POLITECNICO
MILANO 1863

Software Engineering for Geoinformatics

Design Document(DD)

Hananeh AsadiAghbolaghi

Hoda Sadat Mousavi Tabar

Sadra Zahed kachaee

Firoozeh Rahimian

Prof. Quattrocchi

July 202

Table of Contents

1. Introduction

1.1 Purpose

This Design Document outlines the technical design of the SE4GEO Air Quality Dashboard — a client-server system for ingesting, processing, and visualizing air quality data sourced from Dati Lombardia. It refines and translates the requirements described in the Requirement Analysis and Specification Document (RASD) into a detailed development plan.

The system is built around a PostgreSQL/PostGIS spatial database, a Flask-based RESTful API for data access, and a **Dash web application** that delivers interactive charts and maps for end users. The updated architecture replaces the initially proposed Jupyter Notebook interface with a more scalable and modular Dash application, allowing for richer web-based interactions and enhanced user experience.

The system supports both **raw (hourly)** and **aggregated (daily)** pollutant data visualization, empowering users — such as environmental agencies, researchers, and citizens — to monitor pollution levels, identify trends, and explore spatial patterns of air quality across Lombardy.

This document serves as a technical blueprint to guide the implementation team, ensuring all components align with stakeholder goals and performance expectations.

1.2 Scope

This Design Document (DD) defines the system architecture, data pipeline, REST API structure, and Dash-based web interface for the SE4GEO Air Quality Dashboard. It focuses on:

- **Ingestion and storage** of raw hourly pollutant measurements and sensor metadata in a PostgreSQL/PostGIS database.
- **Support for both raw and aggregated data**, with on-the-fly daily statistics calculated after ingestion (e.g., daily average, min, and max per pollutant and sensor).
- **REST API endpoints** developed with Flask to retrieve sensor metadata, raw measurements, and daily aggregates filtered by pollutant, sensor, and time range.
- **Interactive web visualizations** built using Plotly Dash, including:
    - A map view showing spatial pollutant distributions on selected dates.
    - A time-series panel with toggles between raw (hourly) and aggregated (daily) measurements.
- **Data normalization and spatial enrichment**, such as timestamp conversion to Europe/Rome timezone and station-region assignment via PostGIS.

- **Modular design principles** allowing independent updates to ingestion, processing, storage, and visualization modules.

1.3 References

- **Requirement Analysis and Specification Document (RASD)**, SE4GEO Project, 2025.
- **Dati Lombardia Datasets**:
  - *Air Quality Measurements*: https://www.dati.lombardia.it/Ambiente/Dati-sensori-aria-dal-2018/g2hp-ar79/about_data
  - *Sensor Metadata*: https://www.dati.lombardia.it/Ambiente/Stazioni-qualita-dell-aria/ib47-atvt/about_data
- **Technologies and Tools Used**:
  - PostgreSQL 15
  - PostGIS 3.3
  - Flask 3.x
  - Plotly Dash 2.x
  - Python 3.11
  - Visual Studio Code, Anaconda (for local development)
- **Scrum Handbook**, Polimi SE4GEO Course Material (WeBeep, 2025).

2.Overview

The SE4GEO system adopts a client-server architecture structured into three primary tiers: data storage, application logic, and presentation. It is composed of the following integrated components:

- **Database Layer**:

  A PostgreSQL database with PostGIS extension serves as the core data store. It maintains raw and aggregated air quality measurements, sensor metadata, geospatial coordinates, and precomputed statistics. It supports both relational and spatial queries efficiently.

- **Backend**:
  A Flask-based REST API acts as the interface between the database and the frontend dashboard. It exposes multiple endpoints to query sensor metadata, retrieve raw or aggregated pollutant data, and apply filters based on time, location, and pollutant type.
- **Frontend**:
  An interactive web dashboard built with Plotly Dash enables users to explore pollutant concentrations spatially and temporally. Features include:
  - Interactive map of sensor stations with pollutant overlays
  - Time-series graphs with raw and daily-aggregated data

- Date range filtering and pollutant type selection
- Responsive UI with options for data resolution (hourly/daily)

This architecture ensures modularity, scalability, and ease of future enhancements such as real-time streaming and machine learning integration.

