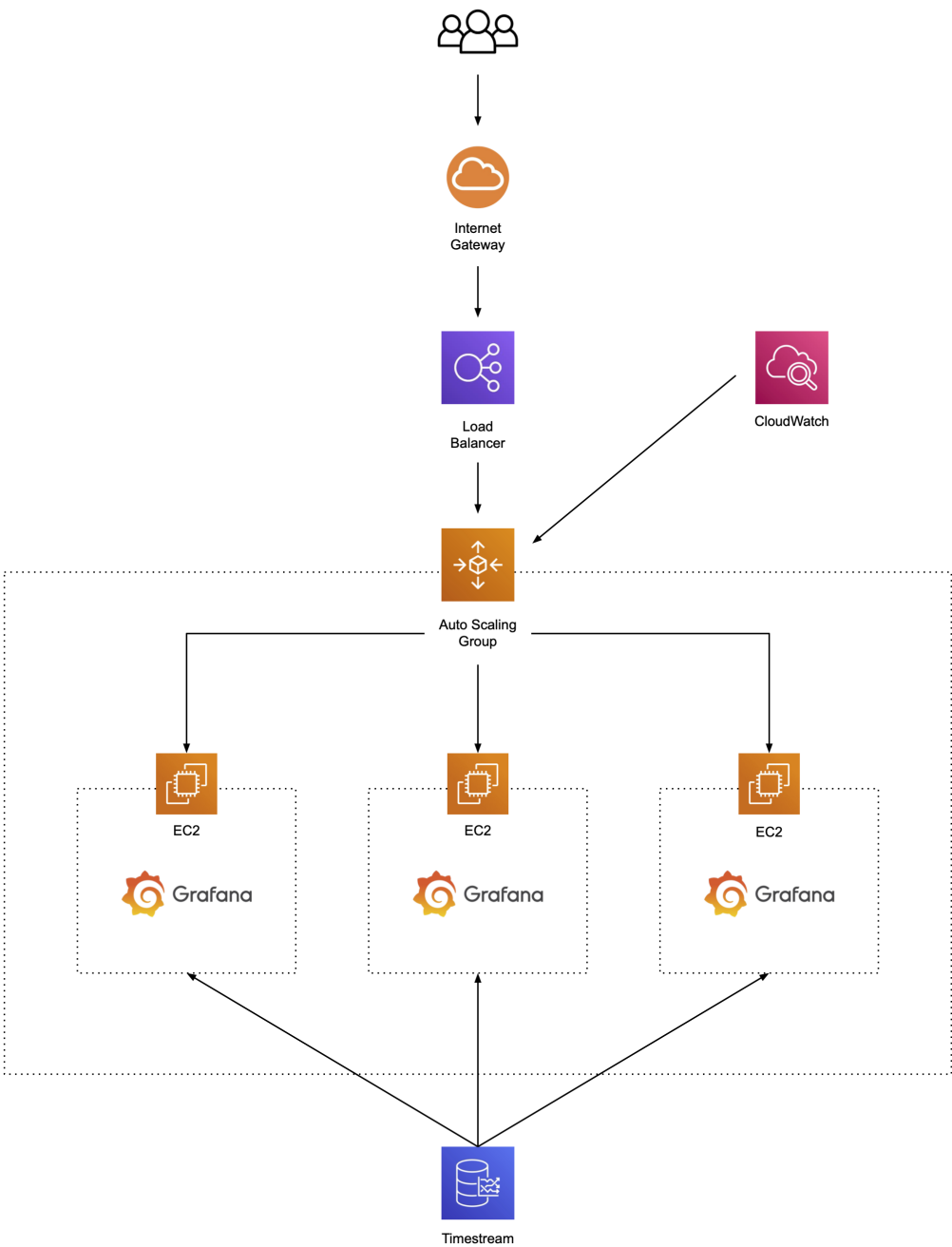


Figure 2.2: Data Distributing Pipeline Diagram

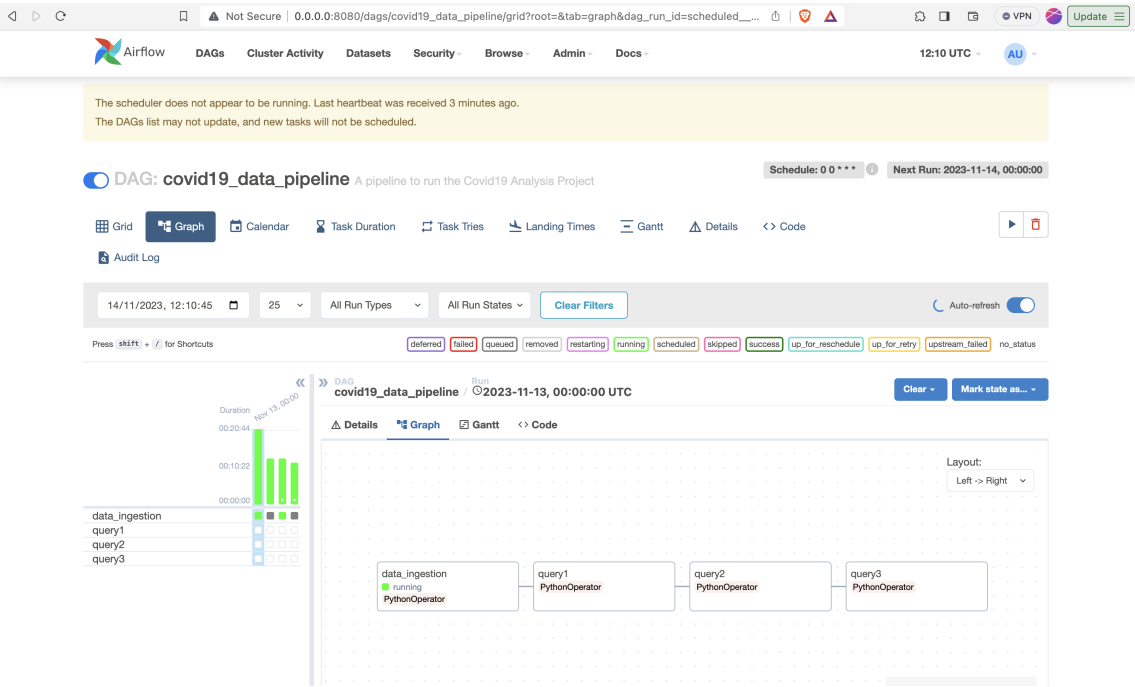


Figure 2.3: Apache Airflow DAG Graph

# Chapter 3

## Results & Discussion

### 3.1 Queries Results

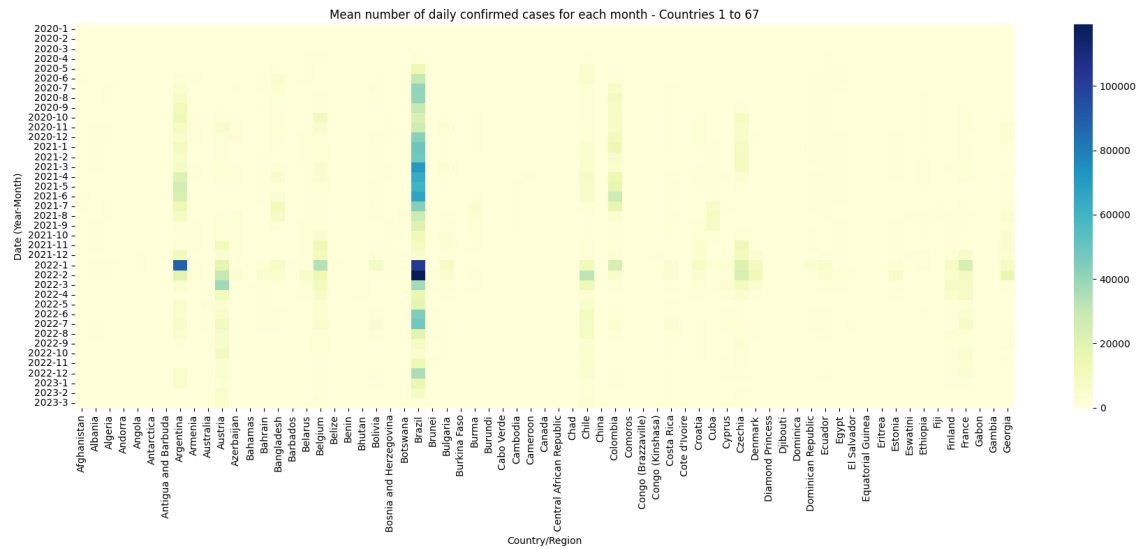
The programme was last run on 18 November 2023. The appendix E shows the output of the program on the terminal.

#### 3.1.1 Query 1

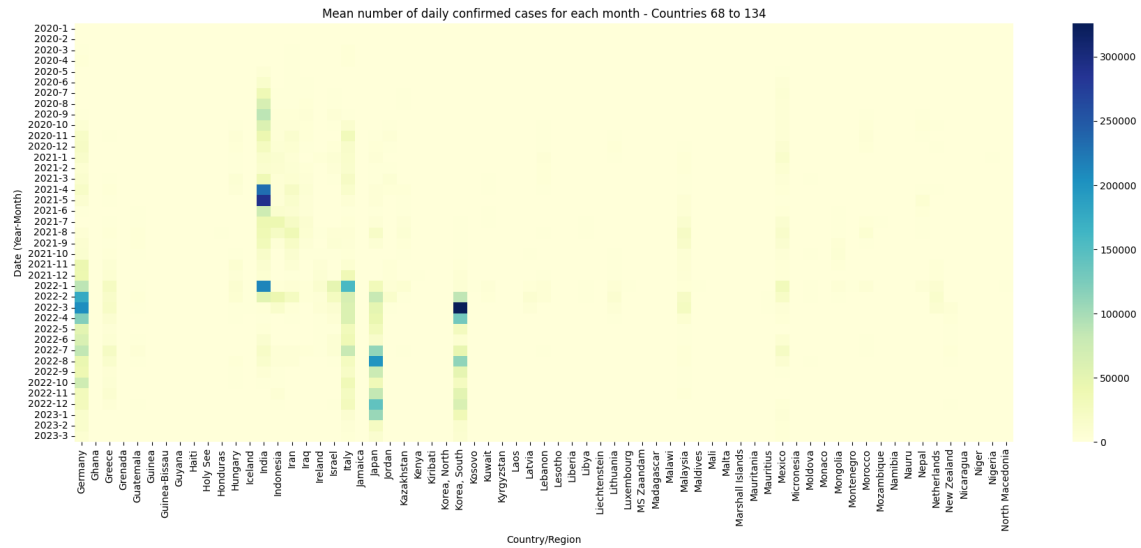
The first query takes around 3 seconds, and the table 3.1 shows a sample of the data calculated during the task. In order to evaluate performance, an equivalent script not using Spark was run. Execution