Politecnico di Milano

Software Engineering for Geoinformatics

REPORT

# Design Document for AQI-GDP App

*Authors*

Iyad Abdi

Filippo Bissi

Valerio Paoloni

*Person Code*

10808624

10438335

10401564

## Table of Contents

# 1. Introduction

## 1.1. Purpose

This design document describes the architecture, components, and interactions of an interactive client-server application for accessing, querying, and visualizing air quality data retrieved from existing public digital archives.

## 1.2. Scope

The interactive client-server application is designed to facilitate the access, querying, and visualization of air quality and weather sensor data from existing public digital archives. The system aims to be an essential tool for environmental researchers, policy makers, educators, and the general public, offering insights into air quality and weather conditions across various locations and time periods. The three main components of the system are:

- **Database**: This component is responsible for the storage of all air quality and weather sensor data ingested from various public digital archives. The database needs to be designed to handle a sufficient volume of data and support efficient querying mechanisms. The ingestion process, including any necessary data cleaning, normalization, and indexing, will be part of the scope. The creation or maintenance of the public digital archives from which the data is sourced.
- **Web Server (Backend)**: This component, which exposes a REST API, handles interactions between the client (dashboard) and the database. It processes client requests, executes queries against the database, and returns the requested data in a structured format that can be easily processed by the client. The development of the API, which includes defining endpoints, request/response structures, error handling, etc., is included in the scope.
- **Dashboard**: This client-side component provides an interactive user interface for requesting, processing, and visualizing the data from the server. This includes developing features for visualizing data in various formats (maps, dynamic graphs, etc.), providing tools for data processing (e.g., space-time aggregations, filters), and implementing user-friendly design and navigation. User account management or access control features. The system is assumed to be publicly accessible without the need for user registration or login.

## 1.3. Definitions, Acronyms, and Abbreviations

**Definitions:**

- **Pollution**: The presence of harmful substances or contaminants in the environment, resulting in adverse effects on ecosystems, human health, or the quality of air, water, or soil.

- **Sensors**: Devices or instruments used to detect and measure physical or chemical properties of the environment, such as air quality, temperature, humidity, or pollutant concentrations.
- **Application**: The software system or program designed to process, analyze, and visualize the data collected by the sensors and provide meaningful insights or services related to pollution monitoring.
- **Regions**: In the context of Global North and Global South, these regions are quasi-geographical, but more are related to countries' historical and economic positions within the global economy.

**Acronyms and abbreviations:**

- **API**: Application Programming Interface
- **REST**: Representational State Transfer
- **UI**: User Interface
- **GUI**: Graphical User Interface
- **DBMS**: Database Management System
- **UX**: User Experience
- **HTTP**: Hypertext Transfer Protocol
- **JSON**: JavaScript Object Notation
- **CSV**: Comma-Separated Values
- **FTP**: File Transfer Protocol
- **SQL**: Structured Query Language
- **AQI**: Air Quality Index
- **PM2.5**: Particulate Matter 2.5 micrometers or less in diameter

