

Carried out by:

ID : S450915

YASSER EI KARKOURI

Module: Cloud Computing



Collecting, Processing and Distributing IoT Data to Client

Module supervisor teacher:

Dr Stuart Barnes

2 January 2024

Content Table

I.	Introduction	5
II.	Assignment Overview:	5
III.	Goals and Scope of the Project:.....	6
1.	Project Goals.....	6
2.	Project Scope:	6
IV.	Cloud Services and AWS Overview:	7
1.	Cloud Services and AWS Overview:	7
2.	Role of AWS in Cloud Computing:	7
V.	Methodology and system overview :	7
1.	Data Extraction and Storage:.....	7
2.	AWS Cloud Tools Utilized:	10
2.1.	Amazon EC2:	10
2.2.	Amazon DynamoDB:.....	10
2.3.	Amazon SQS:	10
2.4.	AWS CloudWatch:.....	10
2.5.	AWS Cost Management Tools:	10
3.	Development Environment and Tools:	11
3.1.	Core Libraries and Languages:.....	11
3.2.	Web Framework and API Development:	11
3.3.	Template Rendering:	11
3.4.	Development and Testing Tools:	11
VI.	System Architecture and Specification:	12
1.	System Design and Architecture:	12
1.1.	Amazon DynamoDB (QualityAirTable):	12
1.2.	Users and Client Application (ClientApp):	13
1.3.	Backend Processing:.....	14
1.4.	Amazon Simple Queue Service (SQS):	14
1.5.	Automation and Deployment:	15
2.	System Specifications:	15
VII.	Code and Implementation Analysis:	17
1.	Automated System Orchestration and Deployment:.....	17
2.	Air Quality Data Retrieval and Storage Automation	19
3.	Air Quality Dashboard Application:.....	20
3.1.	Back-End:.....	20

3.2.	Front-End:	22
VIII.	Testing and Performance Analysis/Evaluation:	23
1.	Assessing System Performance Under heavy load :	23
2.	System Testing And Results interpretation(no worker nodes) :	24
2.1.	Introduction to the Performance Testing :	24
2.2.	Results Analysis :	25
2.3.	Error Rate Analysis Across Load Conditions :	26
2.4.	Error rate chart interpretation :	27
2.5.	Interpretation of The Results of the charts :	27
2.6.	Conclusion about system potential enhancement :	27
IX.	System Performance Scale-up:	28
1.	Implementing Amazon SQS for Request Management:	28
2.	Enhancing DynamoDB Read/Write Capacities:	29
2.1.	Enhancing DynamoDB Read/Write Capacities:	29
3.	Expected Outcomes:	29
4.	Performance Testing after Updates:	29
4.1.	Results Analysis :	30
4.2.	Error Rate Analysis Across Load Conditions :	30
5.	Overall Comparison and Conclusion:	31
5.1.	Error Count vs. Load:	31
5.2.	Success Count at High Load:	32
5.3.	Response Time Under Load:	32
5.4.	Conclusion:	32
X.	System Pricing Optimization and security:	32
1.	EC2 Instances:	33
2.	DynamoDB:	33
3.	Amazon SQS:	33
4.	CloudWatch:	33
5.	Security :	34
XI.	Conclusion:	34
XII.	References :	36
XIII.	Appendix :	37

Figure 1: Air Quality Data Flow	8
Figure 2:Review of the UK Air Quality Index	9
Figure 3:System Architecture Overview	12
Figure 4:DynamoDB Table (QualityAirTable)	13
Figure 5:User Interface of the Air Quality App.....	13
Figure 6:The execution Message Of the Deployed Backend.....	14
Figure 7:Send and Receive Queues.....	15
Figure 8:System's Sequence Diagram	16
Figure 9:Jupyter Notebook script.....	17
Figure 10:Instances Launching.....	18
Figure 11:Output Of "CC_Initialisation.ipynb" program	19
Figure 12:Execution Log of "master.py"	20
Figure 13:DynamoDb Client Setup	21
Figure 14:Example of CSV File Delivered by Our System to the Client.....	22
Figure 15: "TestPerformance.py" execution log	24
Figure 16:System Performance Analysis Before Deploying Worker Nodes	25
Figure 17:Scatter Plot of the Error count Based on Requests and Concurrency	26
Figure 18:Updated System Performance Analysis	30
Figure 19:Scatter Plot of the Error count Based on Requests and Concurrency (updated).....	31

I. Introduction

The realm of cloud computing has revolutionized the way we think about data processing and storage, offering scalable, efficient, and cost-effective. Real-time data processing, which was once a challenging task for traditional computing systems, is now achievable with cloud computing. The cloud's computational power and storage capabilities provide an ideal platform for handling vast streams of real-time data, enabling businesses to make quicker, data-driven decisions. [1].

Cloud computing has revolutionized the IT industry with its ability to provide scalable, on-demand computing resources. Thanks to this technology, users may access data and apps via the Internet without requiring physical infrastructure or hardware. Its flexibility, cost-effectiveness, and efficiency have made cloud computing a cornerstone of modern IT strategies. In this context, our assignment taps into the core of cloud computing's potential. By focusing on the development of a cloud-based solution, we are not only exploring the technical intricacies of cloud computing but also its application in critical areas like real-time data provision for public good [2].

II. Assignment Overview:

As we witness the transformative impact of cloud computing in various sectors, its relevance becomes increasingly pronounced in the context of our assignment. With the growing concerns about environmental health and safety, timely and accurate information about air quality is more important than ever and this is where the cloud computer interfere as it enables collection and analysis of vast amounts of environmental data from sensors and by using cloud-based technologies to interpret this data in real-time, people, communities, and politicians may be informed about the state of the air quality, any health risks, and appropriate safety measures[3].

In this context, our assignment taps into the core of cloud computing's potential. By focusing on the development of a cloud-based solution, We are investigating not only the technical nuances of cloud computing but also its application in crucial domains such as real-time data provision for public benefit by concentrating on the creation of a cloud-based solution. This project not only enhances our understanding of cloud technologies but also underlines the significant role they play in addressing real-world challenges, such as environmental monitoring [4].

III. Goals and Scope of the Project:

1. Project Goals

The primary goal of this assignment is to develop a cloud-based solution for collecting, processing, and distributing real-time air quality data obtained from a network of IoT environmental sensors. The project focuses on handling data related to airborne particulate pollution, specifically PM10 and PM2.5 levels, and calculating the Air Quality Index (AQI) based on this data. The overarching objectives of this project include [4] :

- **Data Collection and Storage**
- **Air Quality Index Calculation**
- **Data Distribution to Clients**
- **System Elasticity and Cost-Effectiveness**
- **Data Security and Sovereignty**

2. Project Scope:

Our assignment incorporates a complex approach to leverage cloud computing for real-time air quality monitoring .In order to tackle this project we will break it down into more manageable tasks:

- **Utilization of Cloud Services for Data Storage and Processing**
- **Management of Cloud-Based Database**
- **Real-Time Data Processing**
- **Front-End Development and User Interaction**
- **Performance Evaluation and Optimization**
- **Discuss data security and sovereignty**

IV. Cloud Services and AWS Overview:

1. Cloud Services and AWS Overview:

Amazon Web Services (AWS) stands at the forefront of the cloud computing revolution, this platform offers over 200 fully featured services from data centers globally. AWS provides a rich array of tools and services, including but not limited to, computing power, storage options, and database management systems. Our project specifically utilizes key AWS services, which are instrumental in managing the data flow and processing tasks required for real-time air quality monitoring. By leveraging services like **Amazon EC2** for computing power, **Amazon SQS** for managing message queues, **Amazon DynamoDB** for data storage, and **AWS CloudWatch** for monitoring and observability. Each of these services plays a critical role in our system's architecture, offering the reliability, scalability, and performance necessary for efficient real-time data handling [5].

2. Role of AWS in Cloud Computing:

AWS plays a fundamental role in cloud computing by providing a stable platform that meets a broad range of computing requirements from various sectors. For our project, AWS provides an essential framework that enables us to process and analyse environmental data effectively. **Amazon EC2** offers flexible compute capacity, which is crucial for handling our project's computational demands. The dependable transfer of messages between the various parts of our system is made possible by **Amazon SQS**, guaranteeing a seamless and effective data flow. **DynamoDB** offers a fast and flexible NoSQL database service, ideal for storing and retrieving our real-time data. Lastly, **AWS CloudWatch** allows us to monitor our application and respond proactively to any operational issues. We can fully utilize cloud computing by merging these AWS services to produce a system that is not only strong and scalable but also extremely responsive to real-time data processing requirements [6].

V. Methodology and system overview :

1. Data Extraction and Storage:

In our pursuit to develop an efficient system for real-time air quality monitoring, we have employed a methodical approach that seamlessly integrates data retrieval, processing, and storage with automated scheduling . This approach uses cloud computing to its full potential for scalability and best performance, while also guaranteeing data timeliness and accuracy. An automated script has been created to retrieve, calculate, format, and store air quality data in a **DynamoDB** database. Our system, which is enhanced by a well-planned **cron job**, is set up to update this data every day at midnight, ensuring a reliable and current source of air quality data. Here's a deeper methodology overview of the employed methodology (*see figure 1*):

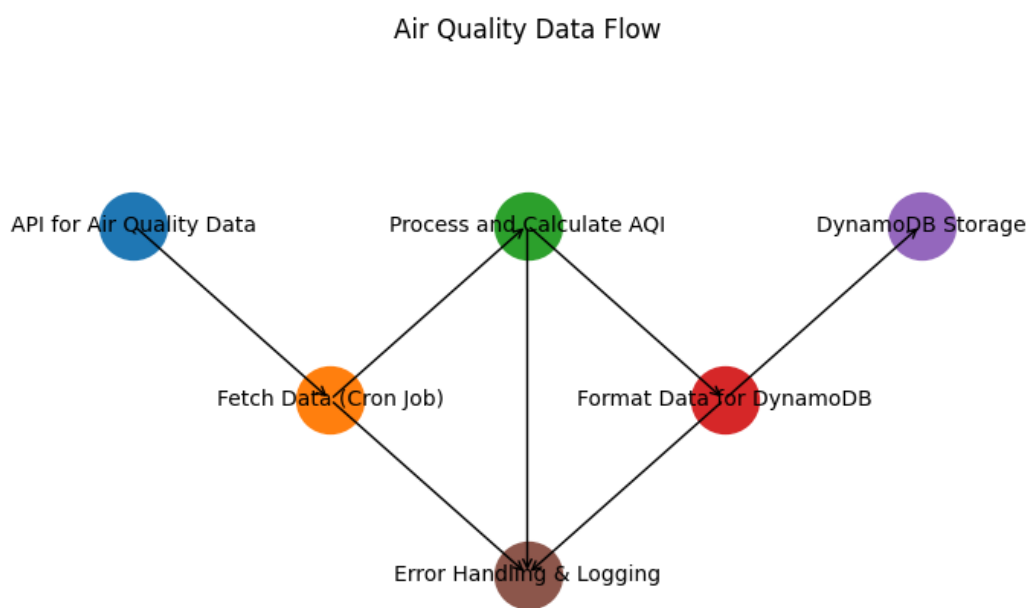


Figure 1: Air Quality Data Flow

➤ **JSON Data Fetching:**

The system is configured to automatically retrieve air quality data from a server API. The data is fetched in JSON format, a widely used standard for data exchange, due to its compatibility and ease of integration. This step ensures access to real-time, structured data from external sources.

➤ **Regular Data Updates:**

A **cron job** is implemented to automate the execution of the data collection and processing tasks. This job runs daily at midnight, ensuring that the data is refreshed and remains relevant.

➤ Data Parsing:

Following retrieval, the system parses the JSON data to extract important data, including sensor locations, timestamps, and pollutant levels. Through this procedure, unstructured data is converted into a format that may be processed further.

➤ Air Quality Index Calculation:

Using the parsed data, the system determines the **Air Quality Index (AQI)**. In order to provide an accurate and consistent estimate of air quality, this entails applying conventional **AQI** computing methods to pollutant data. Air quality can be calculated based on PM_{2.5} and PM₁₀ measurements each range of particulate concentrations corresponds to an AQI value that categorizes the air quality into one of four groups: **Low**, **Medium**, **High**, or **Very High** accordingly to *figure 2* For the rest of the report, we will refer to PM_{2.5} by **P1** and PM₁₀ by **P2**.

Range	Air Quality Index	PM _{2.5} Particles, 24 hour mean (µg/m ³)	PM ₁₀ Particles, 24 hour mean (µg/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

Figure 2:Review of the UK Air Quality Index

➤ Storing Data in DynamoDB:

After the calculation of the AQI we store the data along with the calculated AQI and the range In a **DynamoDB** database (this choice is justified by lot of reasons that you can see in the section ***System Architecture and Design***). DynamoDB's scalability supports the growing volume of data from continuous, automated daily updates.