time was 0.5 sec. The 3.2 section will cover this point. The results are consistent with what was expected (? ). For example, the figure 3.1a shows that Brazil was heavily impacted by the pandemic, reaching an average peak in February 2022. The same is true for Korea in figure 3.1b and the United States in figure 3.1c, which were heavily impacted.

Table 3.1: Query 1 Results Sample

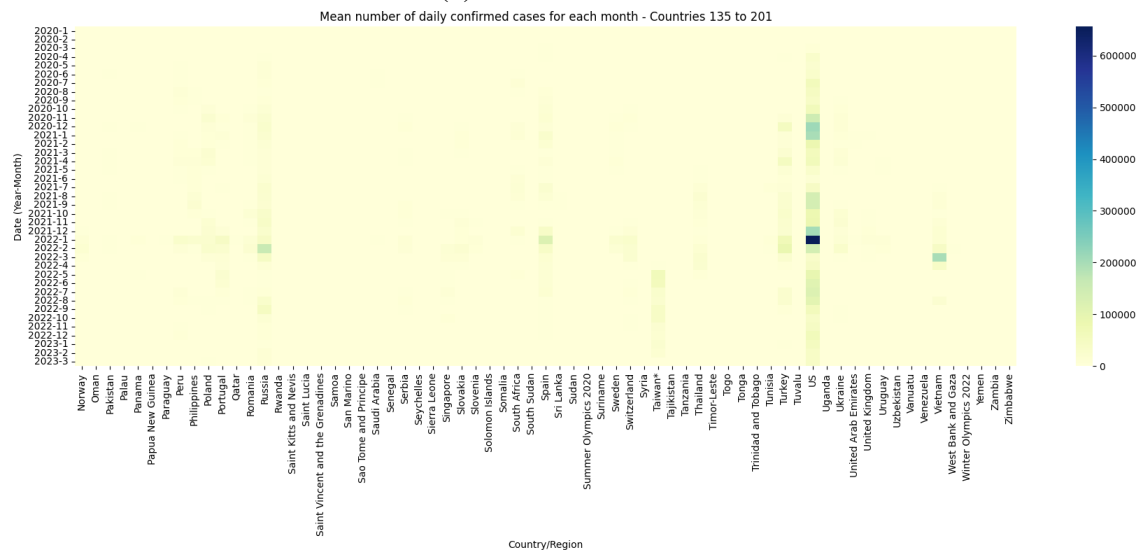
	Country/Region	Year	Month	Average
1	Afghanistan	2020	1	0.0
2	Afghanistan	2020	2	0.1724137931034483
3	Afghanistan	2020	3	5.193548387096774
4	Afghanistan	2020	4	55.36666666666667
5	Afghanistan	2020	5	430.741935483871