# 2. System Overview

## 2.1. System Architecture

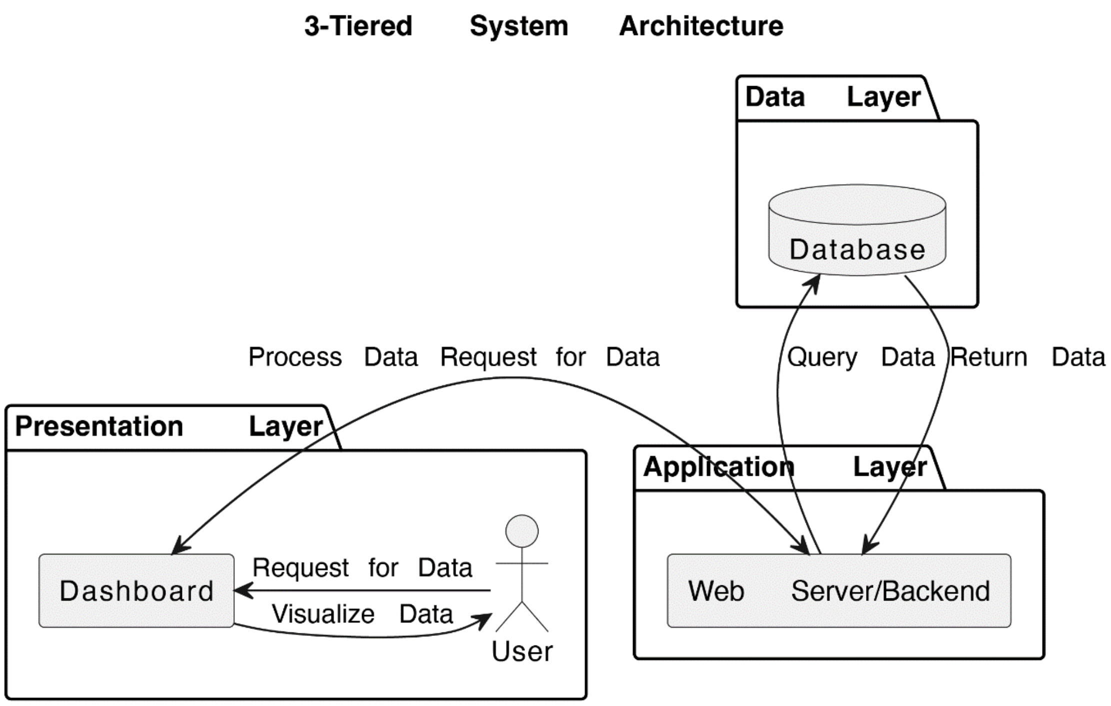
As the interactive client-server application is designed to help users access, query, and visualize air quality and weather sensor data, the data is ingested from public digital archives into our system's database. Once in our database, the data can be queried via a web server that exposes a REST API. Finally, a dashboard provides users with an interactive interface to visualize the data in various formats.

Our system architecture is a three-tiered model composed of:

a. **Data Layer (Database):** This is the foundational layer where all air quality and weather sensor data is stored. The data is ingested and cleaned from various public digital archives. The database is designed for high performance and scalability to accommodate a growing volume of data over time.

b. **Application Layer (Web Server/Backend):** This layer is the bridge between the data layer and the presentation layer. It exposes a REST API, which allows for querying the database and retrieving data. This layer is responsible for processing client requests, managing interactions with the database, and returning the requested data in a structured format.

c. **Presentation Layer (Dashboard):** This is the topmost layer that users interact with. It's a web-based user interface that leverages the REST API to request, process, and visualize data. The dashboard offers various data visualization tools, such as maps and dynamic graphs, to help users understand and interpret the air quality data.



## 2.2. System Workflow

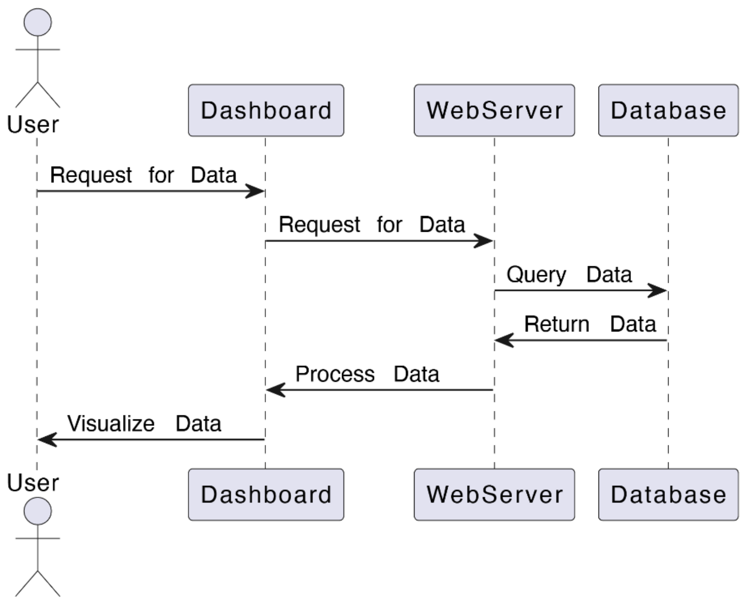
a. **Data Ingestion:** The system retrieves air data from the public digital archives and ingests it into the database. This process includes data cleaning, normalization, and indexing for efficient querying.