2. AWS Cloud Tools Utilized:

2.1.Amazon EC2:

Amazon EC2 provides resizable compute capacity in the cloud, which is used to host the backend processes, including the data retrieval and **AQI** calculation services. **EC2** instances can be scaled up or down automatically, providing the flexibility to match the resource requirements of the system [7].

2.2.Amazon DynamoDB:

The choice of **DynamoDB** is informed by its robustness, scalability, ease of management, and integration capabilities, which are all essential for managing the air quality data lifecycle effectively. Its reliability, security features, and cost structure further solidify its suitability for the system, aligning with the project's goal to provide a resilient and efficient air quality monitoring service [8].

2.3.Amazon SQS:

Amazon SQS (Simple Queue Service) is used to decouple and scale microservices, distributed systems, and serverless applications. It helps manage communication between different components of our system, ensuring that data processing and **AQI** calculation tasks are queued and handled efficiently [9].

2.4.AWS CloudWatch:

AWS CloudWatch is vital for monitoring the system's performance, setting alarms, and automatically reacting to changes in the **AWS** resources. It helps us in tracking metrics, collecting log files, and setting alarms to enable real-time monitoring of our components like **EC2** instances and **DynamoDB** database [10].

2.5.AWS Cost Management Tools:

AWS provides tools to track and analyse the costs. These tools will help us in understanding and controlling the costs associated with the **AWS** services used by our system. [11].

3. Development Environment and Tools:

Our air quality data management system was developed using a carefully chosen set of libraries, development tools, and computer languages. The necessity for reliable service design, effective data processing, and smooth testing and deployment served as the basis for these decisions .

3.1.Core Libraries and Languages:

Python was used as a main programming language for the backend of the client web application of our system and it was integrated with **Boto3** ,which is the **AWS SDK** for python, allowing for straightforward coding of data operations and AWS resource management. For tasks that required **SSH** interactions or subprocess management within our scripts, we utilized **Paramiko** and the **Subprocess** module. These facilitated secure system-level operations and process spawning, essential for our backend automation. The **Requests** library was crucial for **HTTP** requests to external **APIs** for air quality data, while **ThreadPoolExecutor** from the **concurrent.futures** module enabled efficient multi-threaded data fetching.

3.2.Web Framework and API Development:

We used **Flask**, a small and adaptable web framework, to create our web server and RESTful API endpoints. It was a great option for the web interface of our system because of its versatility and ease of use. Extensions like **Flask-CORS** were used to handle Cross-Origin Resource Sharing (**CORS**), ensuring our **API** could be securely accessed from web clients hosted on different domains.

3.3.Template Rendering:

JSON was used for data serialization, facilitating the exchange of data between the server and clients. For data downloads, the csv module was used to generate **CSV** files that could be easily consumed by end-users.

3.4.Development and Testing Tools:

Jupyter Notebook offered an interactive computing environment, ideal for exploratory data analysis, prototyping code, and visualizing data. As our Integrated Development Environment (IDE), **VS Code**

provided comprehensive code editing, debugging, and source control integration. The **AWS Command Line Interface** was indispensable for managing AWS services directly from the terminal, allowing for scripting and automation of cloud resources. The choice of **Amazon Linux** for our server environments ensured a seamless and optimized runtime for **AWS services**, benefiting from **AWS** integration and security features. For testing our web server and RESTful APIs, **Postman** was the tool of choice. It allowed for the creation of comprehensive test suites to validate **API** functionality and performance.

VI. System Architecture and Specification:

1. System Design and Architecture:

Our cloud-based environmental data processing system is designed to be robust, scalable, and efficient, leveraging several AWS services to ensure reliable data handling and processing. The following is an overview of the system's architecture and components (*see figure 3*):

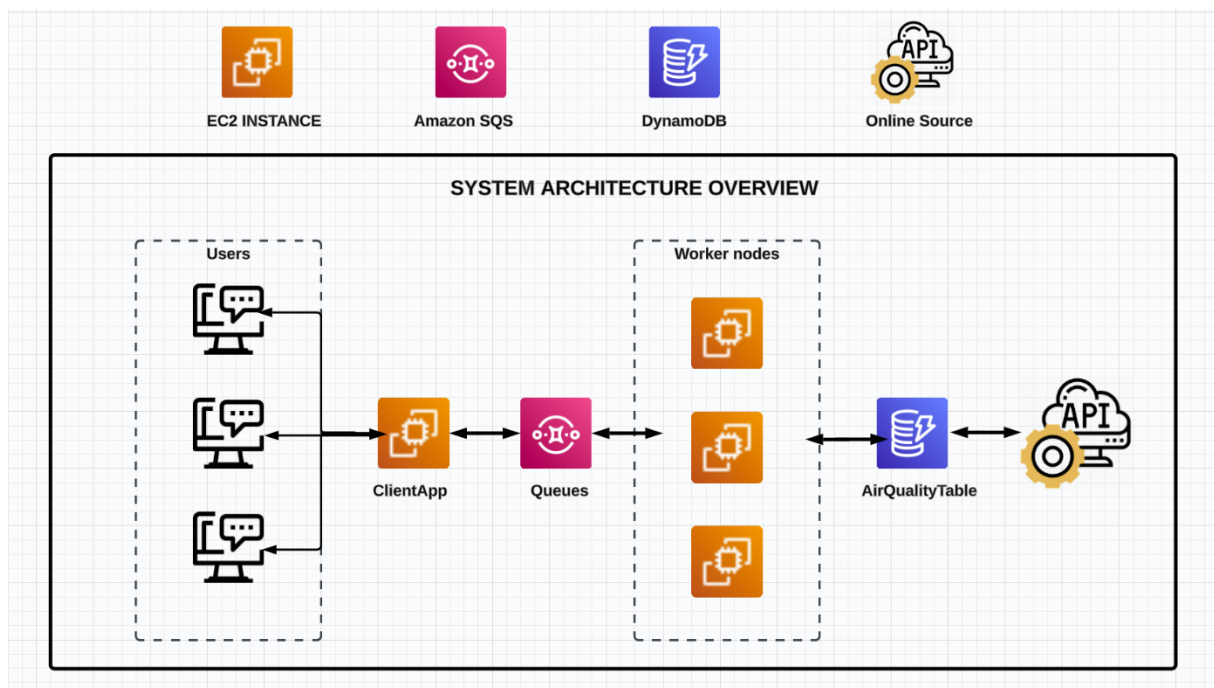


Figure 3: System Architecture Overview

1.1. Amazon DynamoDB (QualityAirTable):

For data storage, we utilize Amazon **DynamoDB**, renowned for its fast and consistent performance, alongside remarkable scalability. Our environmental sensor data is systematically stored in **DynamoDB**

tables, structured for quick and efficient retrieval. This design ensures our system's resilience against increasing workloads, maintaining high performance and quick response times (*see figure 4*).

Éléments retournés (300)

🔄

Actions ▾

Créer un élément

< 1 ... > ⌕ ✖

<input type="checkbox"/>	id (Chaîne) ▾	AQI ▾	loc_alt ▾	loc_country ▾	loc_id ▾	loc_lat ▾	loc_long ▾	P1 ▾	P2 ▾	Range ▾
<input type="checkbox"/>	18612422522	3	106.4	HU	64651	47.54	19.088	27.78	20.89	Low
<input type="checkbox"/>	18612431728	10	452.2	BG	19550	42.872	25.494	112.99	60.41	Very High
<input type="checkbox"/>	18612434761	1	0.5	NL	53918	52.041738...	4.6868571...	9.85	3.31	Low
<input type="checkbox"/>	18612431672	3	609.0	RO	74109	45.636	25.582	33.13	13.12	Low
<input type="checkbox"/>	18612449844	1	234.6	DE	63834	50.905936...	7.4011427...	11.06	4.71	Low
<input type="checkbox"/>	18612445555	1	16.7	NL	28713	51.517260...	5.3916756...	6.07	1.48	Low
<input type="checkbox"/>	18612436583	1	5.4	DE	69727	53.132	8.23	10.07	3.76	Low
<input type="checkbox"/>	18612447180	2	11.8	NL	23002	52.676	6.432	13.77	6	Low
<input type="checkbox"/>	18612444565	2	171.2	DE	7275	50.328	8.72	13.67	9.47	Low

Figure 4:DynamoDB Table (QualityAirTable)

1.2.Users and Client Application (ClientApp):

The system's user interaction is facilitated through **ClientApp**, a web-based frontend designed that allows them to query environmental data. Given that it is designed according to responsive design principles, users will have a smooth and interesting experience using it on a variety of devices. It allows users to easily input queries based on diverse parameters such as date ranges, countries, and Air Quality Index (AQI) levels. **AJAX** requests are used to interface this user-friendly and intuitive frontend with the backend, enabling interactive and real-time data retrieval and querying (*see figure 5*).

Data Query Interface

Start Date:

End Date:

Country:

Minimum AQI:

Maximum AQI:

Range:

Select Range ▾

Submit Query

Figure 5:User Interface of the Air Quality App

1.3.Backend Processing:

The core of our system is a **Python-Flask** backend (**App.py**), hosted on an **AWS EC2 instance** (**ClientApp**) for constant operational availability. This backend is intricately designed to manage complex request processing, including handling time parameters and a wide spectrum of **AQI** categories. Robust error handling and request validation are implemented to ensure data integrity and system stability. Our backend is further bolstered by performance testing scripts that rigorously evaluate its responsiveness and capacity to handle concurrent user requests efficiently, thereby ensuring a reliable user experience even under high load conditions (*see figure 6*).

```
[ec2-user@ip-172-31-28-201 ~]$ cat /home/ec2-user/app.log
* Serving Flask app 'App' (lazy loading)
* Environment: production
  WARNING: This is a development server. Do not use it in a production deployment.
  Use a production WSGI server instead.
* Debug mode: off
* Running on all addresses.
  WARNING: This is a development server. Do not use it in a production deployment.
* Running on http://172.31.28.201:8501/ (Press CTRL+C to quit)
```

Figure 6: The execution Message Of the Deployed Backend

1.4.Amazon Simple Queue Service (SQS):

To further enhance scalability and adeptly manage high-volume requests, we integrate **Amazon SQS** with two specific queues: "**RequestSend**" for incoming user queries and "**RequestReceive**" for delivering processed data. This queuing mechanism effectively separates the processing workload from the frontend, optimizing throughput and load distribution. It plays a critical role in maintaining system responsiveness and smooth user experience during peak traffic periods. (*see figure 7*).

Actions ▾

Créer une file d'attente

Q

Rechercher les files d'attente par préfixe

<1>

⚙

	Nom ▲	Type ▾	Date de création ▾	Messages dis
<input type="radio"/>	RequestReceive	Standard	2023-12-29T19:08+00:00	0
<input type="radio"/>	RequestSend	Standard	2023-12-29T19:06+00:00	0

Figure 7:Send and Receive Queues

1.5.Automation and Deployment:

Deployment on AWS is streamlined through automated **Python** scripts that skilfully manage **EC2 instance initialization**, security protocol configuration, **Flask application** deployment, and the exposure of the public URL for user access. This automation not only simplifies deployment processes but also ensures efficient and manageable long-term system maintenance.