(a) Countries 1 to 67



(b) Countries 68 to 134



(c) Countries 135 to 201

Figure 3.1: Mean Daily Confirmed Cases Per Month

### 3.1.2 Query 2

The second query takes around 15 seconds, and the table 3.2 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 3.2. The area of the circles is proportional to how the location has been affected by the pandemic.

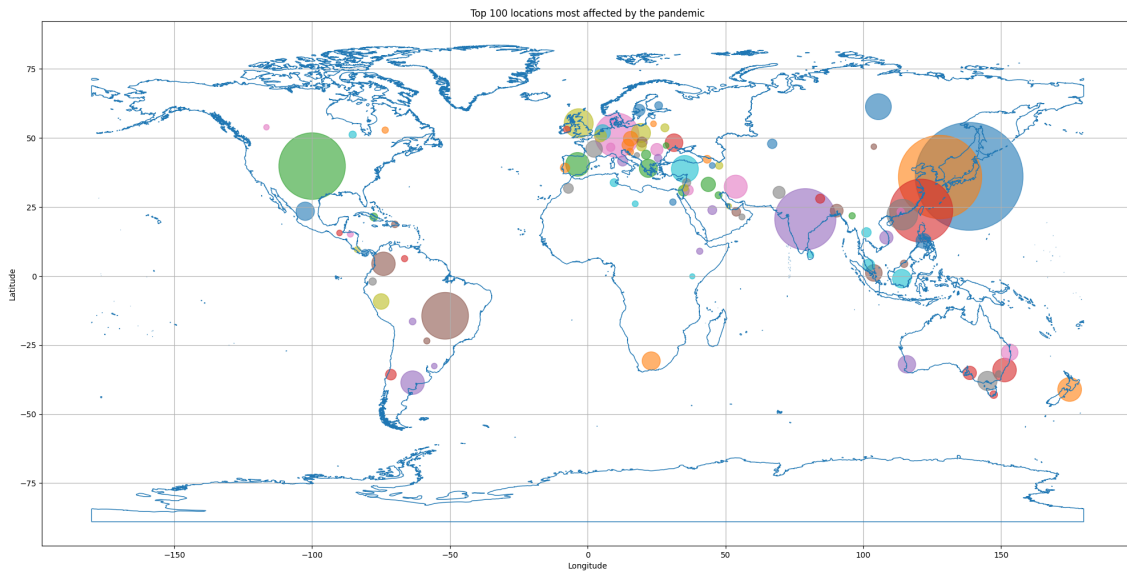


Figure 3.2: Top 100 Locations most affected by the pandemic

Table 3.2: Query 2 Results Sample

	Continent	WeekRange	Mean	Std	Min	Max
1	Africa	19/01/2020-25/01/2020	0.0	0.0	0	0
2	Africa	26/01/2020-01/02/2020	0.0	0.0	0	0
3	Africa	02/02/2020-08/02/2020	0.0	0.0	0	0
4	Africa	09/02/2020-15/02/2020	0.0	0.0	0	0
5	Africa	16/02/2020-22/02/2020	0.0	0.0	0	0
6	Africa	23/02/2020-29/02/2020	0.0	0.0	0	0
7	Africa	01/03/2020-07/03/2020	0.02857	0.16903	0	1
8	Africa	08/03/2020-14/03/2020	0.65714	1.73108	0	9
9	Africa	15/03/2020-21/03/2020	2.74285	4.53964	0	17
10	Africa	22/03/2020-28/03/2020	12.62857	15.33754	0	59
11	Africa	29/03/2020-04/04/2020	14.97142	19.31242	0	82



The figures below 3.3 and 3.4 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 (? ).