b. **Data Request:** Users interact with the dashboard to make requests for data based on their needs. This could be a request for AQI data from a specific city or group of cities within a particular timeframe.

c. **Data Retrieval:** The web server receives the request from the dashboard and translates it into a query that the database can understand. Once the data is retrieved from the database, the web server processes it into a structured format that can be easily visualized on the dashboard.

d. **Data Visualization:** The dashboard receives the processed data from the web server and visualizes it based on user preferences. This could be in the form of maps, dynamic graphs, or other visualization formats. Users can also request for space-time aggregations, toggle different base maps, and export data in various formats.

System Workflow - Data Ingestion, Request, Retrieval, Visualization



## 2.3. System Interactions

The architecture and workflow of the system ensure that it is scalable, efficient, and user-friendly, allowing users to easily access, query, and visualize air quality and weather sensor data.

**a.** The database interacts with the web server, providing data as requested.

**b.** The web server interacts with both the database and the dashboard. It retrieves data from the database, processes it, and sends it to the dashboard.

**c.** The dashboard interacts with the web server, sending data requests and receiving data for visualization.

## 2.4. Design Considerations

Several key factors will be considered to ensure the system's optimal design. These 7 design considerations will guide the development of the system, ensuring that it is robust, scalable, and user-friendly, while providing reliable and accurate service to its users.

These considerations are:

**a.** **Scalability**: The system needs to handle a necessary amount of data from the public digital archives and the number of users. This requires a scalable architecture, with a database capable of managing such volumes of data and a web server able to handle decent loads.

b. **Performance**: The system should provide fast query responses and real-time visualization to ensure a smooth user experience. This requires an efficient database design, optimized queries, and a well-designed REST API. The dashboard should also be optimized for performance, with efficient data processing and routinized data ingestion.

c. **Usability**: The dashboard should be intuitive and user-friendly, allowing users to easily navigate, make requests, and understand the visualized data. This requires a well-structured user interface, clear navigation, and interactive, accessible data visualization tools.

d. **Reliability**: The system needs to be reliable and available, providing consistent service to its users. This involves considerations around system redundancy, error handling, and robustness against failures.

e. **Maintainability**: The system should be designed to be easily maintained and updated. This involves using modular design principles, clear and comprehensive documentation, and following coding best practices for easier debugging and updates.

f. **Interoperability**: Given the system interacts with various online public digital archives, it should be designed to easily integrate with these external systems. This involves following standard protocols and formats, and building a flexible data ingestion process that can handle different types of data sources.

g. **Data Integrity**: The system needs to ensure the accuracy and consistency of the data. This requires careful selection and design of the data ingestion process, including data cleaning.

# 3. Component Design

## 3.1. Database

The Database is the foundation of the system, storing all air quality sensor data ingested from public digital archives.

### 3.1.1. Database Model

The system employs a relational database model, with one main entity:

**Measurement:** This table stores the actual sensor data and metadata for each sensor. Each row represents a sensor reading with unique sensor `id`, `aqi` (air quality reading), `timestamp` (date and time), `city_name` (city), `location` (latitude and longitude), `pm25`, as well as `city_url` (URL link to the sensor).

#### 3.1.2. Database Technology

The database will be implemented using PostgreSQL, a powerful, open-source object-relational database system. It provides scalability and performance benefits, ensuring efficient data storage and retrieval.

## 3.2. Web Server & Data Processing (Backend)

The web server forms the application layer of our system. It exposes a REST API for querying the database and retrieving data.

#### 3.2.1. API Endpoints

The server will expose the following RESTful endpoints:

- **GET /sensors:** Retrieves a list of all sensors and their metadata.
- **GET /measurements:** Retrieves measurements for a given sensor ID, measurement type, and time range.
- **POST /aggregations:** Submits a request for space-time aggregated data based on specified parameters.

#### 3.2.2. Web Server Technologies

The web server will be built using Flask, while pyscopg2 and pickle libraries will be used for database connection and data processing.