2. System Specifications:

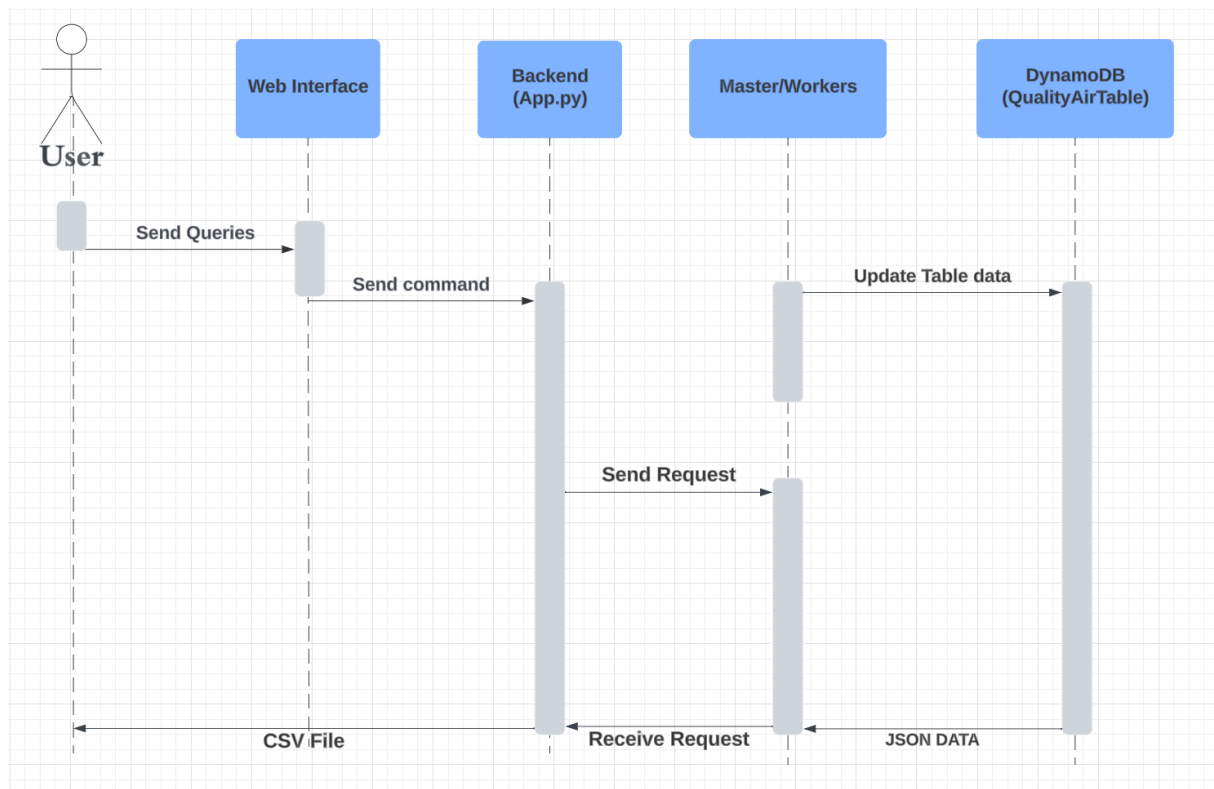


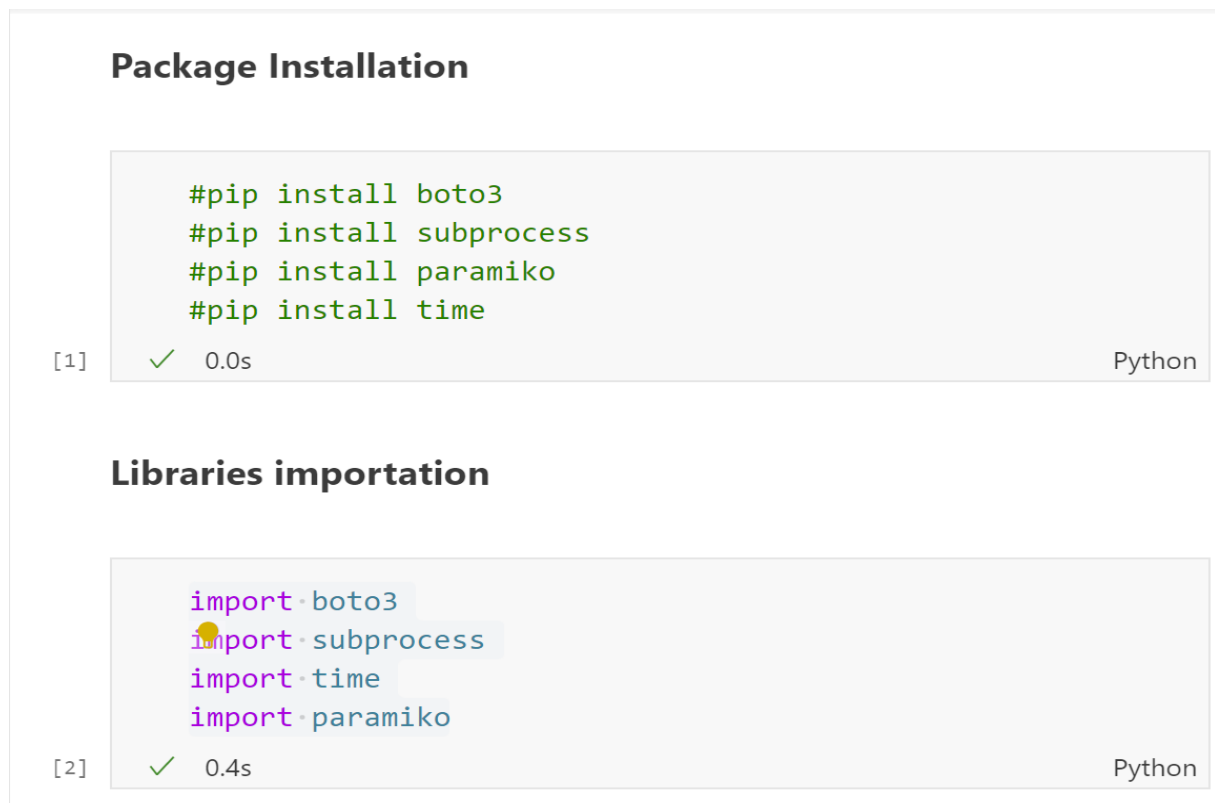
Figure 8: System's Sequence Diagram

This sequence diagram encapsulates the workflow of our cloud-based environmental data processing system. As a user submits a query through the web interface, the system initiates a series of orchestrated processes to handle the request. The query is transmitted to our **Python-Flask** backend, running on **App.py**, which acts as the command centre for the incoming user commands. This backend is designed to parse and validate the query parameters, ensuring the integrity of the request before it's further processed. Once the backend approves the query, it dispatches the request to our **Master/Workers** system. This setup is pivotal in our architecture, as it leverages the power of distributed processing to manage and execute data operations at scale. **The Master** orchestrates the workflow, while the **Workers** are responsible for fetching and processing the environmental data, ensuring that even complex queries are handled swiftly and efficiently. The processed data is then stored in our **DynamoDB** table, **QualityAirTable**, chosen for its high performance and scalability features. **DynamoDB** allows us to handle large datasets with ease, providing fast and consistent access to the processed data. When the data is updated, it's formatted as **JSON**, which is indicative of the structured and organized manner in which we handle data storage. The final step in the workflow is the preparation of the user's results. The **backend** receives the processed data from the **Master/Workers** in **JSON format** and then converts it into a **CSV file**. This file is then made available to the user through the web interface for **download**. By providing the data in **CSV format**, we ensure that users can easily interact with and analyse the environmental data we've processed. This **end-to-end** flow from user query to data delivery underscores our system's commitment to delivering a responsive, reliable, and user-friendly experience.

VII. Code and Implementation Analysis:

1. Automated System Orchestration and Deployment:

The first program we are going to introduce is the “CC_Initialisation.ipynb” , this python script is developed in a **Jupyter Notebook** and this choice is justified by the fact that you can comment each section of the code (*see Figure 9*) .



```
[1] ✓ 0.0s Python
Package Installation

#pip install boto3
#pip install subprocess
#pip install paramiko
#pip install time

[2] ✓ 0.4s Python
Libraries importation

import boto3
import subprocess
import time
import paramiko
```

Figure 9:Jupyter Notebook script

The script is designed to automate the setup and deployment of our distributed computing environment designed for the air quality data management system. Here's a breakdown of the main key functionalities:

➤ **Function definition :**

create_DB (DynamoDB) : This function is responsible for creating a DynamoDB table named 'QualityAirTable'. It defines the table schema, including key attributes and throughput settings. The table serves as the primary storage for air quality data, crucial for the system's data management.

create_sqs_queue(SQS, queue_name) : This function automates the creation of **SQS queues**. Each queue facilitates asynchronous message passing within the distributed system, essential for handling data processing requests and responses.

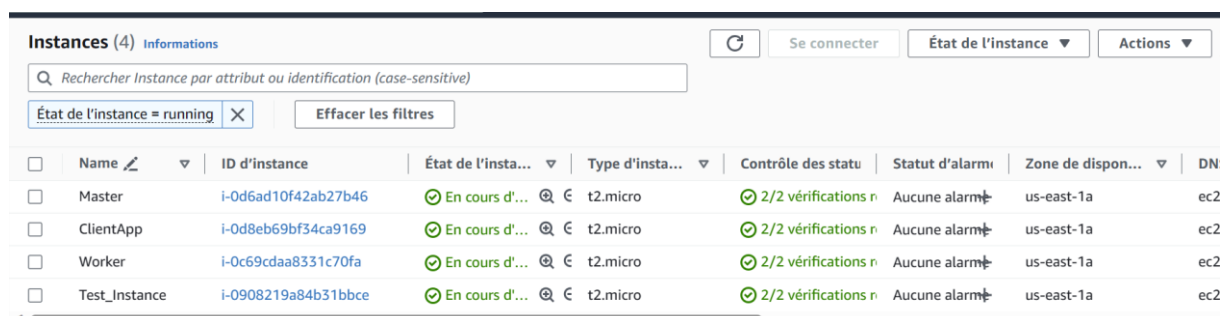
create_instance() : It handles the launch and initial setup of **EC2 instances**. The function specifies instance attributes like **AMI ID**, **type**, **security groups**, and **initialization scripts**. It's pivotal in establishing the computing environment for various system components, including data processing and web hosting.

upload_script() : This function manages the uploading of local scripts to the specified **EC2 instance**. It's crucial for deploying the necessary code and configurations onto the instances, ensuring that they are equipped to perform their designated tasks.

execute_script() : It executes scripts on an **EC2 instance**. The function is designed to remotely run scripts as background processes, which is key for starting applications or services on the instances.

setup_ClientApp() : This function is especially designed for the deployment of the **client-facing** web application on an **EC2 instance**. It involves transferring the application package, configuring the environment, and launching the application, making the system accessible to users.

Main() : The main function encapsulates the entire logic for setting up the air quality data management system's infrastructure. It initiates the “**ClientApp**”, “**Master**”, “**Worker**”, “**Test_Instance**” Instances and it ensures that each of them has been configured with the correct resources set-ups it also launch two SQS queues ; “**RequestSend**” and “**RequestReceive**”. The main function is also responsible for the creation and of the **DynamoDB** database that we will be using next for the storage of the retrieved data from the air quality server (*see Figure 10*).



	Name	ID d'instance	État de l'instance	Type d'instance	Contrôle des statuts	Statut d'alarme	Zone de disponibilité	DN
<input type="checkbox"/>	Master	i-0d6ad10f42ab27b46	En cours d'...	t2.micro	2/2 vérifications n	Aucune alarme	us-east-1a	ec2
<input type="checkbox"/>	ClientApp	i-0d8eb69bf34ca9169	En cours d'...	t2.micro	2/2 vérifications n	Aucune alarme	us-east-1a	ec2
<input type="checkbox"/>	Worker	i-0c69cdaa8331c70fa	En cours d'...	t2.micro	2/2 vérifications n	Aucune alarme	us-east-1a	ec2
<input type="checkbox"/>	Test_Instance	i-0908219a84b31bbce	En cours d'...	t2.micro	2/2 vérifications n	Aucune alarme	us-east-1a	ec2

Figure 10:Instances Launching

➤ Error Handling and Sequential Execution:

Throughout all the functions error handling is implemented to ensure smooth execution. The program was formulated using a class which is a more mature and organized way to code ; It not only enhances the quality and maintainability of the code but also facilitates scalability and collaboration. Once executed the output carefully display the sequences of setup steps to ensure that each component is initialized and ready before moving to the next and to keep the administrator in track of the process of the deployment , the system also print out the link to the web application interface (*see Figure 11*).

```

Queue 'RequestSend' created successfully. URL: https://sqs.us-east-1.amazonaws.com/513771522821/RequestSend
Queue 'RequestReceive' created successfully. URL: https://sqs.us-east-1.amazonaws.com/513771522821/RequestReceive
Instance i-0d6ad10f42ab27b46 (Master) created, waiting for it to run...
Instance i-0d6ad10f42ab27b46 (Master) is running.
Script uploaded and executed on the 'Master' instance.
Instance i-0d8eb69bf34ca9169 (ClientApp) created, waiting for it to run...
Instance i-0d8eb69bf34ca9169 (ClientApp) is running.
b'Last metadata expiration check: 0:00:46 ago on Mon Jan 1 07:58:58 2024.\nDependencies resolved.\nNothing to do.\nComplete!\n'
b'Last metadata expiration check: 0:00:47 ago on Mon Jan 1 07:58:58 2024.\nPackage python3-3.9.16-1.amzn2023.0.6.x86_64 is already ins
b'Archive: /home/ec2-user/ClientApp.zip\n creating: /home/ec2-user/app/ClientApp/\n inflating: /home/ec2-user/app/ClientApp/App.py
b'Collecting Flask==2.0.1\n Downloading Flask-2.0.1-py3-none-any.whl (94 kB)\nCollecting Werkzeug==2.0.0\n Downloading Werkzeug-2.0.0
Application deployed. Access it at http://54.159.31.234:8501
'ClientApp' instance setup completed successfully.
Instance i-0c69cdaa8331c70fa (Worker) created, waiting for it to run...
Instance i-0c69cdaa8331c70fa (Worker) is running.
Script uploaded and executed on the 'Worker' instance .
Instance i-0908219a84b31bbce (Test_Instance) created, waiting for it to run...
Instance i-0908219a84b31bbce (Test_Instance) is running.
Script uploaded and executed on the 'Test_Instance' instance.