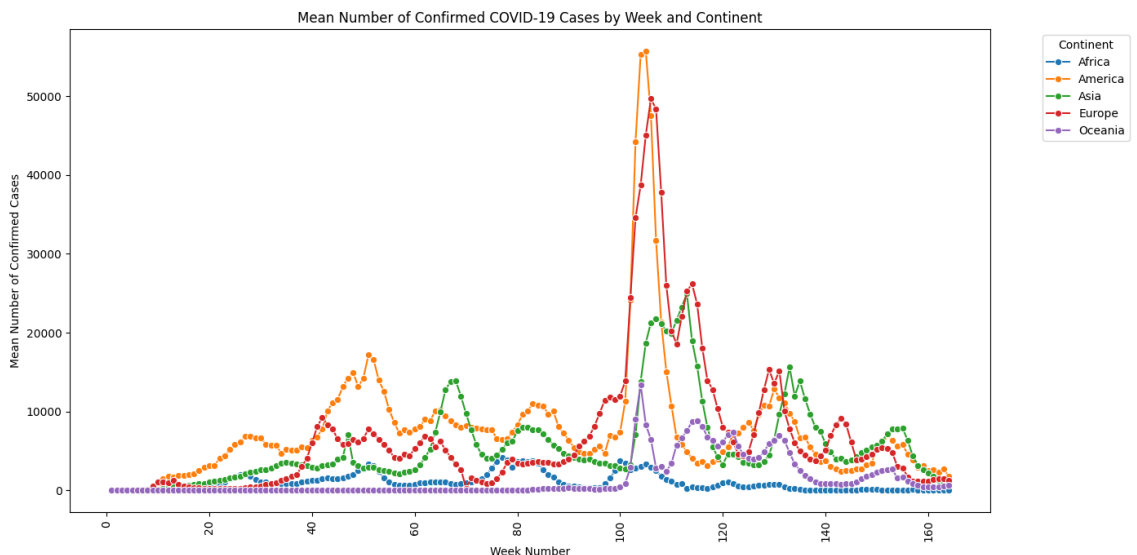


Figure 3.3: Mean Confimed Cases By Week and Continent

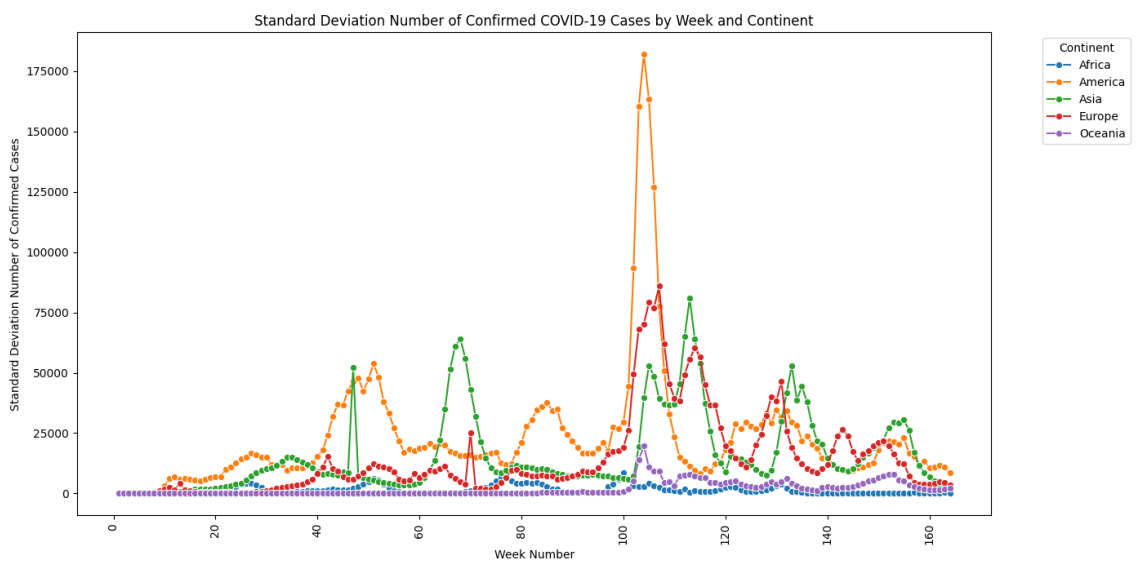


Figure 3.4: Standard Deviation Confimed Cases By Week and Continent

The figures 3.5 and 3.6 show the maximum and minimum of the number of confirmed cases by week and continent.

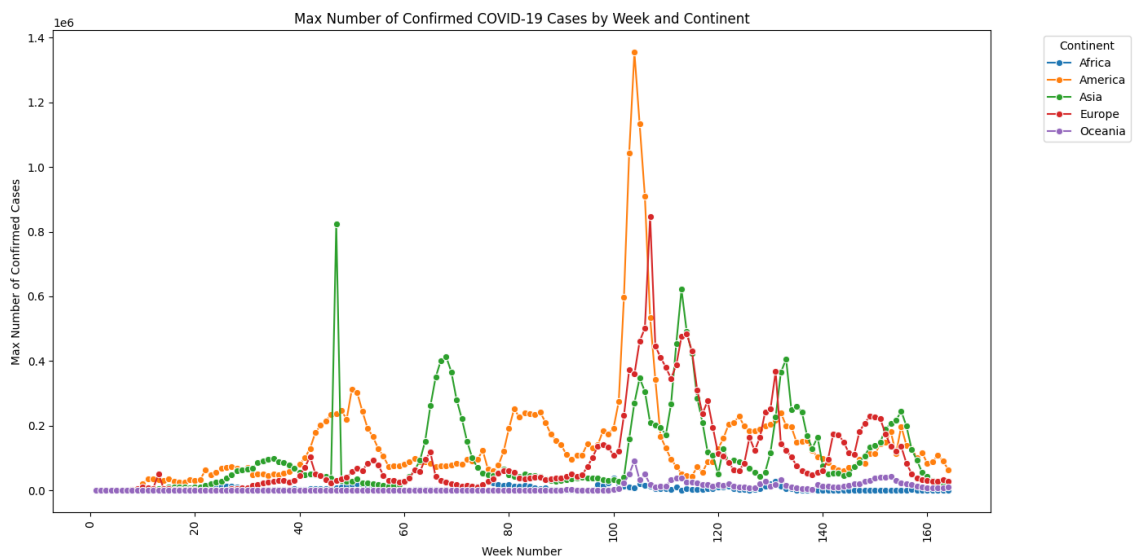


Figure 3.5: Maximum Confirmed Cases By Week and Continent

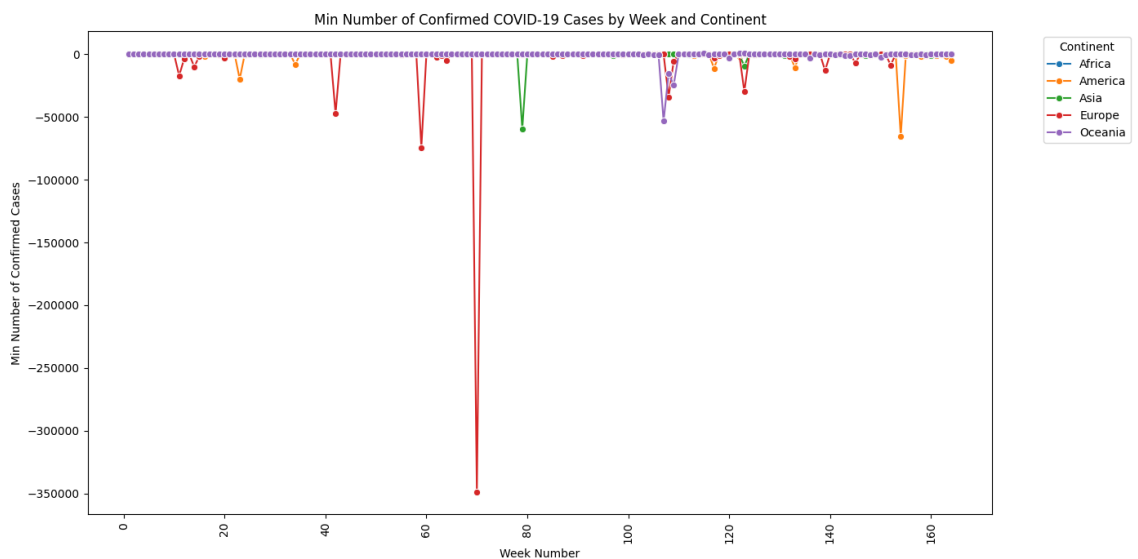


Figure 3.6: Minimum Confirmed Cases By Week and Continent

### 3.1.3 Query 3

The third query takes around 3 minutes as clustering with the custom implementation takes 60 seconds and clustering with the Spark MLlib implementation takes 110 seconds. The table 3.3 shows a sample of the data calculated during the task with the custom implementation and the table 3.4 with the Spark MLlib implementation. Locations used to compute the statistics are shown on the map of figure 3.7. The area of the circles is proportional to how the location has been affected by the pandemic.