**Flask** is a micro web framework written in Python and can be used to create robust APIs. It will be the foundation of our REST API.

**psycopg2** is a PostgreSQL adapter for Python that allows for connecting, retrieving, and interacting with the PostgreSQL database. For the Flask REST API, psycopg2 can be used to establish a connection with a PostgreSQL database and perform various database operations such as querying, inserting, updating, and retrieving data.

**pickle** is a module for object serialization and deserialization used to store and transmit Python objects efficiently. It can be used to serialize and deserialize Python objects, allowing us to store and retrieve them efficiently from the database and send them as a response to an API request.

**jsonify** is a function provided by Flask that is commonly used in Flask REST APIs to convert Python objects into JSON (JavaScript Object Notation) format. It will convert our database data it into a JSON response that can be sent as an HTTP response.

## 3.3. Dashboard

The Dashboard is the user-facing component of the system, providing an interactive interface for users to request, process, and visualize data.

#### 3.3.1. Features

The Dashboard will offer the following features:

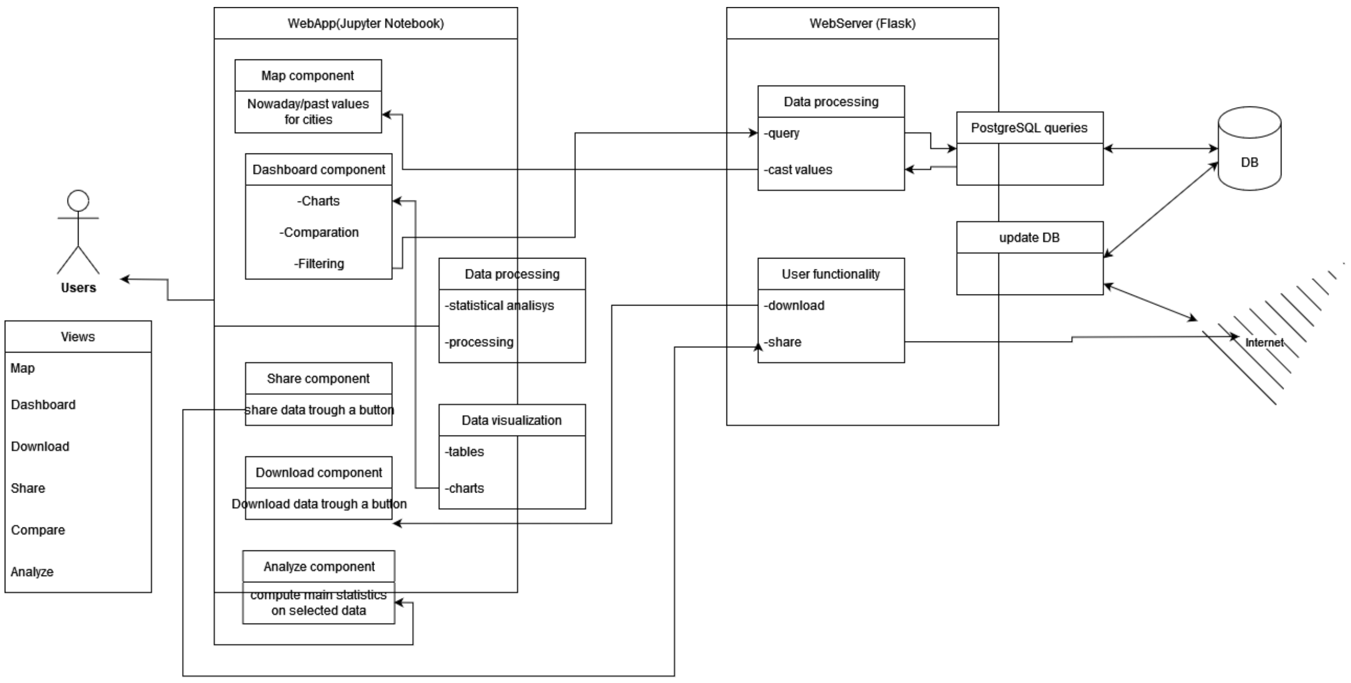
- **Map view:** An interactive map displaying sensor locations and AQI measurements, with the ability for users to toggle between base map styles.
- **Time-series graphs**: Stacked bar graph visualizing AQI measurements over time for selected cities or group of cities. Line graph visualizing AQI averages for each group of cities over time (Global North, Global South).
- **Data processing tools**: Options for users to aggregate and filter data based on selection and time parameters: cities/group-dropdowns and date-range.
- **Export options**: Users can export visualized data in PNG format.