```

Figure 11:Output Of "CC_Initialisation.ipynb" program

2. Air Quality Data Retrieval and Storage Automation

Master.py is a Python script is an integral component of the air quality data management system. It is primarily focused on extracting sensor data from an external online server and seamlessly storing it in a **DynamoDB** table. The instance where this python program is deployed was programmed to execute it each day at midnight to update it with the new data .

➤ Key Functionalities:

AQI Calculation: The **calculate_aqi** function computes the Air Quality Index (AQI) based on **PM2.5** and **PM10** sensor readings. It uses defined thresholds to categorize the air quality into distinct ranges such as Low, Medium, High, and Very High.

Data Type Conversion: The **float_to_decimal** function converts floating-point numbers to a Decimal format. This conversion is necessary for compatibility with DynamoDB, which requires numeric data to be stored as Decimal types.

Data Storage: The **store_data** function encapsulates the logic for data insertion into DynamoDB. It processes each data entry, ensures AQI calculations are performed, and formats the data into the schema

expected by the 'QualityAirTable' before storage , it also perform the cleansing and the structuring of fetched data.

Sensor Data Fetching: The `fetch_sensor_data` function is responsible for making **HTTP** requests to the specified **API endpoint** to retrieve the latest air quality data in a **JSON** form.

➤ Error Handling and Logging:

The program is laced with error handling that captures and logs any exceptions that may occur during the data fetching and storage processes, ensuring the system's robustness and maintainability. Once executed the output carefully display the sequences of data storage keep the administrator in track of the process (see *Figure 12*).

```
2024-01-01 07:57:38,434 - INFO - Stored entry 23511 with ID 18612449073 in DynamoDB.
2024-01-01 07:57:38,434 - INFO - Skipping entry 23512 as it doesn't contain P1 or P2 values.
2024-01-01 07:57:38,440 - INFO - Stored entry 23513 with ID 18601214849 in DynamoDB.
2024-01-01 07:57:38,440 - INFO - Skipping entry 23514 as it doesn't contain P1 or P2 values.
2024-01-01 07:57:38,440 - INFO - Skipping entry 23515 as it doesn't contain P1 or P2 values.
2024-01-01 07:57:38,445 - INFO - Stored entry 23516 with ID 18612437590 in DynamoDB.
2024-01-01 07:57:38,451 - INFO - Stored entry 23517 with ID 18612433757 in DynamoDB.
2024-01-01 07:57:38,451 - INFO - Skipping entry 23518 as it doesn't contain P1 or P2 values.
2024-01-01 07:57:38,456 - INFO - Stored entry 23519 with ID 18612444290 in DynamoDB.
2024-01-01 07:57:38,461 - INFO - Stored entry 23520 with ID 18612429742 in DynamoDB.
2024-01-01 07:57:38,461 - INFO - Skipping entry 23521 as it doesn't contain P1 or P2 values.
2024-01-01 07:57:38,467 - INFO - Stored entry 23522 with ID 18612439586 in DynamoDB.
2024-01-01 07:57:38,472 - INFO - Stored entry 23523 with ID 18612439874 in DynamoDB.
2024-01-01 07:57:38,477 - INFO - Stored entry 23524 with ID 18612446733 in DynamoDB.
2024-01-01 07:57:38,483 - INFO - Stored entry 23525 with ID 18612448140 in DynamoDB.
2024-01-01 07:57:38,483 - INFO - Data fetch and store completed.
```

Figure 12:Execution Log of "master.py"

3. Air Quality Dashboard Application:

The "**ClientApp**" application is a user-centric component of the air quality data management system, intended to run on an **AWS EC2 instance**, referred to as the **ClientApp** instance. This application is comprised of several interconnected files that work together to provide a responsive and informative user interface.

3.1.Back-End:

App.py is the main Python script that utilizes Flask to serve the web application. It initializes the server and sets up routes to handle requests, serving as the entry point for user interactions. The script is

structured to handle web requests, interact with a **DynamoDB** database, and return data to users in a convenient format.

➤ **Key Functionalities:**

Web Server Initialization: The script initializes a Flask application and enables CORS to allow cross-origin requests, ensuring the web application can be accessed from various client-side locations.

Decimal Type Handling: A custom **DecimalEncoder** class extends the **json.JSONEncoder** to serialize Decimal objects returned by **DynamoDB**. This ensures compatibility with **JSON** formats since the standard **JSON** encoder cannot handle Decimal types.

DynamoDB Client Setup: It establishes a connection to **AWS DynamoDB**, which is configured with necessary access credentials and regional settings (*see Figure 13*).

```
# Initialize the DynamoDB client
dynamodb = boto3.resource(
    'dynamodb',
    aws_access_key_id='ASTAXPHZLJMSCKB26JVU',
    aws_secret_access_key='RIE3mYDZWVfG2ohPxA/2FYXfa3suIUe07ztZLb9B',
    aws_session_token='FwoGZXIvYXdzEKH////////wEaDGNfp2RBGuLJKCEaUSLCARa+W/jOE3SWr8Aegdd',
    region_name='us-east-1'
)
```

Figure 13:DynamoDb Client Setup

Data Query Handling : The “/api/query” route is defined to process POST requests processes them immediately if the system is under a certain load threshold, or queues them for later processing by other worker nodes if the system is busy. It ensures that user requests are handled efficiently and in an orderly manner.

Data Query and CSV Generation: The script directly handles queries thanks to the function that we have defined for this purpose **direct_query_processing** , the App.py program quickly fetching and converting data to CSV, which is then immediately available for download when it is under **Low load** . However when the system is under **heavy load** instead of processing the query immediately the script assigns a unique identifier to the query and sends a message containing the query data to the SQS request queue via **sqs.send_message**. Separately, workers are triggered by messages in the SQS request queue .These workers perform the query against **DynamoDB**, generate a CSV, and then send the results to the SQS response queue with the corresponding unique identifier, the **app.py** program proceed to data preparation and send it directly to the client in a private channel **/api/results/<id>** with their unique Identifier. The following figure is an example of the csv file a user receives for a given set of filters .

loc_alt	P1	P2	loc_lat	loc_country	loc_id	timestamp	loc_long	AQI	Range	id
116.5	0.2	0.2	51.332	DE	34307	31/12/2023 15:07	12.334		1 Low	1,8603E+10
15.7	8.13	3.17	53.868	DE	57789	31/12/2023 12:19	10.728		1 Low	1,8601E+10
158.1	6.64	2.42	51.53376480	DE	39106	31/12/2023 23:00	9.913076499		1 Low	1,8607E+10
71.5	3.96	2.05	51.104	DE	47749	31/12/2023 10:03	7.008		1 Low	1,86E+10
77.4	8.59	4.35	53.02118900	DE	3825	31/12/2023 15:22	13.70628200		1 Low	1,8603E+10
30.9	1.13	0.78	51.496	DE	5844	31/12/2023 23:32	6.788		1 Low	1,8608E+10
106.0	6.18	3.02	50.874	DE	49208	31/12/2023 08:10	6.466		1 Low	1,8599E+10
371.8	13.45	8.05	48.752	DE	58089	31/12/2023 13:49	11.338		2 Low	1,8602E+10
33.9	21.6	12.58	52.38900451	DE	69929	31/12/2023 19:57	13.07839214		2 Low	1,8606E+10
454.5	5.56	2.5	48.796	DE	22242	31/12/2023 23:15	9.93		1 Low	1,8608E+10
297.2	2.87	1.56	49.97414235	DE	19958	31/12/2023 15:46	11.02703534		1 Low	1,8603E+10
211.3	13.22	6.8	50.126	DE	16389	31/12/2023 23:58	8.45		2 Low	1,8608E+10
157.4	0.9	0.7	50.074	DE	277	31/12/2023 15:33	8.436		1 Low	1,8603E+10
12.8	7.45	3.82	53.958	DE	25499	31/12/2023 23:50	10.97		1 Low	1,8608E+10
343.4	4.05	0.35	50.562	DE	40818	31/12/2023 16:42	7.454		1 Low	1,8604E+10
37.5	26.38	24.07	52.44	DE	7461	31/12/2023 21:36	13.54		3 Low	1,8607E+10
54.0	1.42	0.9	50.88344459	DE	15252	31/12/2023 14:20	7.090196907		1 Low	1,8602E+10
197.5	4.93	1.76	49.23662370	DE	28856	31/12/2023 15:13	7.000899900		1 Low	1,8603E+10
302.1	4.06	1.73	50.948	DE	17348	31/12/2023 09:03	10.696		1 Low	1,8599E+10
91.6	1.21	0.56	49.734	DE	31212	31/12/2023 15:05	8.49		1 Low	1,8603E+10
63.8	9.55	2.82	51.574	DE	944	31/12/2023 20:07	6.98		1 Low	1,8606E+10
124.1	2.36	1.06	48.88753010	DE	59499	31/12/2023 10:04	8.320859050		1 Low	1,86E+10
256.0	0.96	0.38	48.78	DE	849	31/12/2023 18:53	9.216		1 Low	1,8605E+10
2.5	5.19	3.39	53.478	DE	25426	31/12/2023 14:18	10.19		1 Low	1,8602E+10
52.5	0.62	0.44	50.942	DE	3363	31/12/2023 20:09	7.034		1 Low	1,8606E+10
484.3	1.39	1.06	48.636	DE	22931	31/12/2023 18:39		9	1 Low	1,8605E+10
853.8	2.85	1.17	47.9	DE	22603	31/12/2023 11:38	8.146		1 Low	1,8601E+10
28.2	5.18	2.56	52.634	DE	6667	31/12/2023 23:53	9.226		1 Low	1,8608E+10

Figure 14:Example of CSV File Delivered by Our System to the Client

3.2.Front-End:

The front end for the "ClientApp" features a Data Query **Interface**, specifically crafted to facilitate user engagement with the air quality data management system. **The interface** includes input fields for **start** and **end dates**, **allowing** users to define the specific **timeframe** for which they seek data. Additional filters such as **country**, **minimum AQI**, and **maximum AQI**, give users the ability to refine their search according to geographic and quality parameters. A drop-down menu for selecting the **AQI range** further enhances the customization of the query, ensuring that users can quickly and easily access the data most relevant to their needs. Once the desired parameters are set, the submission of the query is just a click away with the "Submit Query" button.

Procfile : Used in certain deployment platforms, this file specifies the command that should be executed to start the web application. It's critical for the proper launching of the application in environments that support Procfiles.

requirements.txt : Lists all the Python dependencies necessary for the application to run. This file is used by package managers like pip to install all required libraries in one go, ensuring consistency across development and production environments.

Request.css : Contains the Cascading Style Sheets (CSS) for the web application. It defines the styling and layout of the web pages, ensuring the application is visually appealing and user-friendly.

Request.js : This JavaScript file contains client-side code for the web application. It is responsible for interactivity on the web pages, handling user events, and possibly fetching and displaying data dynamically.

Request.html : The HTML template for the application's interface. It structures the content presented to the user and integrates with the CSS and JavaScript files to render the web pages.

VIII. Testing and Performance Analysis/Evaluation:

1. Assessing System Performance Under heavy load :

We have developed a python program intended to evaluate how the air quality data management system withstands and responds to a heavy workload. This script will help us simulating concurrent user requests, thereby stress-testing the system's responsiveness and robustness. The program's methodology for gauging system resistance to workload involves the following steps :

➤ Concurrent Request Simulation:

Using **ThreadPoolExecutor**, the program simulates multiple users making concurrent **POST** requests to the **Flask** application's **/api/query** endpoint. This mimics a real-world scenario where numerous users are querying the system simultaneously, thereby creating a substantial workload.

➤ Response Time Measurement:

Each simulated request records the time immediately before the request is sent and immediately after the response is received. The difference between these two times is the response time, which is a critical measure of performance under load.

➤ Response Time Measurement:

Each simulated request records the time immediately before the request is sent and immediately after the response is received. The difference between these two times is the response time, which is a critical measure of performance under load.