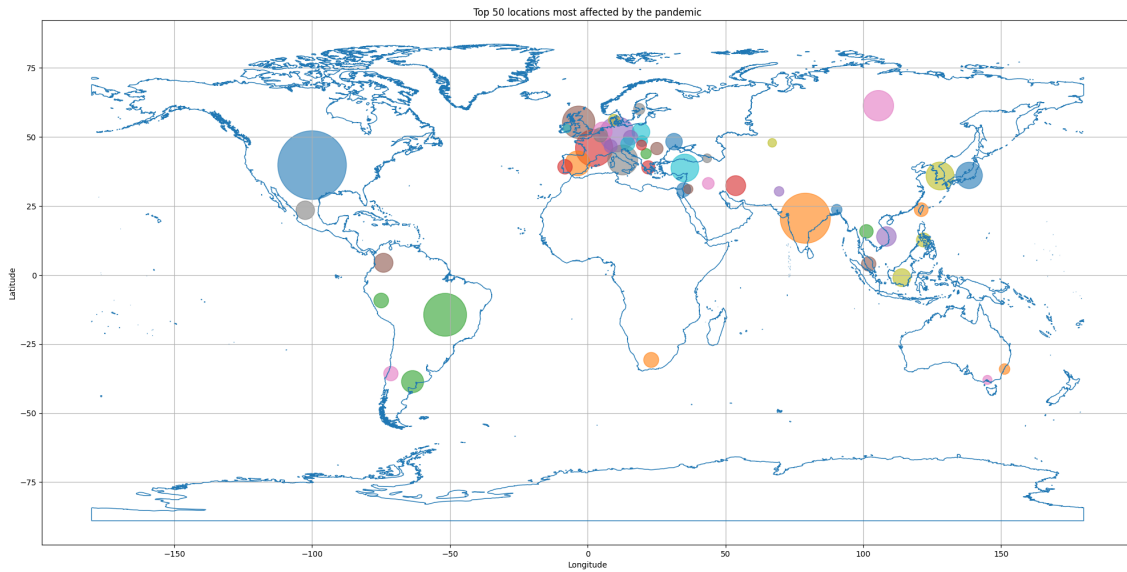


Figure 3.7: Top 50 Locations most affected by the pandemic

Table 3.3: Query 3 Results Sample - Custom KMeans Clustering

	Location	Month	Cluster
1	Argentina	2020-01	2
2	Austria	2020-01	2
3	Brazil	2020-01	2
4	Czechia	2020-01	2
5	France	2020-01	1

Table 3.4: Query 3 Results Sample - Spark MLlib KMeans Clustering

	Location	Month	Cluster
1	Argentina	2020-01	2
2	Austria	2020-01	0
3	Brazil	2020-01	2
4	Czechia	2020-01	2
5	France	2020-01	1

The figures below 3.8 and 3.9 show the clusters of the top 50 locations most affected by the pandemic in March 2020. The clusters are represented by different colours.

