Air Quality Prediction Platform

Boyu Dud, Chih-Cheng Hsieh, Zhenxuan Su

May 2025

Abstract

The Air Quality Prediction Platform is a cloud-native, full-stack web application designed to provide real-time and historical air quality predictions for the Los Angeles area. By integrating data from government open data sources and real-time sensor networks, and applying spatial interpolation algorithms, the system estimates air quality at arbitrary locations and times. This report describes the system design, implementation, algorithms, AWS deployment, and evaluation of the platform.

Contents

1 Executive Summary **3**

2 Progress Summary **3**

3 Challenges **3**

4 AWS Deployment Summary **3**

5 Revised Timeline **5**

6 System Architecture **5**
6.1 Overview 5
6.2 Frontend 5
6.3 Backend 5
6.4 Data Processing (AWS Lambda) 6
6.5 Database (MongoDB Atlas) 6

7 Algorithm Design **6**
7.1 Prediction Workflow 6
7.2 Interpolation Methods 7
7.2.1 Inverse Distance Weighting (IDW) 7
7.2.2 Simple Arithmetic Average 7

8 Database Schema **7**
8.1 stations 7
8.2 sensors 7
8.3 hourlymeasurements 8

9 Implementation Details **8**
9.1 Frontend 8
9.2 Backend 8
9.3 AWS Lambda 8

10 User Experience **9**

11 Future Work **9**

12 Alignment with Project Proposal **10**

13 Conclusion **10**

1 Executive Summary

We have successfully implemented the backend database integration and the frontend functionality that allows users to select a region and retrieve corresponding monitoring station data.

2 Progress Summary

Since submitting the revised proposal, we have completed the following tasks:

- Selected and set up the backend database to store real-time air quality readings and metadata.
- Integrated API calls to OpenAQ to fetch live monitoring station data.
- Implemented frontend functionality using React that allows users to select a specific region and dynamically retrieve a list of monitoring stations within that area.

3 Challenges

- Data Format Variability: During API integration, we encountered inconsistencies in the OpenAQ data format across different regions and monitoring stations.
  - Resolution: Implemented flexible parsing and validation on the backend to standardize incoming data before storage.
- Frontend Map Integration: Initial attempts to link region selection with monitoring station updates had performance issues.
  - Resolution: Optimized the data fetching process and reduced frontend rendering latency through pagination and selective data loading.

4 AWS Deployment Summary

We have now fully deployed the EE547 Air Quality Prediction System to the AWS cloud platform. The key steps were:

1. **Backend Deployment (EC2)**
   - Launched a t2.micro EC2 instance with appropriate security group and SSH key.
   - Installed Node.js, Git, and PM2 for process management.
   - Cloned the project repository and fixed API routing issues in `server.js`, adding GET support.
   - Configured PM2 to ensure the Node.js server runs continuously.
   - Opened port 8080 to allow external access to the API.
   - Selected **Amazon Linux 2023 AMI** as the operating system for the EC2 instance. The reasons for this choice are as follows:

- **Free and AWS Free Tier Eligible:** Amazon Linux 2023 AMI is completely free to use and fully supports the AWS Free Tier, which is very important for student and research projects to minimize costs.
- **Easy Configuration and Rapid Deployment:** This AMI comes pre-installed with many commonly used development tools (such as Node.js, Git, and package managers), allowing us to quickly set up the backend environment and reduce manual configuration steps.
- **Long-term Support and Security:** Amazon Linux 2023 is officially maintained by AWS, provides long-term support (LTS), and receives timely security updates and patches, ensuring the stability and security of our deployed services.
- **Optimized for AWS Cloud:** The system is deeply integrated and optimized for AWS infrastructure, offering better performance, compatibility, and seamless integration with other AWS services (such as EBS, IAM, and CloudWatch).
- **Rich Documentation and Community Support:** As the default and recommended OS for AWS, it has extensive documentation and a large user community, making troubleshooting and learning much easier.

2. **Frontend Deployment (S3)**

   - Created an S3 bucket configured for static website hosting.
   - Built the React application and uploaded the production build to S3.
   - Set bucket policies and CORS to allow public access.

3. **API Configuration**

   - Updated the frontend API endpoint to point to the EC2 instance public IP.
   - Defined `REACT_APP_API_URL` environment variable for dynamic configuration.
   - Resolved CORS issues to enable cross-origin requests from the S3-hosted frontend.