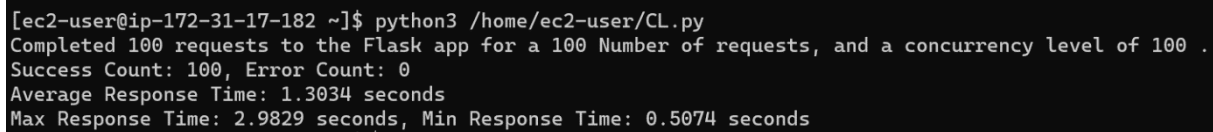
➤ Data Aggregation and Analysis:

Upon completion of the requests, the program aggregates the data to calculate key performance metrics, such as **average response time**, **maximum response time**, and **error rate**. These metrics are essential for identifying performance **bottlenecks** and the system's threshold for maintaining optimal performance.

2. System Testing And Results interpretation(no worker nodes) :

To ensure the air quality data management system delivers reliable and efficient service under various operational conditions, we conducted a series of performance tests. These tests were designed to measure the system's response times under different levels of user demand, providing a comprehensive assessment of its ability to handle both typical and peak loads.

To test our system without the **worker nodes** we will increase the number of simultaneous requests to simulate varying degrees of **user load**, from a baseline of **10 requests** with **10 concurrent** users to a peak of **1000 requests** with **500 concurrent** users. The response time metrics provided clear insights into how the system scales and what users may experience during different traffic volumes. The next (*figure 15*) is a representation of the execution logs of our “**TestPerformance.py**” program



```
[ec2-user@ip-172-31-17-182 ~]$ python3 /home/ec2-user/CL.py
Completed 100 requests to the Flask app for a 100 Number of requests, and a concurrency level of 100 .
Success Count: 100, Error Count: 0
Average Response Time: 1.3034 seconds
Max Response Time: 2.9829 seconds, Min Response Time: 0.5074 seconds
```

Figure 15: "TestPerformance.py" execution log

2.1.Introduction to the Performance Testing :

The performance tests were structured to mimic real-world usage scenarios ranging from **low** to **very high concurrency levels**. By submitting a series of controlled **HTTP POST** requests to the **Flask application's** endpoint, we were able to measure the system's response and record the **highest, average,** and **minimum response times**. This approach allows us to not only understand the system's behaviour under optimal conditions but also to anticipate how it might perform during unexpected surges in traffic (*see figure 16*).

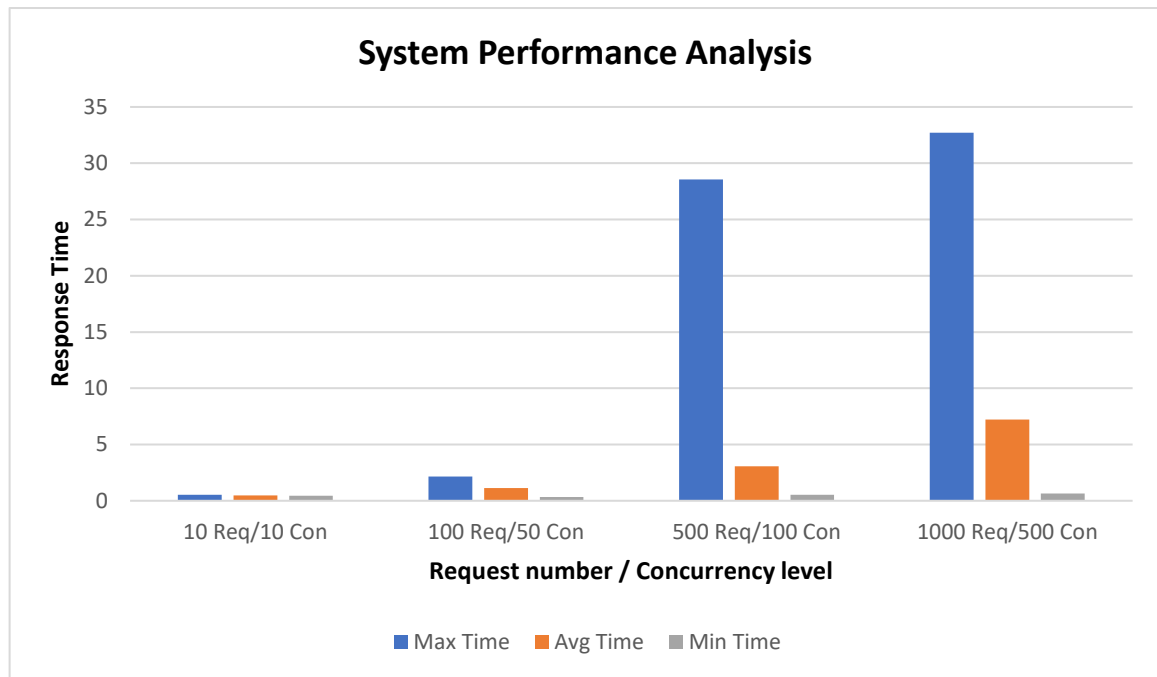


Figure 16: System Performance Analysis Before Deploying Worker Nodes

2.2. Results Analysis :

The performance tests were structured to mimic real-world usage scenarios ranging from low to very high concurrency levels. By submitting a series of controlled **HTTP POST** requests to the Flask application's endpoint, we were able to measure the system's response and record the **highest**, **average**, and **minimum** response times. This approach allows us to not only understand the system's behaviour under optimal conditions but also to anticipate how it might perform during unexpected surges in traffic:

- Under low load (**10 Requests/10 Concurrency**), the system proved efficient with uniform response times.
- Moderate load (**100 Requests/50 Concurrency**) showed a slight increase in max response time, yet the system performed well on average.
- High load (**500 Requests/100 Concurrency**) testing saw a significant rise in response times, signalling potential delays for users.
- At very high load (**500 Requests/100 Concurrency**) the max response time improved, suggesting effective load management, but the increased average response time indicated overall slower performance.

2.3. Error Rate Analysis Across Load Conditions :

For further understanding of the system's behaviour under various load conditions we will proceed to a **3D scatter plot** which will provide us with more reliability during the performance testing. By mapping out the interaction between the number of simultaneous users accessing the system (**Concurrency Level**), the total number of requests submitted (**Number of Requests**) and the resultant error occurrences (**Error Count**). Through this analysis, we aim to understand at what point the system's capacity is reached, informing strategies for enhancement and optimization to better serve user requests efficiently and reliably (*see figure 16*).

Error Count based on Concurrency Level and Number of Requests

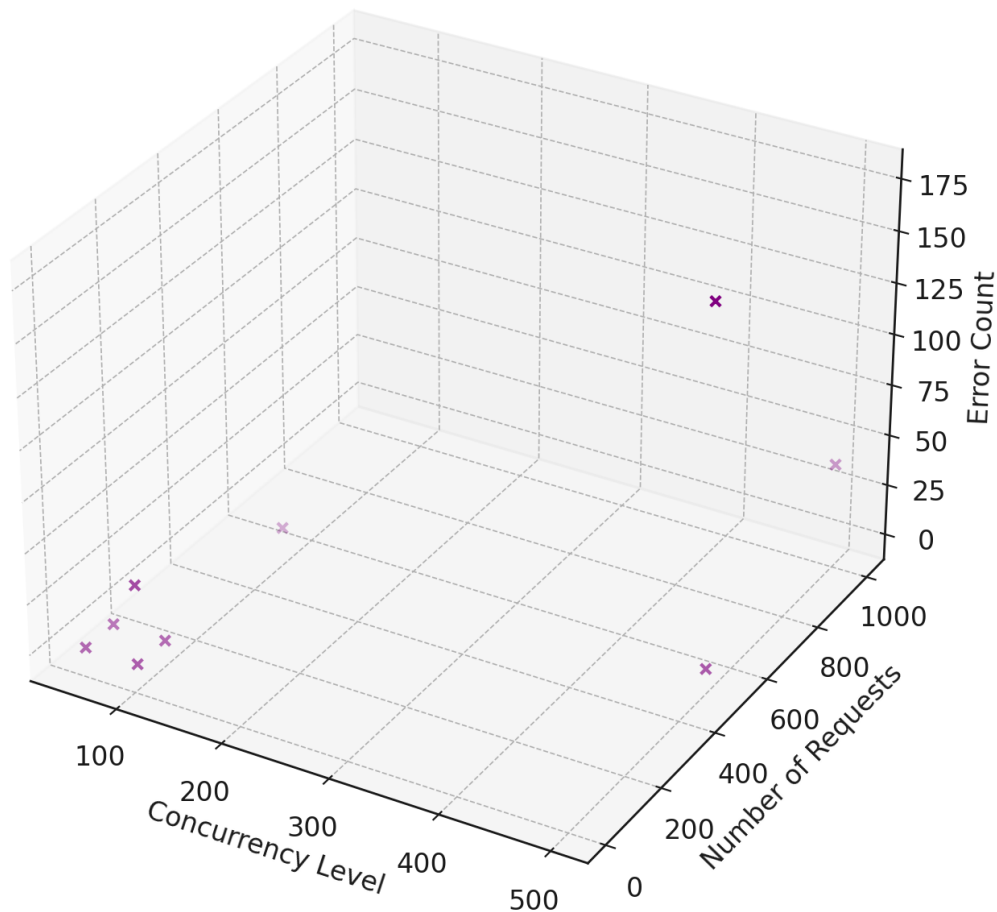


Figure 17: Scatter Plot of the Error count Based on Requests and Concurrency

2.4. Error rate chart interpretation :

By analysing the result displayed on the scatter plot chart that showcases the error count depending on the Concurrency level and the Number of requests there are several important information about our system that we can retrieve :

- The app seems to handle carrying levels of Requests , however error count seems to vary increasingly with the concurrency increasing .

2.5. Interpretation of The Results of the charts :

Concurrency Level and Number of Requests: The system scales relatively well up to moderate load but begins to show signs of strain under high concurrency levels. This is a typical behaviour where systems begin to hit resource limits. This could be due to insufficient resources like CPU, memory, or network limitations which means that the system needs scaling resources by adding **Worker nodes** or increasing the reading and writing capacity of our **DynamoDB** database .

Execution Time and request handling: The decrease in max response time at the highest load level indicates effective queuing and load management strategies that prevent the system from becoming overwhelmed, but these mechanisms may slow down the overall processing time.

Error Count : the error count reflects the number of failed requests and it increases with the number of simultaneous requests which confirm what we have concluded from the first chart analysis.

2.6. Conclusion about system potential enhancement :

From all what has been analysed about the performance of our system under different load conditions. Here are some potential improvements in our system:

Scalability and Load Management : To scale up performance, we will consider enhancing resource allocation, such as the DynamoDB memory capacity, or integrating additional worker nodes since it is just our master program that was executing queries . This should provide a more robust infrastructure to handle high traffic volumes effectively.

Load Balancing and Error Management : The fluctuating error rates with increased concurrency suggest a need for improved load balancing and error handling mechanisms. Implementing more sophisticated load balancing techniques could distribute the traffic more evenly, reducing the strain on the system during peak loads. Additionally, enhancing error detection and management protocols could minimize failed request rates, ensuring a more reliable service.

Database Optimization: Given the pivotal role of our DynamoDB database in handling requests, optimizing its read and write capacities could significantly improve performance. This could involve implementing auto-scaling for the database or optimizing query performance.

IX. System Performance Scale-up:

This section details the implementation of two critical solutions aimed at scaling up the performance of the air quality data management system under heavy load. The solutions focus on enhancing request handling efficiency and optimizing database interaction.

1. Implementing Amazon SQS for Request Management:

To manage increased traffic efficiently, we will integrate Amazon Simple Queue Service (SQS) into our system. This integration involves two queues: "**RequestSend**" for incoming requests and "**RequestReceive**" for processed data.

➤ Request Handling and automate scalability Under Heavy Load:

When active requests exceed **100**, the system will automatically redirect additional requests to the "**RequestSend**" SQS queue. This ensures that incoming traffic is managed effectively without overloading the system.

➤ Master Node Processing (App.py):

The master node continuously scans the "**RequestSend**" queue for new messages and upon detecting a message, it retrieves the query data from the **DynamoDB**. The node then processes the request and sends the results back through the "**RequestReceive**" queue.

➤ Integration with Flask Application (App.py):

The Flask application is configured to monitor the "**RequestReceive**" queue constantly. When it receives processed data from **worker nodes**, it compiles the information into a **CSV file**. The **CSV** file is then sent to the client, ensuring a seamless data delivery experience.