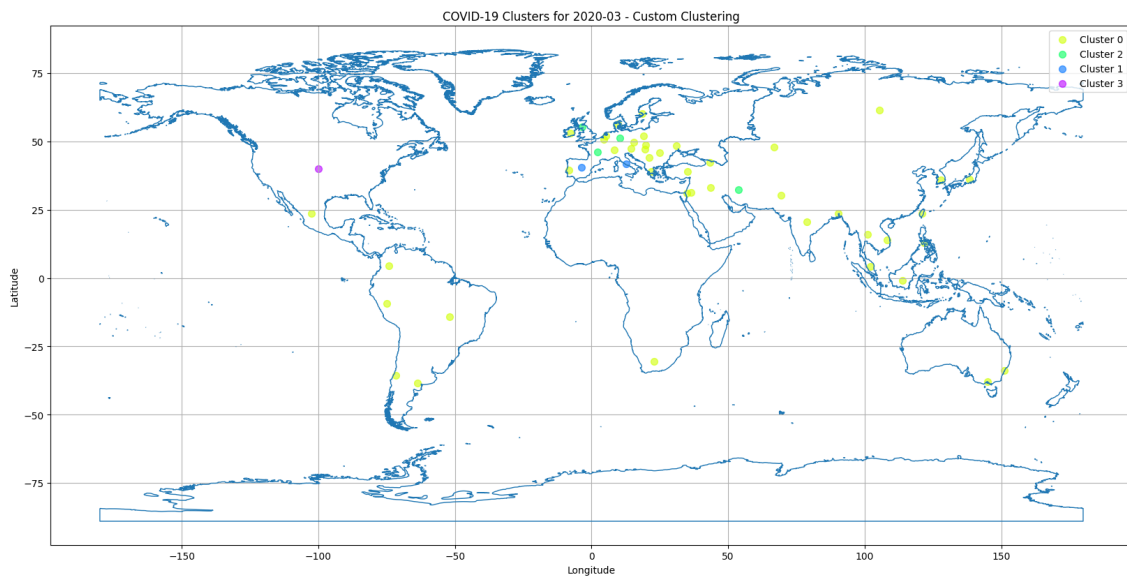


Figure 3.8: Custom KMeans Clustering on 03/2020

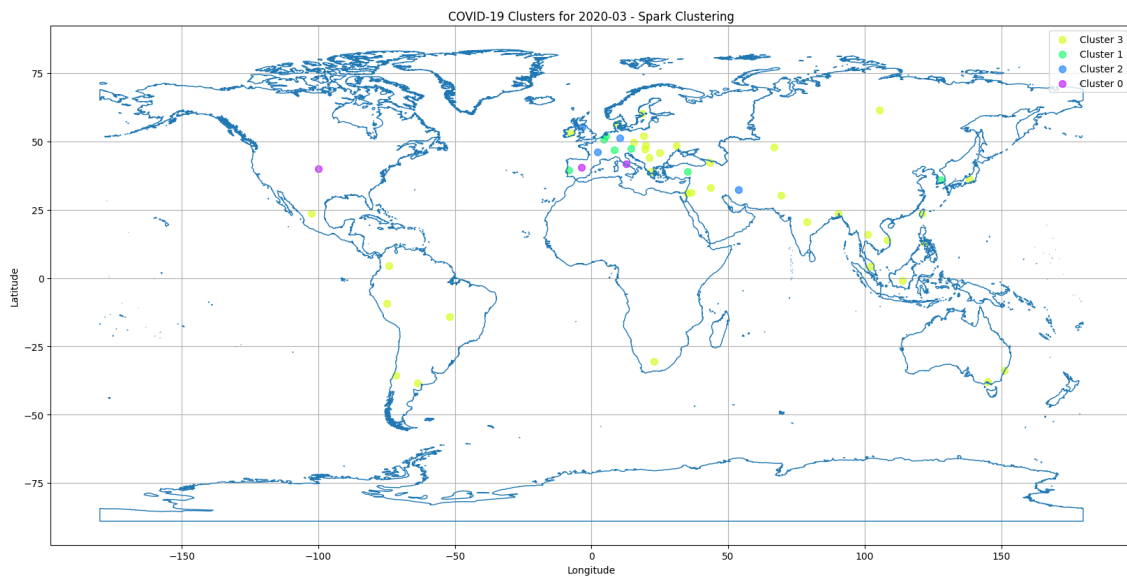


Figure 3.9: Spark MLlib KMeans Clustering on 03/2020

## 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

In the context of the COVID-19 pandemic, the use of Machine Learning and Big Data, while beneficial for crisis management, raises complex ethical issues and challenges.

Firstly, the management of health data, which is essential for tracking and preventing the spread of the virus, creates major risks for privacy. preventing the spread of the virus, creates major privacy risks. Massive data collection and analysis can lead to intrusive surveillance, where the boundaries between public security and individual privacy become blurred. This can constitute an infringement of the right to privacy. In addition, some countries have taken the opportunity to monitor their populations. For example, China has used facial recognition technology to track people's movements and apply quarantine measures.

Secondly, algorithmic biases represent a major challenge. Machine learning systems, although powerful, can incorporate and amplify existing biases in the data. In the context of the pandemic, this could result in inequitable distribution of medical resources, biased diagnoses or discriminatory public health policies, disproportionately affecting certain social or ethnic groups.

Thirdly, transparency and accountability are crucial issues. Complex and often opaque algorithms make it difficult to understand and challenge AI-based decisions. This raises questions of governance and regulation: who is responsible for errors or harm caused by automated decisions? How can these technologies be effectively controlled and regulated?

Finally, there is the overall challenge of striking a balance between the benefits of using these technologies and respect for fundamental rights. Striking the right balance between effective management of the pandemic and protection of individual freedoms is a delicate exercise, requiring careful ethical reflection and judicious regulation.

# Chapter 4

## Conclusion

This study demonstrated the effectiveness of Big Data and Machine Learning technologies in the analysis and management of the COVID-19 pandemic. By exploiting the potential of Apache Spark to process large datasets, the analysis revealed significant trends in the spread of the virus, providing essential information for public health decisions. The methodologies adopted provided an in-depth understanding of the dynamics of the pandemic, highlighting geographical and temporal variations in confirmed cases. At the same time, the ethical challenges raised, such as privacy protection and algorithmic fairness, underline the need for a balanced and regulated approach to the use of information technologies. In short, this project illustrates how the judicious integration of AI and Big Data can transform our response to health emergencies, while reminding us of the importance of ethical responsibility in the application of technological advances.

# References



# Appendix A

## Documentation

### 1. 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
artillery_load_test.yml
ingestion_iot_data_flatten.py
main.py
README.md
requirements.txt
```

### 2. 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).

### 3. Detailed Features of Classes and Functions

#### Classes

**Spark:** Class to create a Spark session and load the data set into a DataFrame. This class has the following methods:

- `__init__(self, appName, master)`: Constructor to initialise the class with an application name and a master URL.
- `getSpark(self)`: Method to return the Spark session.
- `getSparkDf(self, path)`: Method to load the data set into a DataFrame and return it, given a path.
- `stopSpark(self)`: Method to stop the Spark session.

**Query1:** Class to perform Query 1, which is to calculate the average number of confirmed daily cases of people infected with COVID-19 for each country in the dataset, and save the results to a CSV file, using Spark. This class has the following methods:

- `__init__(self, sparkSession, covidDataDf)`: Constructor to initialise the class with a Spark session and a DataFrame containing COVID-19 data.
- `run(self)`: Method to run the query.

**Query2:** Class to perform Query 2, which is to calculate the mean, standard deviation, minimum and maximum of the number of confirmed cases daily for each week and continent, and save the results to a CSV file, using Spark. This class has the following methods:

- `__init__(self, sparkSession, covidDataDf)`: Constructor to initialise the class with a Spark session and a DataFrame containing COVID-19 data.
- `run(self)`: Method to run the query.

**Query3:** Class to perform Query 3, which is to perform clustering on the top 50 affected locations based on the maximum slope of monthly confirmed cases, and save the results to a CSV file, using Spark. This class has the following methods:

- `__init__(self, sparkSession, covidDataDf)`: Constructor to initialise the class with a Spark session and a DataFrame containing COVID-19 data.
- `run(self)`: Method to run the query.

**CustomKMeans:** Class to perform a custom KMeans clustering algorithm. This class has the following methods:

- `__init__(self, k, max_iter)`: Constructor to initialise the class with the number of clusters and the maximum number of iterations.
- `fit(self, X)`: Method to fit the model to the data.
- `predict(self, X)`: Method to predict the cluster of each data point.

CustomQuery1: Class to perform Query1, using Pandas without Spark. This class has the following methods:

- `__init__(self, covidDataDf)`: Constructor to initialise the class with a DataFrame containing COVID-19 data.
- `run(self)`: Method to run the query.

CustomQuery2: Class to perform Query2, using Pandas without Spark. This class has the following methods:

- `__init__(self, covidDataDf)`: Constructor to initialise the class with a DataFrame containing COVID-19 data.
- `run(self)`: Method to run the query.

### Functions

`ingestion.py`

- `fetch_covid_data()`: Function to ingest the data set from the source and store it in the data lake.

`utils.py`

- `getContinentByCoordinates(longitude, latitude)`: Function used by CustomQuery2 class to assign a continent to a location based on its coordinates.
- `compute_slope(row)`: Function used by CustomQuery2 class to compute the slope of a location.
- `format_date_range(date)`: Function used by CustomQuery2 class to format a date range.

## Appendix B

# 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
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 2 minutes.\n"
68         n"
69     )
70     time.sleep(10)
71 except Exception as e:
72     print(f"Exception: {e}")