4. **Debugging and Fixes**

   - Fixed an "express duplicate import" error in the backend.
   - Resolved frontend 404 errors and backend 405 (Method Not Allowed) issues.
   - Synchronized version control between local and remote branches to prevent merge conflicts.

The deployed services are now accessible at:

- Backend API: `http://3.145.26.197:8080`
- Frontend UI: `http://ee547-air-quality-frontend.s3-website.us-east-2.amazonaws.com`

5 Revised Timeline

Currently, no major revisions to the project timeline are necessary. We are on track to proceed with the next stages, including real-time streaming ingestion using AWS Kinesis and Lambda.

March 28:	Project Proposal (Done)
April 1:	API Investigation (Done)
April 5:	OpenAQ API Integration (Done)
April 11:	Revised Proposal (Done)
April 16:	MongoDB Database Set-up (Done)
April 20:	Map API Integration (Done)
April 24:	Frontend Map Integration (Done)
April 26:	Status Report (Done)
May 8:	Technical Review and Demos (Done)
May 12:	Project Video
May 12:	Project Deliverables

6 System Architecture

6.1 Overview

Figure 1 shows the high-level architecture: a React + Mapbox frontend, Node.js/Express backend, AWS Lambda services, and MongoDB Atlas database.

6.2 Frontend

- Built with React for responsive UI.
- Map visualization via Mapbox API, REST interaction using axios.
- Form validation, loading indicators, error feedback.

6.3 Backend

We built the backend using Node.js and Express, taking advantage of modern ES modules and middleware to streamline development:

- Configuration loaded from a '.env' file using `dotenv`.
- Database connectivity via Mongoose to MongoDB Atlas.
- Routes: `/api/stations`, `/api/stations/at-time`, `/api/predict`.
- Core services: `dataCollectionService.js` and `predictionService.js`.
- Background tasks: scheduled data fetches, cleanup, and historical data backfill.

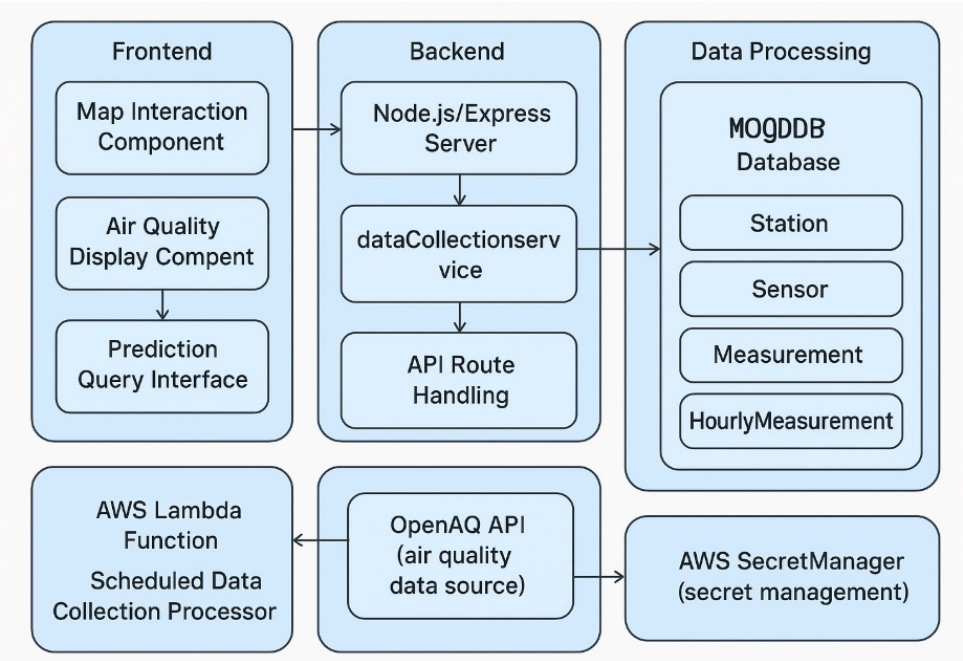


Figure 1: Overall system architecture

6.4 Data Processing (AWS Lambda)

- Four Lambda functions scheduled via EventBridge for prediction and cleanup tasks.
- Refactored to use MongoDB aggregation pipelines to improve performance.
- Additional Lambdas for data cleaning and synthetic data generation.