2.1 Data Flow

1. **Ingestion**:
   Raw air quality measurements and sensor metadata from *Dati Lombardia* are preprocessed (including timestamp normalization and validation) and ingested into a PostgreSQL/PostGIS database. Both raw (raw_measurements) and aggregated (measurements) tables are maintained.
2. **Backend Processing**:

   The Flask REST API handles incoming HTTP requests from the frontend. Depending on the request parameters, it:

   - Queries raw or aggregated data from the appropriate database tables
   - Applies optional filters (e.g., pollutant type, sensor ID, date range)
   - Formats and returns results as JSON for visualization
3. **Frontend Visualization**:

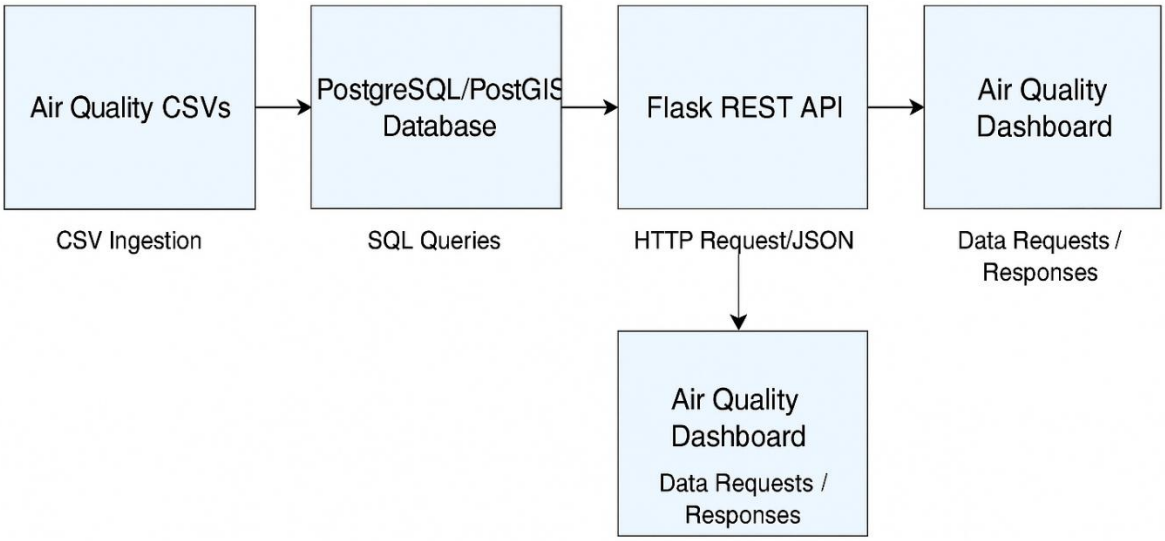
The Plotly Dash dashboard sends API requests based on user interactions (e.g., selected date, pollutant, sensor). It:

- Parses JSON responses into pandas DataFrames
- Generates interactive visualizations using Plotly (time-series, scatter maps)
- Offers toggles for selecting data resolution (hourly or daily) and dynamically updates figures based on user input

2.2 Architecture Diagram

The diagram illustrates the end-to-end data flow within the air quality monitoring system. Initially, air quality CSVs (including measurements and station metadata) are ingested into a PostgreSQL/PostGIS database. The Flask REST API layer then handles HTTP requests from the Air Quality Dashboard, querying the database and returning data in JSON format. The dashboard, implemented using Dash (built on top of Plotly and Flask), renders the results in the form of interactive maps and time-series charts. The architecture enables efficient querying, geospatial analysis, and responsive visualization for both real-time exploration and historical data analysis.

System Architecture



3. Database Design

3.1 Overview

The database is implemented using **PostgreSQL 17** with **PostGIS 3.4** to efficiently store, query, and spatially analyze air quality data derived from the Dati Lombardia datasets. It contains the following five primary tables:

- **sensors** – stores metadata for each air quality monitoring station, including unique ID, station name, location (with geographic coordinates), type, and municipality.
- **sensor_pollutants** – defines which pollutants each sensor is capable of measuring, supporting queries and filtering by sensor capability.
- **raw_measurements** – holds high-resolution pollutant measurements recorded hourly for each sensor and pollutant. Timestamps are normalized to the Europe/Rome timezone to ensure consistency.
- **measurements** – contains aggregated statistics (daily averages, maxima, minima) derived from the raw data to improve dashboard responsiveness.

3.2 Schema

Measurements Table

- This table contains daily aggregated pollutant values for each sensor.