```

# Appendix C

## 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
11     Spark DataFrame
12
13     Args:
14         sparkSession (SparkSession): The SparkSession instance
15
16     Returns:
17         df (DataFrame): The DataFrame containing the last data from
18         the sensors
19     """
20     # Fetches the latest data from the data.sensor.community API
21     url = "https://data.sensor.community/static/v2/data.json"
22     # Use a session to avoid creating a new connection for each
23     request
24     session = Session()
25     try:
26         print(
27             dt.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
28             + " 1. Fetching the latest data..."
29         )
30         response = session.get(url)
31         # If the response was successful, no Exception will be
32         raised
33         if response.status_code == 200 and response.content:
34             # Convert the response to a Spark DataFrame
35             df = sparkSession.read.option("multiline", "true").json(
36                 (
37                     sparkSession.sparkContext.parallelize([response.
38                         text])
39                 )
40             )
```

```
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 D

## 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
86         sensors
87
88     Returns:
89         df_grouped (DataFrame): The DataFrame containing the AQI
90         for each sensor
91     """
92
93     print(dt.datetime.now().strftime("%Y-%m-%d %H:%M:%S") + " 2.
94     Computing the AQI...")
95     df_exploded = df.withColumn(
96         "sensordatavalue",

```

```

94     F.explode("sensordatavalues"), # Explode the
sensordatavalues column
95     ).withColumn(
96         "aqi",
97         F.when(
98             F.col("sensordatavalue.value_type") == "P1",
99             get_aqi_value_p25(
100                 F.col("sensordatavalue.value").cast(FloatType())
101             ), # Cast the value to float and compute the AQI of
PM2.5
102         ).when(
103             F.col("sensordatavalue.value_type") == "P2",
104             get_aqi_value_p10(
105                 F.col("sensordatavalue.value").cast(FloatType())
106             ), # Cast the value to float and compute the AQI of
PM10
107         ),
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 timestamp
112         .agg(
113             F.first("id").alias("id"),
114             F.first("location").alias("location"),
115             F.first("sensor").alias("sensor"),
116             F.max("aqi").alias("aqi"), # Compute the maximum AQI
between PM2.5 and PM10
117             F.collect_list("sensordatavalue").alias("
sensordatavalues"),
118         ) # Aggregate the AQI and the sensordatavalues
119         .selectExpr(
120             "sensor.id as sensor_id",
121             "sensor.pin as sensor_pin",
122             "sensor.sensor_type.id as sensor_type_id",
123             "sensor.sensor_type.manufacturer as
sensor_type_manufacturer",
124             "sensor.sensor_type.name as sensor_type_name",
125             "location.country as country",
126             "location.latitude as latitude",
127             "location.longitude as longitude",
128             "location.altitude as altitude",
129             "location.id as location_id",
130             "aqi",
131             "sensordatavalues",
132             "timestamp",
133         ) # Select the columns to keep
134     )
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 E

## 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
16     only the updated 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
22         stored
23
24     Returns:
25         df_updated (DataFrame): The DataFrame containing only the
26         updated rows
27     """
28     print(
29         dt.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
30         + " 3. Filtering the data to keep only the updated rows..."
31     )
32     query = """
33     SELECT sensor_id, MAX(time) as last_timestamp
34     FROM {}.{}
35     GROUP BY sensor_id
36     """
37     .format(
38         database_name, table_name
39     )
```

```

38 # Initialize the boto3 client
39 session = boto3.Session() # Create a boto3 session
40 query_client = session.client(
41     "timestream-query", config=Config(region_name="us-east-1")
42 ) # Create a boto3 client
43 paginator = query_client.get_paginator("query") # Create a
paginator
44
45 # Get the last timestamp for each sensor
46 last_timestamps = (
47     {}
48 ) # Initialize a dictionary to store the last timestamp for
each sensor
49 response_iterator = paginator.paginate(QueryString=query) #
Paginate the query
50 for response in response_iterator:
51     for row in response["Rows"]:
52         sensor_id = row["Data"][0]["ScalarValue"]
53         last_timestamps[sensor_id] = row["Data"][1]["
ScalarValue"]
54
55 # If there is no data in Timestream, return the DataFrame as is
56 if len(last_timestamps) == 0:
57     print("No data in Timestream")
58     return df
59
60 # Define an UDF to check if the row is updated
61 @F.udf(returnType=BooleanType())
62 def isUpdated(sensor_id, timestamp):
63     """
64     Checks if the row is updated
65
66     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"
][["HTTPStatusCode"]]
131         )
132     except client.exceptions.RejectedRecordsException as err:
133         _print_rejected_records_exceptions(err)
134     except Exception as err:
135         print("Error:", err)
136
137
138 def writeToTimestream(database_name, table_name, partitioned_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         partitioned_df (DataFrame): The DataFrame containing the
146         data to be stored
147     """
148
149     # Initialize the boto3 client for each partition
150     session = boto3.Session()
151     write_client = session.client(
152         "timestream-write",
153         config=Config(
154             read_timeout=20, max_pool_connections=5000, retries={"
155             max_attempts": 10}
156         ),
157     )
158
159     # Create a list of records
160     records = []
161     for row in partitioned_df:
162         try:
163             # Skip rows that are not of type Row
164             if not isinstance(row, Row):
165                 continue
166
167             # Convert timestamp to Unix epoch time in milliseconds
168             timestamp_datetime = dt.datetime.strptime(
169                 row.timestamp, "%Y-%m-%d %H:%M:%S"
170             )
171             row_timestamp = str(int(timestamp_datetime.timestamp()
172 * 1000))
173
174             # altitude
175             altitude = row.altitude if row.altitude != "" else 0
176
177             # Create dimensions list
178             dimensions = [
179                 {"Name": "country", "Value": str(row.country)},
180                 {"Name": "latitude", "Value": str(row.latitude)},
181                 {"Name": "longitude", "Value": str(row.longitude)},
182                 {"Name": "altitude", "Value": str(altitude)},
183                 {"Name": "location_id", "Value": str(row.
184 location_id)},
185                 {"Name": "sensor_id", "Value": str(row.sensor_id)},
186                 {"Name": "sensor_pin", "Value": str(row.sensor_pin)
187             },
188             {
189                 "Name": "sensor_type_manufacturer",
190                 "Value": str(row.sensor_type_manufacturer),
191             },
192             {"Name": "sensor_type_name", "Value": str(row.
193 sensor_type_name)},

```