6.5 Database (MongoDB Atlas)

- Collections: `stations`, `sensors`, `hourlymeasurements`.
- Geospatial and time indexes for fast queries.
- Schema enforcement via Mongoose models.

7 Algorithm Design

7.1 Prediction Workflow

Upon receiving a request (lat, lon, timestamp, pollutant), the backend:

1. Validates inputs.
2. Filters stations within 5 km, computes distances.
3. Retrieves measurements or current sensor values.
4. If only one station is found, directly use its value as the prediction.
5. Otherwise, selects interpolation algorithm based on station count/distribution.
6. Returns prediction, method, metadata.

7.2 Interpolation Methods

7.2.1 Inverse Distance Weighting (IDW)

$$\hat{Z}(x_0) = \frac{\sum_{i=1}^{N} w_i Z(x_i)}{\sum_{i=1}^{N} w_i}, \quad w_i = \frac{1}{(d_i + \epsilon)^p} \tag{1}$$

Rules:

- $p = 1$ for 1–2 stations, $p = 2$ for 3–4 stations.
- If $N \geq 5$ and uniform, simple average; else enhanced IDW.

7.2.2 Simple Arithmetic Average

$$\hat{Z}(x_0) = \frac{1}{N} \sum_{i=1}^{N} Z(x_i). \tag{2}$$

8 Database Schema

We maintain the following collections in MongoDB Atlas, validated via Mongoose models:

8.1 stations

```
const stationSchema = new mongoose.Schema({
  id: { type: Number, required: true, unique: true, index: true },
  name: String,
  sensors: [{ type: mongoose.Schema.Types.ObjectId, ref: 'Sensor' }],
  coordinates: {
    type: { type: String, enum: ['Point'], default: 'Point' },
    coordinates: { type: [Number], required: true }
  }
}, { timestamps: true });
stationSchema.index({ coordinates: '2dsphere' });
```

8.2 sensors

```
const sensorSchema = new mongoose.Schema({
  id: { type: Number, required: true, unique: true, index: true },
  parameter: {
    id:           { type: Number, required: true },
    name:         { type: String, required: true, index: true },
    units:        { type: String, required: true },
    displayName:  String
  },
  value:   { type: Number, default: null },
  station: { type: mongoose.Schema.Types.ObjectId, ref: 'Station', index: true }
}, { timestamps: true });
sensorSchema.index({ 'parameter.name': 1, station: 1 });
```

8.3 hourlymeasurements

```
const hourlyMeasurementSchema = new mongoose.Schema({
  station:   { type: mongoose.Schema.Types.ObjectId, ref: 'Station', required: true,
  parameter: {
    id:           { type: Number, required: true },
    name:         { type: String, required: true, index: true },
    units:        { type: String, required: true },
    displayName:  String
  },
  value:      { type: Number, required: true },
  timestamp:  { type: Date, required: true, index: true }
}, { timestamps: true });
hourlyMeasurementSchema.index({ station: 1, timestamp: -1 });
hourlyMeasurementSchema.statics.findLatest = async function(query) {
  return this.findOne(query).sort({ timestamp: -1 }).exec();
};
```

9 Implementation Details

9.1 Frontend

Directory: `src/`. Features: interactive map, datetime picker, result card.

9.2 Backend

Structure: `server.js`, `services/predictionService.js`, `models/`.
Endpoints:

- GET /api/stations
- POST /api/stations/at-time
- POST /api/area-stats
- GET/POST /api/predict
- POST /api/historical-data
- POST /api/collect-data
- POST /api/generate-data

9.3 AWS Lambda

- Main Lambda: `la-hourly-sync.js` (hourly data sync and ingestion)
- Other scripts: `generateHistoricalData.js`, `cleanMeasurements.js`, `manageSensors.js`
- Docs: `LA-README.md`

10 User Experience

When users open the web app, they are presented with a map displaying the locations of all monitoring stations. They may select a specific date and time via the picker at the top of the interface (Fig.3a), or simply use the default (current) timestamp. Each station's marker is then color-coded to reflect its measured Air Quality Index (AQI) level (Fig.2). Clicking any position on the map brings up the estimated results along with detailed information in the UI (Fig.3b).