Column Name	Data Type	Description
measurement_id	SERIAL (Primary Key)	Auto-incremented unique identifier for each daily measurement entry.
sensor_id	VARCHAR(50)	Foreign key referencing sensors(sensor_id), identifies the measurement source.
timestamp	DATE	The date of the aggregated measurement (e.g., 2023-01-01).
pollutant	VARCHAR(50)	Type of pollutant (e.g., PM2.5, NO2, O3).
daily_avg	DOUBLE PRECISION	Daily average concentration (μg/m³ or ppm depending on pollutant).
daily_min	DOUBLE PRECISION	Minimum recorded value in the day.
daily_max	DOUBLE PRECISION	Maximum recorded value in the day.

Raw Measurements Table

This table stores hourly pollutant measurements as directly ingested from CSV files, before aggregation.

Column Name	Data Type	Description
measurement_id	INTEGER (Primary Key)	Auto-incremented unique identifier for each raw measurement.
sensor_id	VARCHAR(50)	Foreign key referencing the sensors table, identifying the station.
timestamp	TIMESTAMP WITHOUT TIME ZONE	Timestamp of the measurement (e.g., "2018-01-01 01:00:00").
pollutant	VARCHAR(50)	Name of the measured pollutant (e.g., "Biossido di Azoto").
value	DOUBLE PRECISION	Raw pollutant concentration value in μg/m³ (e.g., 56.9).

Sensor_Pollutants Table

This table defines which pollutants are measured by each sensor station, supporting many-to-many relationships.

Column Name	Data Type	Description
sensor_id	VARCHAR(50) (Primary Key)	Unique identifier of the sensor station (e.g., "10001").
pollutant	VARCHAR(50) (Primary Key)	Name of the pollutant measured by the sensor (e.g., "Ossidi di Azoto").

Sensors Table

This table contains metadata for each air quality sensor station, including location and administrative attributes.

Column Name	Data Type	Description
sensor_id	VARCHAR(50) (Primary Key)	Unique identifier of the sensor (e.g., "10001").
station_name	VARCHAR(100) (Not Null)	Name of the monitoring station (e.g., "Cassano d'Adda 2-v.Milano").
province	VARCHAR(50)	Province abbreviation (e.g., "MI" for Milan).
latitude	DOUBLE PRECISION (Not Null)	Latitude coordinate of the station (e.g., 45.52647322).
longitude	DOUBLE PRECISION (Not Null)	Longitude coordinate of the station (e.g., 9.515991).
geom	GEOMETRY(Point, 4326)	PostGIS geometry field representing the point location for spatial queries.

3.3 Indexes

- **raw_measurements**:
  - Indexed on sensor_id and timestamp to allow fast filtering by sensor and time.
- **measurements**:
  - Indexed on sensor_id, timestamp, and pollutant to speed up queries on processed and aggregated data.
- **sensors**:

    - A spatial index is created on the geom column to support efficient geographic queries (e.g., finding nearby stations).
- **sensor_pollutants**:
    - Indexed on sensor_id and pollutant to quickly identify which pollutants are measured by each sensor.

3.4 Data Ingestion

Data is ingested from cleaned and merged CSV files containing both air quality measurements and sensor metadata. A Python script (manage_table.py) uses the psycopg2 library to connect to the PostgreSQL database and populate the following tables:

- **sensors**: The script first inserts unique sensor metadata (station name, coordinates, etc.) into the sensors table. It ensures no duplicate entries are inserted by checking the primary key sensor_id.
- **sensor_pollutants**: This table is populated next to establish the relationship between sensors and the pollutants they measure.
- **raw_measurements**: Then, hourly pollutant measurements are inserted into the raw_measurements table, including timestamps, values, and pollutant types. The script handles missing or invalid values and ensures all foreign key references are valid.
- **measurements**: After raw data is inserted, aggregated statistics (daily averages, maxima) are computed and stored in the measurements table to support efficient visualization.

This ingestion pipeline ensures data quality and referential integrity, enabling reliable downstream analysis.

3.5 Entity-Relationship Diagram

The ERD illustrates the relationships among the four core tables in the database: sensors, sensor_pollutants, raw_measurements, and measurements.

- The sensors table stores metadata for each sensor, including its geographic location and administrative region.
- The sensor_pollutants table models the many-to-many relationship between sensors and pollutants, using a composite primary key (sensor_id, pollutant).
- The raw_measurements table contains raw hourly pollutant readings from each sensor, with fields for pollutant type, timestamp, and value.
- The measurements table holds aggregated values (e.g., daily averages, minima, and maxima) for each sensor and pollutant.