```

188         {"Name": "sensor_type_id", "Value": str(row.
189         sensor_type_id)},
190     ]
191
192     # Create a record for each measurement
193     measuresValues = []
194     for measure in row.sensordatavalues:
195         measureValue = {
196             "Name": measure.value_type,
197             "Value": str(measure.value),
198             "Type": "DOUBLE",
199         }
200         measuresValues.append(measureValue)
201
202         if measure.value_type == "P2" and row.aqi is not
203         None:
204             aqi_measureValue = {
205                 "Name": "aqi",
206                 "Value": str(row.aqi),
207                 "Type": "BIGINT",
208             }
209             measuresValues.append(aqi_measureValue)
210
211     # Create a record for each sensor
212     record = {
213         "Dimensions": dimensions,
214         "Time": row_timestamp,
215         "TimeUnit": "MILLISECONDS",
216         "MeasureName": "air_quality",
217         "MeasureValueType": "MULTI",
218         "MeasureValues": measuresValues,
219     }
220     records.append(record)
221
222     # Write records to Timestream if there are 98 records
223     if len(records) >= 98:
224         write_records(
225             database_name, table_name, write_client,
226             records
227         ) # Write records to Timestream
228         records = [] # Reset the records list
229         time.sleep(1) # Sleep for 1 second
230
231     except Exception as e:
232         print(f"Error processing row: {row}")
233         print(f"Exception: {e}")
234
235     # Write records to Timestream if there are any remaining
236     records
237     if len(records) > 100:
238         while len(records) > 100:
239             write_records(
240                 database_name, table_name, write_client, records
241                 [:99]
242             ) # Write records to Timestream
243             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 F

## Scripts & Services

### Appendix F.A Scripts

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

```
1 #!/bin/bash
2
3 # Get the authentication token from the EC2 metadata service
4 TOKEN=$(curl -X PUT "http://169.254.169.254/latest/api/token" -H "X-aws-ec2-metadata-token-ttl-seconds: 21600" -s)
5
6 # Name of the IAM role to assume
7 ROLE_NAME="LabRole"
8
9 # Get temporary credentials using the IAM role
10 IAM_ROLE_CREDENTIALS=$(curl -H "X-aws-ec2-metadata-token: $TOKEN" -s http://169.254.169.254/latest/meta-data/iam/security-credentials/$ROLE_NAME)
11
12 # Extract the credentials and session token
13 AWS_ACCESS_KEY_ID=$(echo $IAM_ROLE_CREDENTIALS | jq -r .AccessKeyId)
14 AWS_SECRET_ACCESS_KEY=$(echo $IAM_ROLE_CREDENTIALS | jq -r .SecretAccessKey)
15 AWS_SESSION_TOKEN=$(echo $IAM_ROLE_CREDENTIALS | jq -r .Token)
16 AWS_DEFAULT_REGION="us-east-1"
17
18 # Export the credentials and session token
19 export AWS_ACCESS_KEY_ID
20 export AWS_SECRET_ACCESS_KEY
21 export AWS_SESSION_TOKEN
22 export AWS_DEFAULT_REGION
```

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

```
1 #!/bin/bash
2
3 # Run the spark job in the background and log output to output.log file
```

```
4 nohup python3 /home/ubuntu/iot_project/ingestion_iot.py >/home/
  ubuntu/iot_project/output.log 2>&1 &
```

## Appendix F.B Services

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

```
1 [Unit]
2 Description=Script to setup AWS cli thanks to the attached IAM
  Profile
3
4 [Service]
5 ExecStart=/usr/local/bin/get_iam_credentials.sh
6
7 [Install]
8 WantedBy=multi-user.target
```

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

```
1 [Unit]
2 Description=Script to run the ingestion python script
3
4 [Service]
5 ExecStart=/usr/local/bin/start_spark_job.sh
6
7 [Install]
8 WantedBy=multi-user.target
```

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

```
1 [Unit]
2 Description=Grafana instance
3 Documentation=http://docs.grafana.org
4 Wants=network-online.target
5 After=network-online.target
6 After=postgresql.service mariadb.service mysqld.service influxdb.
  service
7
8 [Service]
9 EnvironmentFile=/etc/sysconfig/grafana-server
10 User=grafana
11 Group=grafana
12 Type=notify
13 Restart=on-failure
14 WorkingDirectory=/usr/share/grafana
15 RuntimeDirectory=grafana
16 RuntimeDirectoryMode=0750
17 ExecStart=/usr/share/grafana/bin/grafana server
  \
```

```

18                                     --config=${CONF_FILE}
19                                     \
server.pid                           \
20                                     --pidfile=${PID_FILE_DIR}/grafana-
21                                     \
22                                     --packaging=rpm
23                                     \
24                                     \
25                                     \
26                                     \
27                                     \
28                                     \
29                                     \
30                                     \
31                                     \
32                                     \
33                                     \
34                                     \
35                                     \
36                                     \
37                                     \
38                                     \
39                                     \
40                                     \
41                                     \
42                                     \
43                                     \
44                                     \
45                                     \
46                                     \
47                                     \
48                                     \
49                                     \
50                                     \
51                                     \
52                                     \

```

```

25 LimitNOFILE=10000
26 TimeoutStopSec=20
27 CapabilityBoundingSet=
28 DeviceAllow=
29 LockPersonality=true
30 MemoryDenyWriteExecute=false
31 NoNewPrivileges=true
32 PrivateDevices=true
33 PrivateTmp=true
34 ProtectClock=true
35 ProtectControlGroups=true
36 ProtectHome=true
37 ProtectHostname=true
38 ProtectKernelLogs=true
39 ProtectKernelModules=true
40 ProtectKernelTunables=true
41 ProtectProc=invisible
42 ProtectSystem=full
43 RemoveIPC=true
44 RestrictAddressFamilies=AF_INET AF_INET6 AF_UNIX
45 RestrictNamespaces=true
46 RestrictRealtime=true
47 RestrictSUIDSGID=true
48 SystemCallArchitectures=native
49 UMask=0027
50
51 [Install]
52 WantedBy=multi-user.target

```