2. Enhancing DynamoDB Read/Write Capacities:

2.1. Enhancing DynamoDB Read/Write Capacities:

To further boost system performance, we will increase the read and write capacities of our **DynamoDB** from **150** to **1000 units**. This expansion is designed to accommodate higher data throughput, especially during peak usage times.

3. Expected Outcomes:

Scalability: This new approach allows the system to handle a significant increase in user requests without compromising performance. It also upgrade the database to handle larger columns of data seamlessly, a key aspect of scalability

Efficiency: **Queue-based** processing ensures that each request is handled in an organized manner, reducing the chances of system crashes or delays. Also higher capacities should decrease the time taken for database operations, contributing to overall system efficiency.

Flexibility: The system can dynamically adjust to varying loads, providing a robust solution for peak traffic scenarios.

4. Performance Testing after Updates:

We are going to follow the same strategy of testing that we have proceed with to evaluate the performance of our scaled up system. We will execute the same program, “**TestPerformance.py**” to confirm our expected outcomes.

To ensure that our results are valid we will use same test values and we will use same query , because the size of the requested data may affect the procedure . The following chart showcases the obtained performance results (*see figure 18*)

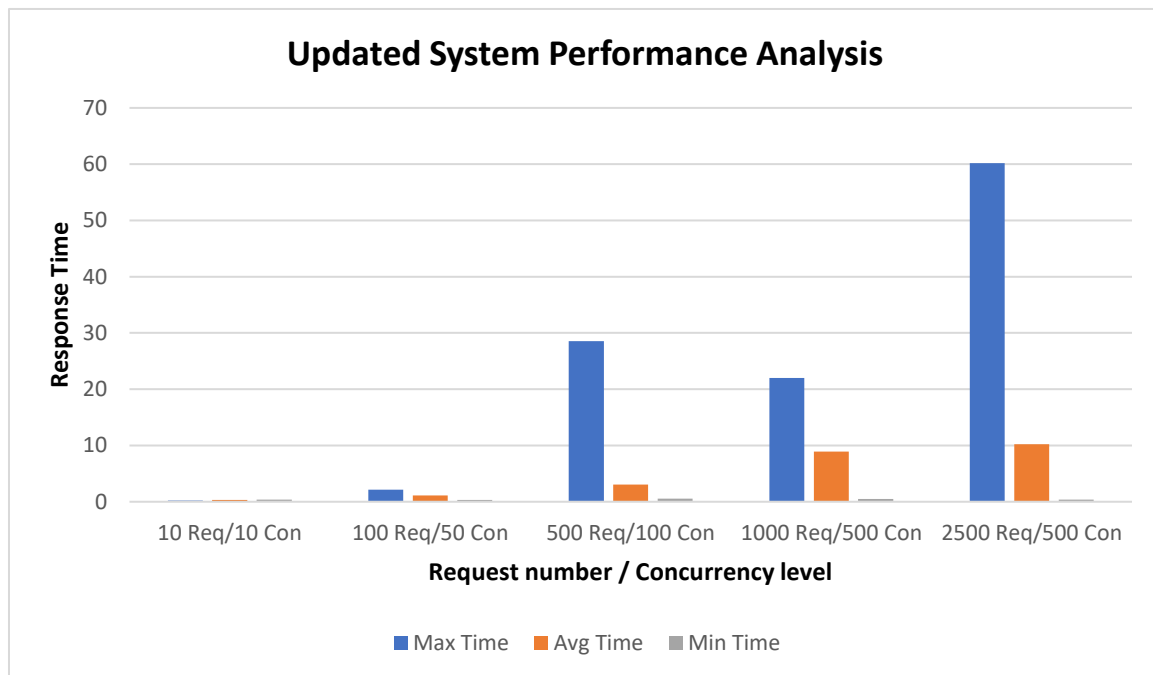


Figure 18:Updated System Performance Analysis

4.1. Results Analysis :

Increased Capacity: The new chart includes a data point for 2500 requests with 500 concurrent users, which was not present in the previous analysis. And this is because the system has been scaled up to handle a higher number of concurrent requests.

Response Time Analysis: The overall time response has significantly decreased especially the time assigned to **1000 Request** and **500 concurrent requests** .

Scalability Insights: The addition of a higher concurrency and request level (**2500 Req/500 Con**) without a drastic change in average response times suggests that the system scales well.

4.2. Error Rate Analysis Across Load Conditions :

we will proceed to a 3D scatter plot which will provide us with more reliability during the performance testing (*see figure 19*);

Error Count vs Number of Requests and Concurrency Level

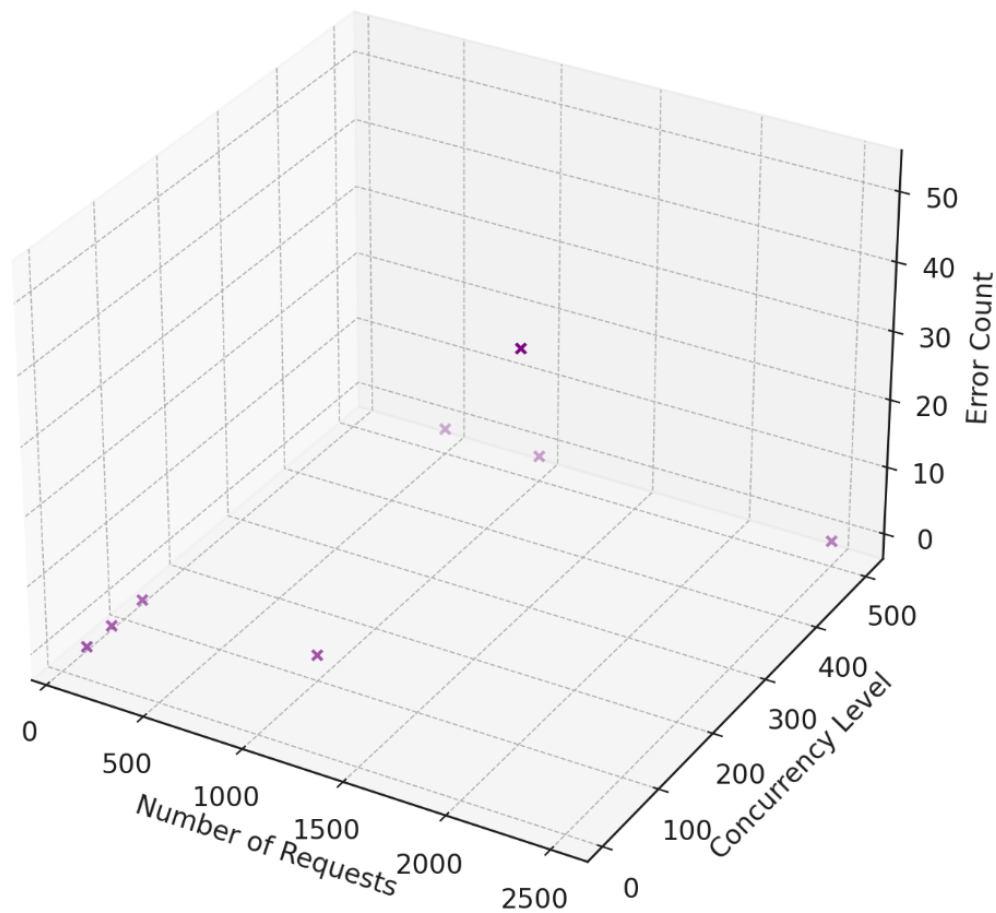


Figure 19: Scatter Plot of the Error count Based on Requests and Concurrency (updated)

5. Overall Comparison and Conclusion:

5.1. Error Count vs. Load:

When the number of requests and concurrency levels climb in the original system, we see that error counts also rise. This is to be expected; when systems get closer to their capacity limitations, failures tend to occur more often. Better performance is shown if the scaled-up system has a greater threshold before mistake counts rise.

5.2. Success Count at High Load:

the scaled-up system demonstrates a higher success count at similar or higher loads than the original system, this suggests that the scale-up has effectively increased the system's reliability and robustness.

5.3. Response Time Under Load:

A crucial metric for user experience is the response time as the system is subjected to higher loads. Our scaled-up system maintains lower response times at higher concurrency levels compared to the original system, this is a strong indicator that the scale-up has enhanced the system's capacity to handle more simultaneous users without degradation in performance.

5.4. Conclusion:

The **enhanced system** demonstrates a **fortified capacity** to manage augmented loads, reflected in **diminished error rates**, a testament to its **bolstered performance** and **refined error** management strategies. Preservation or **enhancement of response times** in the face of amplified demand signals a system that remains agile and responsive, assuring users of seamless interactions even amidst substantial operational pressures. An uplift in successful transactions, particularly at the juncture of **heightened concurrency** and **voluminous requests**, speaks to the system's augmented reliability and fortitude, eclipsing its predecessor's capabilities. When the upgraded infrastructure consistently triumphs over its antecedent across critical performance indicators, it is indicative of a scale-up triumph. Such a system has been adeptly fine-tuned to not only accommodate a surge in user traffic but to also expedite its responses and solidify the dependability of the user experience.

X. System Pricing Optimization and security:

In the next section we will be discussing the methodology and strategy we are going to adopt in order to optimize system cost and make it more efficient. Our strategy encompasses a multi-pronged approach to optimize AWS cloud architecture costs while ensuring high system availability and performance for our clients.

1. EC2 Instances:

We have decided to use **EC2 Instances** for most of the component of our system , we have used mainly 3 ; 1 for the master and 1 for running our application backend an 1 as a worker node , and then we can automate scaling up to minimize costs .Also we will be using the free tier that comes with 750 hours per month of a **t2.micro** or t3.micro instance, which is enough hours to run continuously each month for **12 months [11]** .

2. DynamoDB:

We have already estimated our capacity of **DynamoDB** database reading and writing not in the range of **1000 read/write** unit , so we consider purchasing reserved capacity for **DynamoDB** when we commit to a specific amount of read and write throughput for a one or three-year term and receive a discount over on-demand pricing. [11]

3. Amazon SQS:

For **Amazon SQS**, we're utilizing the 1 million requests per month included in the Free Tier. By structuring our message processing to batch operations effectively, we can process multiple messages within a single request, significantly decreasing the total request count. Additionally, we're implementing long polling to reduce the number of empty receive requests, further conserving our eligible Free Tier requests.

4. CloudWatch:

To ensure that the system remains within the AWS Free Tier while meeting user requirements, it is crucial to monitor usage closely using AWS Cost Explorer and CloudWatch. Set up billing alerts to notify you when usage approaches the free tier limits. Regularly review and adjust resource provisioning to align with actual usage patterns. Consider using AWS Budgets to manage costs proactively and to stay informed about how close your usage is to exceeding the free tier benefits.

5. Security :

In today's data-driven society, the implications of data security and sovereignty regarding environmental sensor data are critical. Sensitive and important information, such as geographic coordinates, climatic trends, and even personal information, are frequently contained in environmental sensor data. Protecting this data from unauthorized access and ensuring its integrity are essential. Data privacy regulations, such as GDPR and HIPAA, necessitate strict adherence to security protocols to safeguard sensitive information within environmental sensor datasets. Any breaches or mishandling of this data could lead to privacy infringements, legal consequences, and damage to an organization's reputation.

It is crucial to keep this data safe in today's data driven society, the implications of data security and sovereignty regarding environmental sensor data are critical. Sensitive and important information, such as geographic coordinates, climatic trends, and even personal information, are frequently contained in environmental sensor data. Data sovereignty also introduces another level of difficulty. Environmental sensors are often used in many nations and areas, each of which has its own rules and regulations pertaining to data privacy.

XI. Conclusion:

In concluding this report on our cloud computing project was a valuable experience for us , we have outlined a thorough process from conception to the effective implementation and expansion of a reliable cloud-based air quality data management system By using agile development methods, careful planning, and wise use of AWS resources, we have built a system that not only satisfies original criteria but also performs better than anticipated in terms of scalability, reliability, and performance.

Our initial objectives to leverage cloud computing for efficient data handling and real-time processing were achieved through the deployment of services like Amazon EC2, DynamoDB, SQS, and CloudWatch. The system was engineered to handle real-time data streams and cater to user queries through a responsive web interface, ensuring data integrity and accessibility remained paramount.

The performance testing stage was quite informative, giving us priceless knowledge about how the system behaved under different load scenarios. The results were encouraging, demonstrating that our scaling strategies were effective. As user demand increased, our system scaled seamlessly, maintaining low error rates and consistent response times, thereby affirming our commitment to user satisfaction.

Our capacity to manage several requests at once has significantly improved, according to the error count study conducted both before and after the scale-up. This enhancement not only contributed to a more stable system but also instilled confidence in our scaling approach, which proved to be both resilient and cost-effective.