### 3.3.2. Dashboard Technologies

The Dashboard will be built using **Dash**, a powerful and flexible Python framework for creating dynamic and responsive web-based dashboards and data visualization applications. It is built on top of Flask and Plotly.

Dash applications can be deployed as standalone web servers or embedded within existing Flask applications, and supports both local and cloud deployment options, making it easy to share and distribute our dashboard.

**Plotly** is a popular data visualization library, to create interactive and customizable charts and graphs.



Each component of the system is designed with specific technologies and structures to ensure they perform their roles effectively while seamlessly interacting with each other.

From data storage in the database, data retrieval and processing in the web server, to data visualization on the dashboard, each component serves a critical function in the system's operation.

## 4. Data Design

The Data Design section focuses on the structure and management of data within the system. This includes the data stored in the database as well as the data that is transferred between different components of the system.

### 4.1. Database Design

As mentioned in the System Overview, the system uses a relational database model to store the air quality sensor data. The database consists of the main entity: Measurement.

**Measurement Table:**

- `id` **(Primary Key):** Unique identifier for each sensor.
- `aqi`**:** AQI value for each measurement.
- `timestamp`**:** Time when the measurement was taken.
- `city_name`**:** Name of the sensor from that city.
- `location`**:** Geographical coordinates of the sensor (latitude, longitude).
- `pm25`**:** The recorded value of PM2.5.

### 4.2. Data Transfer

Data is transferred between different components of the system in a structured format. The main form of data transfer in this system is through the REST API exposed by the web server.

- **Request Data:** When the dashboard sends a request to the web server, it includes parameters such as the `id`, `aqi`, `city_name`, `timestamp` range, etc. in the request. This data is used by the web server to query and retrieve the relevant data from the REST API.
- **Response Data:** The web server sends the requested data back to the dashboard in a structured JSON format. As an example, a response should look something like this:

```
[
    {
        "id": 1,
        "aqi": 69,
        "timestamp": "2023-05-14T12:00:00Z",
        "city_name": "London",
        "latitude": 51.5073509,
        "longitude": -0.1277583,
        "pm25": 69.0
    },
]
```

This response includes an array of measurements, with each measurement including the `id`, `aqi`, `city_name`, `timestamp`, `latitude`, `longitude` and `pm25`.

### 4.3. Data Ingestion

Data is ingested into our database using a free API and is received in JSON format. Upon retrieval, the data undergoes cleaning and transformation processes to ensure its compatibility with the database format.

The specific operations performed on the data during processing may vary depending on its structure and format. These steps are designed to prepare the data for efficient and reliable storage in the database.

The system's data design prioritizes efficient storage and effective transfer of data between different components. It also addresses the necessary processing requirements during the data ingestion process.

In summary, the system ingests data in JSON format from an API. The data is processed to ensure compatibility with the database and is then stored efficiently and reliably.

## 5. Interface Design

Interface design in this context refers to the interaction points between different components of the system. For this application, the primary interfaces are the REST API between the database and the web server, and between the web server and the dashboard.

## 5.1. Database to Web Server Interface

The database and the web server communicate via SQL queries. The web server constructs and sends SQL queries to the database to insert, update, retrieve, or delete data. The database then returns a response to the web server, indicating the status of the operation and any data requested.

For instance, to retrieve measurements for a specific sensor, the web server might send a query like:

```
SELECT * FROM Measurement WHERE id = 1 AND timestamp
BETWEEN '2023-05-01' AND '2023-05-31';
```

The database would then return all measurements that meet the criteria.

## 5.2. Web Server to Dashboard Interface

The web server and dashboard communicate via the REST API. The dashboard sends HTTP requests to the web server, and the web server returns HTTP responses.