Key relationships:

- raw_measurements.sensor_id and measurements.sensor_id are foreign keys referencing sensors.sensor_id.
- sensor_pollutants.sensor_id also references sensors.sensor_id, and sensor_pollutants.pollutant matches values in both measurement tables.

This structure ensures data integrity while supporting both detailed time-series analysis and efficient querying of aggregated trends

4.Overview

The REST API, built with Flask 3.x, exposes endpoints that query the PostgreSQL/PostGIS database and return structured, JSON-formatted data. It acts as the communication layer between the database and the Jupyter Notebook-based dashboard, enabling dynamic data exploration. The API supports retrieval of sensor metadata (including geospatial information), time-filtered air quality measurements, and pollutant-specific statistics aggregated by day.

4.1 Endpints

GET /api/sensors/<sensor_id>

Returns details for a specific sensor.

- **Parameters**:
  - sensor_id (string, required): Unique sensor ID
- **Response**: JSON object with sensor metadata
- **Example**:


```
{
  "sensor_id": "10001",
  "station_name": "Cassano d'Adda 2-v.Milano",
  "province": "MI",
  "latitude": 45.52647322,
  "longitude": 9.515991,
  "geometry": { "type": "Point", "coordinates": [9.515991, 45.52647322] }
}
```

GET /api/measurements

Returns daily aggregated measurement values, optionally filtered.

- **Query Parameters**:
  - sensor_id (optional): Sensor identifier
  - pollutant (optional): Pollutant name (e.g., "PM2.5")
  - start (optional): Start date (YYYY-MM-DD)
  - end (optional): End date (YYYY-MM-DD)
- **Response**: JSON array of measurement objects
- **Example**:


```
[
  {
    "measurement_id": 1,
    "sensor_id": "10001",
    "date": "2023-01-01",
    "pollutant": "PM2.5",
    "daily_avg": 24.5,
    "daily_min": 18.2,
    "daily_max": 30.1
  },
]
```


GET /api/sensors/<sensor_id>/measurements

Returns all measurements for a specific sensor, ordered by date.

- **Parameters**:
  - sensor_id (string, required)
- **Response**: JSON array of measurements
- **Example**:


```
[
  {
    "measurement_id": 1234,
```

```
    "date": "2023-01-01",
    "pollutant": "PM10",
    "daily_avg": 42.1,
    "daily_min": 38.0,
    "daily_max": 47.5
  },
]
```


4.3 Implementation

- The API is defined in app.py.
- SQL queries are executed using psycopg2.
- Sensor geometries are returned in GeoJSON format using ST_AsGeoJSON.
- Aggregated measurements include daily averages, minima, and maxima.
- Data cleaning (e.g., NULL/negative filtering) is handled before ingestion, so the API assumes clean data.

4.4 Security Considerations

- Parameterized queries are used to prevent SQL injection.
- Input validation is performed for query parameters.

5. Dashboard Design

5.1 Overview

The dashboard is developed using **Dash** by Plotly and runs as a standalone web application. It provides an interactive interface for exploring air quality data served by a Flask REST API. The dashboard integrates **Plotly** for dynamic charts and **Mapbox** for geospatial visualization. It enables users to analyze pollutant trends, sensor behavior, and regional pollution patterns across time and space.

5.2 Features

- **Interactive Map Panel:**
    - Displays sensor locations on a map using **Plotly Mapbox**.

    - Markers are color-coded and sized based on pollutant concentration (daily_avg) for a selected day.
    - Hover tooltips display the station name and pollutant values.
- **Time-Series Panel:**
    - Visualizes pollutant concentration trends for a selected sensor over a specified date range.
    - Users can switch between **raw hourly measurements** and **daily aggregates** using a resolution selector.
    - Interactive features include zooming, panning, and hover tooltips.
- **Filtering Options:**
    - Dropdowns allow selection of:
        - Pollutant type (e.g., PM10, NO_2, O_3)
        - Sensor station (by station name)
        - Date or date range (with a calendar widget)
        - Data resolution (hourly or daily)
- **Automatic Data Retrieval:**
    - Pollutants, date bounds, and sensor metadata are dynamically fetched from the API at startup.