As we close this chapter, we look forward to the continued evolution of our system, bolstered by the lessons learned and successes achieved. We are committed to realizing the full potential of the cloud and are prepared to embrace new technologies and approaches to improve system performance and user experiences.

XII. References :

- [1] VentureBeat : How cloud computing has changed the future of internet technology;
<https://venturebeat.com/datadecisionmakers/how-cloud-computing-has-changed-the-future-of-internet-technologyT>
- [2] Datamation : How Cloud Computing Changed the World ;
<https://www.datamation.com/cloud/how-cloud-computing-changed-the-world/>
- [3] Westlake : Is cloud computing good for the environment ?;
<https://www.westlake-it.co.uk/news/2022/08/30/is-cloud-computing-good-for-the-environment/>
- [4] CC_Assignment2023.pdf ;
<https://www.accenture.com/us-en/cloud/insights/cloud-computing-index>
- [5] AWS : Cloud computing avec AWS ;
<https://aws.amazon.com/fr/what-is-aws/>
- [6] Geeksforgeeks : Introduction to Amazon Web Services ;
https://www.espressif.com/sites/default/files/documentation/esp32_technical_reference_manual_en.pdf
- [7] AWS : Amazon EC2 ;
<https://aws.amazon.com/fr/ec2/>
- [8] AWS : Amazon DynamoDB ;
<https://aws.amazon.com/fr/pm/dynamodb/>
- [9] AWS : Amazon SQS ;
<https://aws.amazon.com/fr/pm/sqs/>
- [10] AWS CloudWatch ;
<https://aws.amazon.com/fr/pm/CloudWatch/>
- [11] AWS Calculator ;
<https://calculator.aws/>

XIII. Appendix :

```
-----CC-Initialisation.ipynb-----
# **Cloud Computing Assignment**
**Introduction** : This code is meant for the automation for the set up and
the deployment of our distributed computing environment that consists of
various elements including DynamoDB tables, SQS queues, and EC2 instances .We
are going to deploy a web application where the user can retrieve valuable
informations about the air quality according to various criteria(location,
date, range .....),The script is loaded with error handlers for easier
debugging.

**Attention !** : You can find further information about the execution in the
file "Instructions.pdf".
#pip install boto3
#pip install subprocess
#pip install paramiko
#pip install time
import boto3
import subprocess
import time
import paramiko
def create_DB(dynamoDB):
    table = dynamoDB.create_table(
        TableName='QualityAirTable',
        KeySchema=[{'AttributeName': 'id', 'KeyType': 'HASH'}],
        AttributeDefinitions=[{'AttributeName': 'id', 'AttributeType': 'S'}],
        ProvisionedThroughput={'ReadCapacityUnits': 1000,
'WriteCapacityUnits': 1000}
    )
    table.wait_until_exists()
    return table
def create_sqs_queue(SQS, queue_name):
    try:
        response = SQS.create_queue(QueueName=queue_name)
        print(f"Queue '{queue_name}' created successfully. URL:
{response['QueueUrl']}")
        return response['QueueUrl']
    except Exception as e:
        print(f"Error creating SQS queue '{queue_name}': {e}")
        return None
def create_instance(ec2, ami_id, instance_type, security_group_ids, key_name,
user_data_script, instance_name):
    try:
        instances = ec2.run_instances(
            ImageId=ami_id,
            MinCount=1,
```

```

        MaxCount=1,
        InstanceType=instance_type,
        UserData=user_data_script,
        SecurityGroupIds=security_group_ids,
        KeyName=key_name,
        TagSpecifications=[
            {
                'ResourceType': 'instance',
                'Tags': [
                    {
                        'Key': 'Name',
                        'Value': instance_name,
                    },
                ],
            },
        ],
    )['Instances']

    instance_id = instances[0]['InstanceId']
    print(f"Instance {instance_id} ({instance_name}) created, waiting for
it to run...")

    ec2.get_waiter('instance_running').wait(InstanceIds=[instance_id])
    print(f"Instance {instance_id} ({instance_name}) is running.")
    instance_description =
ec2.describe_instances(InstanceIds=[instance_id])
    time.sleep(60)
    return instance_description['Reservations'][0]['Instances'][0]

except Exception as e:
    print(f"Error creating EC2 instance: {e}")
    return None

def upload_script(ec2, instance_id, script_local_path, key_path):
    instance_description = ec2.describe_instances(InstanceIds=[instance_id])
    DNS =
instance_description['Reservations'][0]['Instances'][0]['PublicDnsName']

    scp_command = f"scp -o StrictHostKeyChecking=no -i {key_path}
{script_local_path} ec2-user@{DNS}:/home/ec2-user/"
    scp_result = subprocess.run(scp_command, shell=True,
stdout=subprocess.PIPE, stderr=subprocess.PIPE, text=True)

    if scp_result.returncode != 0:
        print("SCP Error (stdout):", scp_result.stdout)
        print("SCP Error (stderr):", scp_result.stderr)
        return False
    time.sleep(5)
    return True

```

```

def execute_script(ec2, instance_id, key_path, execution_file):
    instance_description = ec2.describe_instances(InstanceIds=[instance_id])
    DNS =
instance_description['Reservations'][0]['Instances'][0]['PublicDnsName']

    # Execute the script immediately
    ssh_command = f"ssh -o StrictHostKeyChecking=no -i \"{key_path}\" ec2-
user@{DNS} \"/usr/bin/python3 /home/ec2-user/{execution_file} > /home/ec2-
user/execute.log 2>&1 &\""

    ssh_result = subprocess.run(ssh_command, shell=True,
stdout=subprocess.PIPE, stderr=subprocess.PIPE, text=True)

    if ssh_result.returncode != 0:
        print("Execution Error (stdout):", ssh_result.stdout)
        print("Execution Error (stderr):", ssh_result.stderr)
        return False

    return True

def setup_ClientApp(instance, app_zip_path, public_ip):
    try:
        key =
paramiko.RSAKey.from_private_key_file('C:/Users/elkar/Downloads/labsuser.pem')
        client = paramiko.SSHClient()
        client.set_missing_host_key_policy(paramiko.AutoAddPolicy())

        DNS = instance['PublicDnsName']
        client.connect(hostname=DNS, username='ec2-user', pkey=key)

        sftp = client.open_sftp()
        sftp.put(app_zip_path, '/home/ec2-user/ClientApp.zip')
        sftp.close()

        commands = [
            'sudo yum update -y',
            'sudo yum install python3 python3-pip unzip -y',
            'unzip /home/ec2-user/ClientApp.zip -d /home/ec2-user/app/',
            'pip3 install --user -r /home/ec2-
user/app/ClientApp/requirements.txt'
        ]
        for command in commands:
            stdin, stdout, stderr = client.exec_command(command)
            print(stdout.read())

        print(f"Application deployed. Access it at http://{public_ip}:8501")

```

```

        client.close()
        return True # Indicate successful completion
    except Exception as e:
        print(f"Error setting up ClientApp instance: {e}")
        return False
def main():
    ec2 = boto3.client('ec2', region_name='us-east-1')
    dynamoDB = boto3.resource('dynamodb', region_name='us-east-1')
    SQS = boto3.client('sqs', region_name='us-east-1')

    ami_id = 'ami-079db87dc4c10ac91' # Replace with your AMI ID
    instance_type = 't2.micro'
    security_group_ids = ['sg-0e517e6668cedabd3'] # Replace with your
security group ID
    key_name = 'vockey' # Replace with your key name
    key_path = 'C:/Users/elkar/Downloads/labsuser.pem' # Replace with your
key path

    # User data scripts
    user_data_master = '''#!/bin/bash
        sudo yum update -y
        sudo yum install python3 python3-pip -y
        sudo yum install cronie -y
        sudo service crond start
        sudo chkconfig crond on
        sudo pip3 install requests boto3
        echo "0 0 * * * /usr/bin/python3 /home/ec2-
user/master.py" | crontab -
        ''' # User data for 'Master' instance

    user_data_client_load = '''#!/bin/bash
        sudo yum update -y
        sudo yum install python3 python3-pip -y
        sudo pip3 install requests boto3
        ''' # User data for 'Test_Instance' instance

    # Create DynamoDB table
    create_DB(dynamoDB)

    # Create SQS queues
    request_send_queue_url = create_sqs_queue(SQS, 'RequestSend')
    request_receive_queue_url = create_sqs_queue(SQS, 'RequestReceive')

    # Launch and set up the 'Master' EC2 instance
    master_instance = create_instance(ec2, ami_id, instance_type,
security_group_ids, key_name, user_data_master, 'Master')
    if master_instance:

```



```

        if upload_script(ec2, master_instance['InstanceId'],
'C:/Users/elkar/Downloads/430915_CC_Code/Master_Instance/master.py',
key_path):
            if execute_script(ec2, master_instance['InstanceId'], key_path,
'master.py'):
                print("Script uploaded and executed on the 'Master'
instance.")
            else:
                print("Failed to execute script on the 'Master' instance.")
        else:
            print("Failed to upload script to the 'Master' instance.")
            time.sleep(100)

# Launch and set up the 'ClientApp' EC2 instance after 'Master' instance
client_app_zip_path =
'C:/Users/elkar/Downloads/430915_CC_Code/ClientApp.zip'
client_app_instance = create_instance(ec2, ami_id, instance_type,
security_group_ids, key_name, '', 'ClientApp')
if client_app_instance:
    public_ip = client_app_instance.get('PublicIpAddress')
    if setup_ClientApp(client_app_instance, client_app_zip_path,
public_ip):
        print(f"'ClientApp' instance setup completed successfully.")
    else:
        raise Exception("Failed to set up 'ClientApp' instance.")
else:
    raise Exception("Failed to create 'ClientApp' instance.")

# Launch and set up the 'Worker' EC2 instance after 'Master' instance
worker_instance = create_instance(ec2, ami_id, instance_type,
security_group_ids, key_name, user_data_client_load, 'Worker')
if worker_instance:
    if upload_script(ec2, worker_instance['InstanceId'],
'C:/Users/elkar/Downloads/430915_CC_Code/Worker_node/Worker.py', key_path):
        if execute_script(ec2, worker_instance['InstanceId'], key_path,
'Worker.py'):
            print("Script uploaded and executed on the 'Worker' instance
.")
        else:
            print("Failed to execute script on the 'Worker' instance.")
    else:
        print("Failed to upload script to the 'Worker' instance.")

# Launch and set up the 'Test_Instance' EC2 instance
client_load_instance = create_instance(ec2, ami_id, instance_type,
security_group_ids, key_name, user_data_client_load, 'Test_Instance')
if client_load_instance:

```

```

        if upload_script(ec2, client_load_instance['InstanceId'],
'C:/Users/elkar/Downloads/430915_CC_Code/Test_Instance/TestPerformance.py',
key_path):
            if execute_script(ec2, client_load_instance['InstanceId'],
key_path, 'TestPerformance.py'):
                print("Script uploaded and executed on the 'Test_Instance'
instance.")
            else:
                print("Failed to execute script on the 'Test_Instance'
instance.")
        else:
            print("Failed to upload script to the 'Test_Instance' instance.")

if __name__ == '__main__':
    main()

-----App.py-----
""" This is the Backend program of the ClientApp.
    It is used to query the data from the DynamoDB table and return the
results in CSV format.
    Attention !! : You can find further information about the execution in the
file "Instructions.pdf"
"""

# Import necessary libraries
from flask import Flask, request, jsonify, render_template, Response
from flask_cors import CORS
import boto3
from boto3.dynamodb.conditions import Key, Attr
import decimal
import json
import csv
from io import StringIO
import threading
import uuid
from threading import Lock

app = Flask(__name__)
CORS(app) # Enable CORS for all routes

# Helper class to handle Decimal types in DynamoDB
class DecimalEncoder(json.JSONEncoder):
    def default(self, o):
        if isinstance(o, decimal.Decimal):
            if o % 1 > 0:
                return float(o)
            else:
                return int(o)