The REST API exposed by the web server has several endpoints, as mentioned in the Component Design section. Here's a more detailed example of the interface for one of these endpoints:

- **GET /measurements**
    - Request: The dashboard sends a GET request to this endpoint, including parameters for `id`, `city_name`, and `timestamp` range in the request.
    - Response: The web server sends back a JSON object containing an array of measurements. Each measurement includes `id`, `aqi`, `city_name`, `timestamp`, and `longitude` and `latitude`.

Example of a GET request from the dashboard:

```
GET/measurements?id=1&city_name=London&start_time=2023-05-
01&end_time=2023-05-31 HTTP/1.1

Host: http://localhost:5000
```

Example of a response from the web server:

```
HTTP/1.1 200 OK

Content-Type: application/json

[

    {
        "id": 1,
        "aqi": 72,
        "timestamp": "2023-05-14T12:00:00Z",
        "city_name": "London",
        "latitude": 51.5073509,
        "longitude": -0.1277583,
        "pm25": 72.0
    },
]
```

In this way, the interfaces between the different components of the system are designed to facilitate efficient and reliable communication, with clearly defined protocols and data structures.

## 6. User Interface Design

The user interface for this application is a Jupyter Notebook dashboard that provides an interactive interface for users to query and visualize the air quality and weather sensor data. The dashboard interacts with the backend via the REST API and uses various libraries for data processing and visualization.

### 6.1. Dashboard Layout

The dashboard is designed to be intuitive and user-friendly, with clear instructions and visual cues. The layout of the dashboard is as follows:

- **Query Parameters:** At the top of the dashboard, there are input fields where users can enter their query parameters. These might include the `city_name`, `city_groups` and `timestamp` range. Users can input their desired parameters and submit the query to retrieve the relevant data.
- **Data Table:** Below the query parameters, there is a table that displays the retrieved data in a structured format. This table updates dynamically based on the user's query.
- **Visualizations:** Below the data table, there are various visualizations of the data. These might include maps and graphs, depending on the nature of the data and the user's query.

### 6.2. Interactive Maps

The dashboard uses the Dash Plotly library to create an interactive map. This map can be used to visualize the geographic distribution of the city sensors, as well as the AQI from each sensor. For instance, users might see a map of cities with markers for each sensor, with the color of the marker indicating the AQI for that city.

### 6.3. Dynamic Maps

The dashboard uses the Plotly library to create dynamic graphs. These graphs can be used to visualize trends in the data over time or across different cities and city groups. For instance, users might see a bar graph representing the AQI value of selected cities, or a line graph comparing the average AQI for a group of cities over a specified time period.

### 6.4. Data Processing

The dashboard uses the **Pandas** and **Numpy** libraries for data processing. This involves cleaning the data, transforming it into a suitable format for the visualizations, and performing any necessary calculations or aggregations.

### 6.5. User Interaction

The dashboard is designed to be interactive, allowing users to explore the data in a hands-on way. Users can adjust their query parameters, zoom in or out on the maps, hover over the graphs to see more details, and more. The dashboard updates dynamically based on the user's interaction, providing a responsive and engaging user experience.

Overall, the user interface design of the dashboard aims to provide a user-friendly and interactive way for users to access, query, and visualize the air quality data of our cities.

## 7. System Deployment

The deployment of the system involves setting up the environment where the application will run, configuring the application according to the environment, and starting the application.

### 7.1. Environment Setup

Firstly, the environment where the application will run needs to be prepared. This involves setting up the servers, installing the necessary software, and configuring the network. The following environments need to be set up:

- **Database Server:** A server to host the database needs to be set up. This server needs to have the PostgreSQL software installed, and the database needs to be created and configured according to the specifications of the Data Design.
- **Web Server:** A server to host the Flask web server needs to be set up. This server needs to have Python installed, along with the Flask framework and jsonify function, as well as psycopg2 PostgreSQL adapter for Python and the pickle Python module for database interactions and data processing.
- **Jupyter Notebook Server / Dash application:** A server to host the Jupyter Notebook needs to be set up. This server needs to have Python and Jupyter

Notebook installed, along with any Python packages required by the dashboard, such as dash, plotly, pandas, numpy, requests, and datetime data visualization.