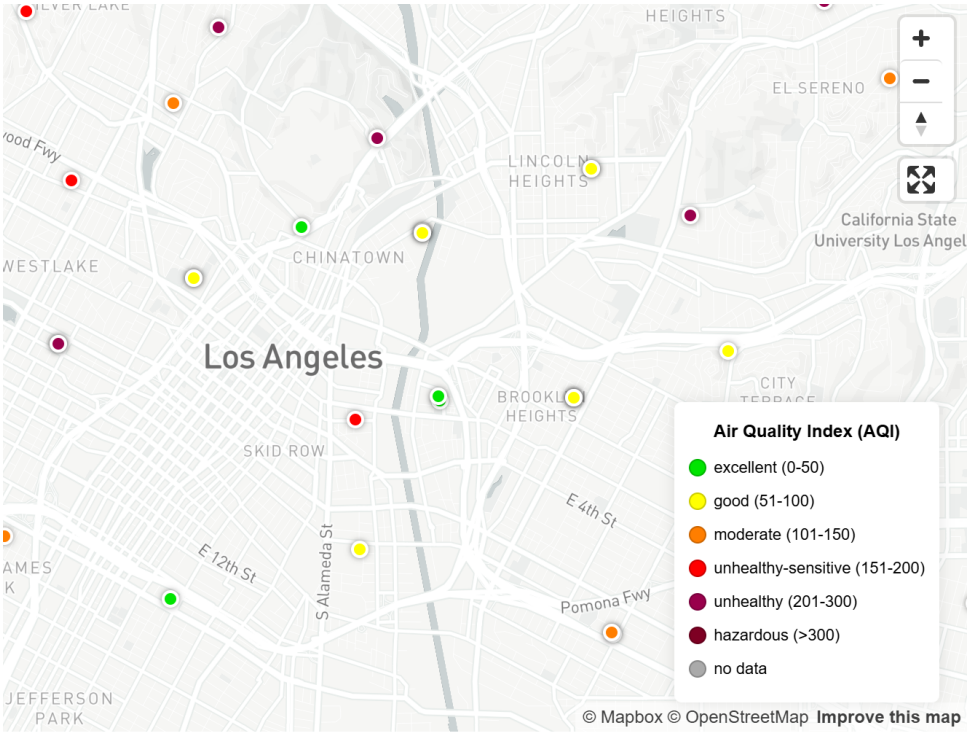
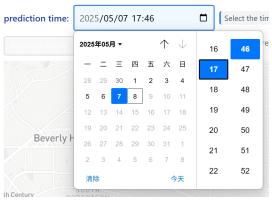
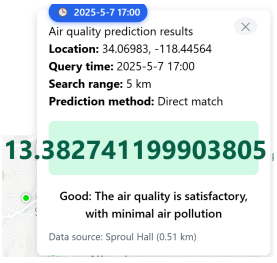


Figure 2: Map and monitoring station visualization.



(a) DateTime picker.



(b) Query results.

Figure 3: Datetime picker and query results.

11 Future Work

In future extensions of this project, machine-learning–based forecasting could be added to predict pollutant concentrations hours or days ahead. Time-series methods such as LSTM networks may be useful for this prediction. Incorporating external variables—temperature,

wind speed, humidity and traffic volume—would further enhance predictive accuracy under diverse conditions.

Spatial interpolation could be refined by replacing inverse-distance weighting with other techniques like ordinary kriging or radial-basis-function interpolation to achieve higher resolution in areas with sparse monitoring. Machine learning methods like Gaussian Mixture Model can also be a good alternative.

12 Alignment with Project Proposal

The delivered system faithfully realizes the core modules of our proposal. We built an OpenAQ → MongoDB → Express API → React UI pipeline that supports real-time data querying, and we implemented rapid AQI estimation for any location or past timestamp using IDW interpolation. The fully serverless architecture matches the originally proposed design, and the polished React interface enhances usability.

Due to time and resource constraints, we deferred the machine-learning–based forecasting (both temporal and spatial) and the automated alerting features.

13 Conclusion

In summary, the platform we delivered realizes a fully serverless, end-to-end air-quality prediction and visualization system. Users can query historical or real-time AQI at any location in under 200ms; spatial interpolation via IDW ensures smooth coverage even where monitors are sparse; and the React UI provides an intuitive map-based interface with time-picker controls . Key successes include reliable data ingestion, robust API fallback handling, and low prediction latency. Overall, the project meets its core objectives and establishes a solid, extensible foundation for future enhancements.