```

```

        return super(DecimalEncoder, self).default(o)

# Initialize the DynamoDB client
dynamodb = boto3.resource(
    'dynamodb',
    aws_access_key_id='ASIAXPHZLJMCR65BIJ74',
    aws_secret_access_key='o1VpRBo6IM61Mtw+UecBqrIf1He
Qn22Ke7c5W3wv',
    aws_session_token='FwoGZXIvYXdzEKv////////wEaD0w
jsIuUW4eNVBTG9CLCARGDcY/Hqd0i5Q+AQLs1fhIV9f1znTftR7mZmcS8xnCsbV0NKi9ILdTL4trD8
GJKRBg60TrA06Ceq/WyxNtpnPZygWZ2ynm5FhELbyPpGFhDKkC80h+Lq00ckkrWpjPq61xkP14eJK2
JYtfAmf6mqODqjTt4Gq16wLV12/605kkTtjMLMotIjX4riobe/ELmhZgY4DaryTp7Iv3gXxL9IjfPn
+otdGqPyxBriLpvw9XAEjFganwJrF741zbMjsxLN3TDKNPiy6wGMi1Pn40+qMEjD9S99DnogNguMSu
BmE42E2zVDs3abAuQ9GAwmUBHa/b6hLcLsiQ=',
    region_name='us-east-1'
)

sqs = boto3.client(
    'sqs',
    aws_access_key_id='ASIAXPHZLJMCR65BIJ74',
    aws_secret_access_key='o1VpRBo6IM61Mtw+UecBqrIf1He
Qn22Ke7c5W3wv',
    aws_session_token='FwoGZXIvYXdzEKv////////wEaD0w
jsIuUW4eNVBTG9CLCARGDcY/Hqd0i5Q+AQLs1fhIV9f1znTftR7mZmcS8xnCsbV0NKi9ILdTL4trD8
GJKRBg60TrA06Ceq/WyxNtpnPZygWZ2ynm5FhELbyPpGFhDKkC80h+Lq00ckkrWpjPq61xkP14eJK2
JYtfAmf6mqODqjTt4Gq16wLV12/605kkTtjMLMotIjX4riobe/ELmhZgY4DaryTp7Iv3gXxL9IjfPn
+otdGqPyxBriLpvw9XAEjFganwJrF741zbMjsxLN3TDKNPiy6wGMi1Pn40+qMEjD9S99DnogNguMSu
BmE42E2zVDs3abAuQ9GAwmUBHa/b6hLcLsiQ=',
    region_name='us-east-1'
)

request_queue_name = 'RequestSend'
response_queue_name = 'RequestReceive'
request_queue_url =
sqs.get_queue_url(QueueName=request_queue_name)['QueueUrl']
response_queue_url =
sqs.get_queue_url(QueueName=response_queue_name)['QueueUrl']
results = {}
active_requests = 0
active_requests_lock = Lock()
@app.route('/')
def index():
    return render_template('Request.html')

@app.route('/api/query', methods=['POST'])
def query_data():
    global active_requests
    with active_requests_lock:
        active_requests += 1

```

```

try:
    data = request.json
    if active_requests > 100:
        unique_id = str(uuid.uuid4())
        data['id'] = unique_id
        try:
            sqs.send_message(QueueUrl=request_queue_url,
MessageBody=json.dumps(data))
        except boto3.exceptions.Boto3Error as e:
            print(f"Error sending message to SQS: {e}")
            return jsonify({"error": "Error processing request"}), 500
        response = jsonify({"id": unique_id}), 202
    else:
        response = direct_query_processing(data)
finally:
    with active_requests_lock:
        active_requests -= 1

return response

def direct_query_processing(data):
    try:
        table = dynamodb.Table('QualityAirTable')
        filtering_exp = build_filter_expression(data)
        response = table.scan(FilterExpression=filtering_exp)
        return generate_csv_response(response.get('Items', []))
    except Exception as e:
        print(f"Error in direct query processing: {e}")
        return Response("Internal Server Error", status=500)

def build_filter_expression(data):
    filtering_exp = Attr('timestamp').between(data['startDate'],
data['endDate'])
    if data.get('country'):
        filtering_exp &= Attr('loc_country').eq(data['country'])
    if data.get('range'):
        filtering_exp &= Attr('Range').eq(data['range'])
    if data.get('minAqi') is not None:
        filtering_exp &= Attr('AQI').gte(int(data['minAqi']))
    if data.get('maxAqi') is not None:
        filtering_exp &= Attr('AQI').lte(int(data['maxAqi']))
    return filtering_exp

def generate_csv_response(items):
    if not items:
        return Response("No data found.", mimetype='text/plain', status=404)
    si = StringIO()
    cw = csv.writer(si)

```

```

@app.route('/api/results/<query_id>', methods=['GET'])
def get_results(query_id):
    # Assuming 'results' is a dictionary holding the results temporarily
    if query_id in results:
        # Retrieve the message corresponding to the query_id
        message = results.pop(query_id)

        # Check if 'id' key is present in the message and remove it
        message.pop('id', None)

        # Convert the message back to JSON and return as response
        return jsonify(message)
    else:
        return jsonify({"status": "processing"}), 202

def poll_response_queue():
    while True:
        try:
            messages_response = sqs.receive_message(
                QueueUrl=response_queue_url,
                MaxNumberOfMessages=10,
                WaitTimeSeconds=20
            )

            if 'Messages' in messages_response:
                for message in messages_response['Messages']:
                    body = json.loads(message['Body'])
                    receipt_handle = message['ReceiptHandle']

                    with active_requests_lock:
                        query_id = body['id']
                        query_results = body['results']
                        results[query_id] = query_results

                    sqs.delete_message(
                        QueueUrl=response_queue_url,
                        ReceiptHandle=receipt_handle
                    )
        except Exception as e:
            print(f"Error polling response queue: {e}")

threading.Thread(target=poll_response_queue, daemon=True).start()

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=8501, debug=False)

```

```
""" This script fetches the sensor data from the luftdaten.info
    API and stores it in a DynamoDB table.
    Attention !! : You can find further information about the execution in the
    file "Instructions.pdf"
"""

# Import necessary libraries
import requests
import boto3
from datetime import datetime
from decimal import Decimal, InvalidOperation
import logging
import time

# Initialize logging
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s -
%(message)s')

# Initialize a DynamoDB client
dynamodb = boto3.resource('dynamodb',
                           aws_access_key_id='ASIAXPHZLJMCR65BIJ74',
                           aws_secret_access_key='olVpRBo6IM6lMtw+UecBqrIf1He
Qn22Ke7c5W3wv',
                           aws_session_token='FwoGZXIvYXdzEKv////////wEaDOW
jsIuUW4eNVBTG9CLCARGDcY/HqdOi5Q+AQLS1fhIV9f1znTftR7mZmcS8xnCsbV0NKi9ILdTL4trD8
GJKRBg60TrA06Ceq/WyxNtpnPZygWZ2ynm5FhELbyPpGFhDKkC80h+Lq00ckkrWpjPq61xkP14eJK2
JYtfAmf6mqODqjTt4Gq16wLV12/605kkTtjMLMotIjX4riobe/ELmhgzY4DaryTp7Iv3gXxL9IjfPn
+otdGqPyxBriLpvw9XAEjFganwJrF74lzbMjsxLN3TDKNPiy6wGMi1Pn40+qMEjD9S99DnogNguMSu
BmE42E2zVDS3abAuQ9GAwmUBHa/b6hLcLsiQ=',
                           region_name='us-east-1')

# Define our DynamoDB table ,"QualityAirTable"
table = dynamodb.Table('QualityAirTable')

# Define a function to calculate the AQI and range label
def calculate_aqi(p1, p2):
    aqi_sensors = {
        'PM2.5': [
            (0, 11, 1), (12, 23, 2), (24, 35, 3), (36, 41, 4), (42, 47, 5),
            (48, 53, 6), (54, 58, 7), (59, 64, 8), (65, 70, 9), (71,
float('inf'), 10)
        ],
        'PM10': [
            (0, 16, 1), (17, 33, 2), (34, 50, 3), (51, 58, 4), (59, 66, 5),
            (67, 75, 6), (76, 83, 7), (84, 91, 8), (92, 100, 9), (101,
float('inf'), 10)
    ]
```

```

    ]
}

# Define the AQI ranges for PM2.5 and PM10
aqi_p1 = next((aqi for low, high, aqi in aqi_sensors['PM2.5'] if p1 is not
None and low <= p1 <= high), None)
aqi_p2 = next((aqi for low, high, aqi in aqi_sensors['PM10'] if p2 is not
None and low <= p2 <= high), None)

if aqi_p1 is not None or aqi_p2 is not None:
    aqi = max(filter(lambda x: x is not None, [aqi_p1, aqi_p2]))
    range_label = 'Low' if aqi <= 3 else 'Medium' if aqi <= 6 else 'High'
if aqi <= 9 else 'Very High'
    return aqi, range_label
else:
    logging.error("Both PM values are None, cannot calculate AQI.")
    return None, None

# Define a function to convert floats to decimals
def float_to_decimal(obj):
    if isinstance(obj, float):
        return Decimal(str(obj))
    elif isinstance(obj, dict):
        return {k: float_to_decimal(v) for k, v in obj.items()}
    elif isinstance(obj, list):
        return [float_to_decimal(x) for x in obj]
    return obj

# Define a function to store the data in DynamoDB
def store_data(data):
    logging.info("Storing data in DynamoDB...")
    for index, entry in enumerate(data):
        try:
            entry = float_to_decimal(entry)
            p1_value = next((Decimal(sdv['value']) for sdv in
entry['sensordatavalues'] if sdv['value_type'] == 'P1'), None)
            p2_value = next((Decimal(sdv['value']) for sdv in
entry['sensordatavalues'] if sdv['value_type'] == 'P2'), None)

            if p1_value is None and p2_value is None:
                logging.info(f"Skipping entry {index} as it doesn't contain P1
or P2 values.")
                continue

            aqi, range_label = calculate_aqi(p1_value, p2_value)
            if aqi is None or range_label is None:
                continue

```

```

        item = {
            'id': str(entry['id']),
            'timestamp': entry['timestamp'],
            'P1': p1_value,
            'P2': p2_value,
            'loc_id': str(entry['location']['id']),
            'loc_lat': entry['location']['latitude'],
            'loc_long': entry['location']['longitude'],
            'loc_alt': entry['location']['altitude'],
            'loc_country': entry['location']['country'],
            'AQI': aqi,
            'Range': range_label
        }

        table.put_item(Item=item)
        logging.info(f"Stored entry {index} with ID {item['id']} in
DynamoDB.")

    except Exception as e:
        logging.error(f"Error at entry {index}: {e}")

# Define a function to fetch the sensor data
def fetch_sensor_data():
    url = "https://data.sensor.community/static/v2/data.24h.json"
    try:
        logging.info("Fetching sensor data...")
        response = requests.get(url)
        response.raise_for_status()
        return response.json()
    except requests.RequestException as e:
        logging.error(f"Error fetching sensor data: {e}")
        raise

def main():
    logging.info("Script started.")
    sensor_data = fetch_sensor_data()
    store_data(sensor_data)
    logging.info("Data fetch and store completed.")

if __name__ == "__main__":
    main()

```

```
import requests
from concurrent.futures import ThreadPoolExecutor
import time

# Endpoint URL of your Flask application
endpoint_url = 'http://127.0.0.1:5000/api/query'

# Sample payload for your Flask app's expected input
payload = {
    'startDate': '2024-01-01 00:00:00',
    'endDate': '2024-01-01 01:00:00',
    'country': 'DE',
    'minAqi': 0,
    'maxAqi': 10,
    'range': 'Low'
}

# Function to make a POST request and measure response time
def make_request():
    start_time = time.time()
    try:
        response = requests.post(endpoint_url, json=payload)
        end_time = time.time()

        # Calculate the response time in seconds
        response_time = end_time - start_time

        if response.status_code == 200:
            return {'status': 'success', 'response_time': response_time}
        else:
            return {'status': 'error', 'response_time': response_time,
                    'error_message': response.text}
    except requests.exceptions.RequestException as e:
        end_time = time.time()
        response_time = end_time - start_time
        return {'status': 'error', 'response_time': response_time,
                'error_message': str(e)}

# Number of requests and concurrency level
num_requests = 1000
concurrency_level = 100
```

```

# Using ThreadPoolExecutor to simulate concurrent requests
success_count = 0
error_count = 0
total_response_time = 0
max_response_time = 0
min_response_time = float('inf')

with ThreadPoolExecutor(max_workers=concurrency_level) as executor:
    futures = [executor.submit(make_request) for _ in range(num_requests)]

    for future in futures:
        result = future.result()
        total_response_time += result['response_time']
        max_response_time = max(max_response_time, result['response_time'])
        min_response_time = min(min_response_time, result['response_time'])

        if result['status'] == 'success':
            success_count += 1
        else:
            error_count += 1
            print(f"Error: {result['error_message']}")

# Calculating average response time
average_response_time = total_response_time / num_requests if num_requests
else 0

# Output the results
print(f"Completed {num_requests} requests to the Flask app.")
print(f"Success Count: {success_count}, Error Count: {error_count}")
print(f"Average Response Time: {average_response_time:.4f} seconds")
print(f"Max Response Time: {max_response_time:.4f} seconds, Min Response Time: {min_response_time:.4f} seconds")

```