### 7.2. Configuration

Next, the application needs to be configured according to the environment. This involves setting the appropriate configuration parameters in the application, such as the database connection details in the web server and the API endpoint details in the dashboard.

Environment variables are often used for this purpose, as they allow the configuration to be changed without modifying the application code. They also keep sensitive information like database passwords out of the code.

### 7.3. Deployment

Finally, the application can be deployed. This involves copying the application code to the servers, starting the PostgreSQL service on the database server, starting the Flask service on the web server, and starting the Jupyter Notebook service and Dash app on the Jupyter Notebook server.

The specific steps to start the services depend on the environment and the way the application is structured. For example, the Flask service might be started by running a command like `flask run`, and the Jupyter Notebook service might be started by running a command like `jupyter notebook`.

### 7.4. Maintenance

After the system is deployed, it needs to be maintained. This involves monitoring the system to ensure it's working correctly, troubleshooting any issues that arise, updating the application as needed, and backing up the data regularly.

Overall, the deployment of the system involves careful planning and execution to ensure that the application runs smoothly in its intended environment. It's also important to have a plan for maintaining the system once it's deployed.

## 8. Testing and Validation

Testing and validation are crucial to ensure the functionality and reliability of the system. They involve checking that each component of the system works as expected, both independently and when integrated with the other components.

### 8.1. Unit Testing

Unit testing involves testing individual units of the code to ensure they work as expected. For example, each function in the web server code could be tested to ensure it returns the expected results for a given input.

In the case of the web server, unit tests might be written to test the functions that handle the database queries. For example, a unit test for the function that retrieves measurements from the database might check that it returns the correct measurements for a given `id`, `city_name` and `timestamp` range.

For the dashboard, unit tests could be written to test the data processing and visualization functions. For example, a unit test for a function that calculates the average AQI measurement for each sensor might check that it returns the correct averages for a given set of measurements.

### 8.2. Integration Testing

Integration testing involves testing the interactions between different components of the system. For example, the interaction between the database and the web server could be tested to ensure the web server can successfully query the database and handle the responses.

Similarly, the interaction between the web server and the dashboard could be tested to ensure the dashboard can successfully send requests to the web server and handle the responses.

### 8.3. System Testing

System testing involves testing the system as a whole to ensure it meets the specified requirements. This might involve testing the system with a range of different inputs and scenarios to ensure it behaves as expected in all cases. For instance, a system test might involve entering various queries into the dashboard, checking that the correct data is retrieved and displayed, and that the visualizations update correctly.

### 8.4. Validation

Validation involves checking that the system meets the needs of the users and fulfills its intended purpose. This might involve user testing, where colleagues serving as preliminary end-users, use the system and provide feedback.
It might also involve checking that the system handles any error conditions gracefully.

## 9. Maintenance

Maintenance is a crucial aspect of the application’s lifecycle to ensure its smooth operation and satisfaction to the user needs. Mainly, maintenance will focus on bug fixes and performance monitoring.

### 9.1. Bug Fixing

Despite thorough testing, it's likely that some bugs will emerge once the system is in use. These bugs should be logged, tracked, and fixed in a timely manner to ensure the system continues to function correctly and meet users' needs. Depending on the severity of the bug, this might involve immediate hotfix deployments or longer-term fixes scheduled in future updates.

### 9.2. Performance Monitoring

Continuously monitoring the system’s performance is necessary to ensure it remains efficient and can handle the load. This can involve monitoring server resources, database query performance, response times, and error rates. If performance issues are detected, they should be analyzed and addressed to keep the system running smoothly.

## 10. Appendix

*Documentation*. PostgreSQL. (n.d.). https://www.postgresql.org/docs/

Software Design Document - Washington University in St. Louis. (n.d.). https://classes.engineering.wustl.edu/ese497/images/9/96/2004Schalk_BCI2000Implementation.pdf

William. (2022). *Web Application Architecture: The Latest Guide 2022*. ClickIT. https://www.clickittech.com/devops/web-application-architecture/