5.3 Data Flow

- The dashboard uses the requests library to fetch data from the Flask API.
- Endpoint calls include:
    - /api/sensors – Retrieves metadata and coordinates for all sensors.
    - /api/measurements – Retrieves daily aggregated pollutant data based on user-specified filters.
    - /api/raw_measurements – Retrieves hourly measurements for time-series plots.
- API responses are parsed into Pandas DataFrames for filtering, merging, and plotting.
- Data is visualized using Plotly's scatter_mapbox and line plots depending on the panel.

6. Implementation Considerations

6.1 Tools and Technologies

- **Database**: PostgreSQL 17 with PostGIS 3.4 for geospatial support.
- **Web API**: Flask 3.x REST API serving JSON endpoints.
- **Dashboard**: Dash (by Plotly) using Python 3.8+, with dash, pandas, requests, and plotly.express for interactive web-based visualization.

- **Development Environment**: Visual Studio Code with Git for version control and collaborative development.

6.2 Constraints

- The project is limited to publicly available **Dati Lombardia** datasets.
- Data ingestion is performed **once** during setup using a preprocessing script (manage_table.py). Continuous real-time ingestion is out of scope.
- Project deadline (July 4, 2025) constrains additional features such as user authentication or province-level aggregation endpoints.

6.3 Assumptions

- Sensor IDs are consistently formatted and aligned between measurement files and metadata tables.
- Users can run the Dash app locally via terminal with Python installed (no Jupyter Notebook interface is used).
- Target systems are assumed to support PostgreSQL and Flask (recommended: minimum 8 GB RAM, dual-core CPU).

6.4 Performance and Scalability

- Proper indexing on sensor_id and timestamp ensures measurement queries return within 1–2 seconds, even for full-year datasets.
- The API is stateless and capable of serving up to **10 concurrent users** under typical conditions.
- Dashboard visualizations (both map and time-series panels) render in under **5 seconds** for common queries, thanks to lightweight front-end design and optimized API responses.

7. Testing Strategy

7.1 Unit Testing

- Use pytest to verify **Flask API endpoints** return correct status codes and valid JSON structures.
    - Example: GET /api/sensors should return a list of sensors with fields sensor_id, station_name, geometry, etc.
- Test data filtering logic on the /api/measurements endpoint to ensure parameters like pollutant, sensor_id, and date ranges return correct subsets.
- Validate date formatting, type conversions, and error responses (e.g., 404 for non-existent sensor).

7.2 Integration Testing

- Confirm that Flask API connects correctly to the **PostgreSQL/PostGIS** database and retrieves accurate results.
- Test that the **Dash dashboard** successfully pulls data from API endpoints using requests and updates:
    - **Map visualizations** with selected pollutant and date.
    - **Time-series plots** for the correct sensor, resolution, and time range.
- Ensure the data flow from the database → API → dashboard functions seamlessly for real-time user interaction.

7.3 User Testing

- Team members simulate roles such as environmental analysts or decision-makers to explore:
    - Sensor selection
    - Pollutant filtering
    - Map-based hotspot identification
- Validate that:
    - All widgets (Dropdowns, DatePickers) behave as expected.
    - Charts display correct pollutant values and trends (e.g., PM2.5 changes over time).
    - Visual consistency exists between raw and aggregated views when switching resolution.
- Manual checks are performed to verify that the dashboard matches expected values from the source datasets (raw_measurements.csv, etc.).

8. Conclusion

- This Design Document presents a complete and detailed blueprint for the development of the Air Quality Dashboard system. The architecture integrates a robust PostgreSQL/PostGIS database for storing and spatially querying both raw and aggregated air quality data, enriched with metadata from sensor stations.
- The Flask-based REST API forms the communication bridge between the data layer and the front-end, offering structured and filterable access to measurements, metadata, and statistical summaries. It ensures clean, validated, and structured data delivery to the visualization layer.
- The user interface is implemented using Dash (Plotly), replacing the earlier Jupyter Notebook design to offer a more responsive, web-based dashboard. The dashboard provides interactive visualizations—including pollutant maps and time-series charts—with flexible filtering by pollutant, station, date, and resolution (raw or aggregated). This setup

allows users to explore air quality patterns, identify hotspots, and assess pollutant trends effectively.

- The entire design aligns with the functional and non-functional requirements specified in the RASD, supporting data ingestion, preprocessing, spatial mapping, and user-friendly visualization. It provides environmental analysts, decision-makers, and the public with a reliable tool for real-time air quality monitoring and policy planning in the Lombardy region.