Workshop 3 - Project

Air Quality Analysis Platform

Members:

Jose Alejando Cortazar – Cod.: 20181020022
Stivel Pinilla Puerta – Cod.: 20191020024
Johan Castaño Martinez – Cod.: 20191020029

Professor:

Carlos Andrés Sierra

Universidad Distrital Francisco José de Caldas
School of Engineering
Computer Engineering Program

Databases II

September 2025, Bogotá D.C.

Contents

1 Introduction

The project consists of a web-based platform that collects, processes, and presents air-quality information using relational and scalable data-management techniques. Its primary motivation is to provide reliable, uniform, and accessible environmental data that can support public decision-making, research activities, and citizen awareness. Air pollution directly affects health, mobility, and long-term urban planning, making timely and well-structured information essential for institutions and the general population.

The goal of the platform is to integrate heterogeneous external data sources, transform them into a consistent operational dataset, and expose processed information through dashboards, historical queries, reports, and alerts. The scope of this project is strictly limited to the data-management layer: data ingestion, storage, normalization, querying, and visualization of core indicators. Advanced features such as predictive analytics, recommendation engines, mobile applications, and geographic exploration are considered out of scope for the current deliverable and are not included unless required for demonstrating database concepts.

The system focuses on demonstrating ingestion pipelines, a hybrid storage model (SQL + structured raw data), performance-oriented query design, and a basic user interaction layer for consuming processed information. This allows the project to remain aligned with the learning objectives of the course while providing a realistic foundation for future extensions.

2 Business Model

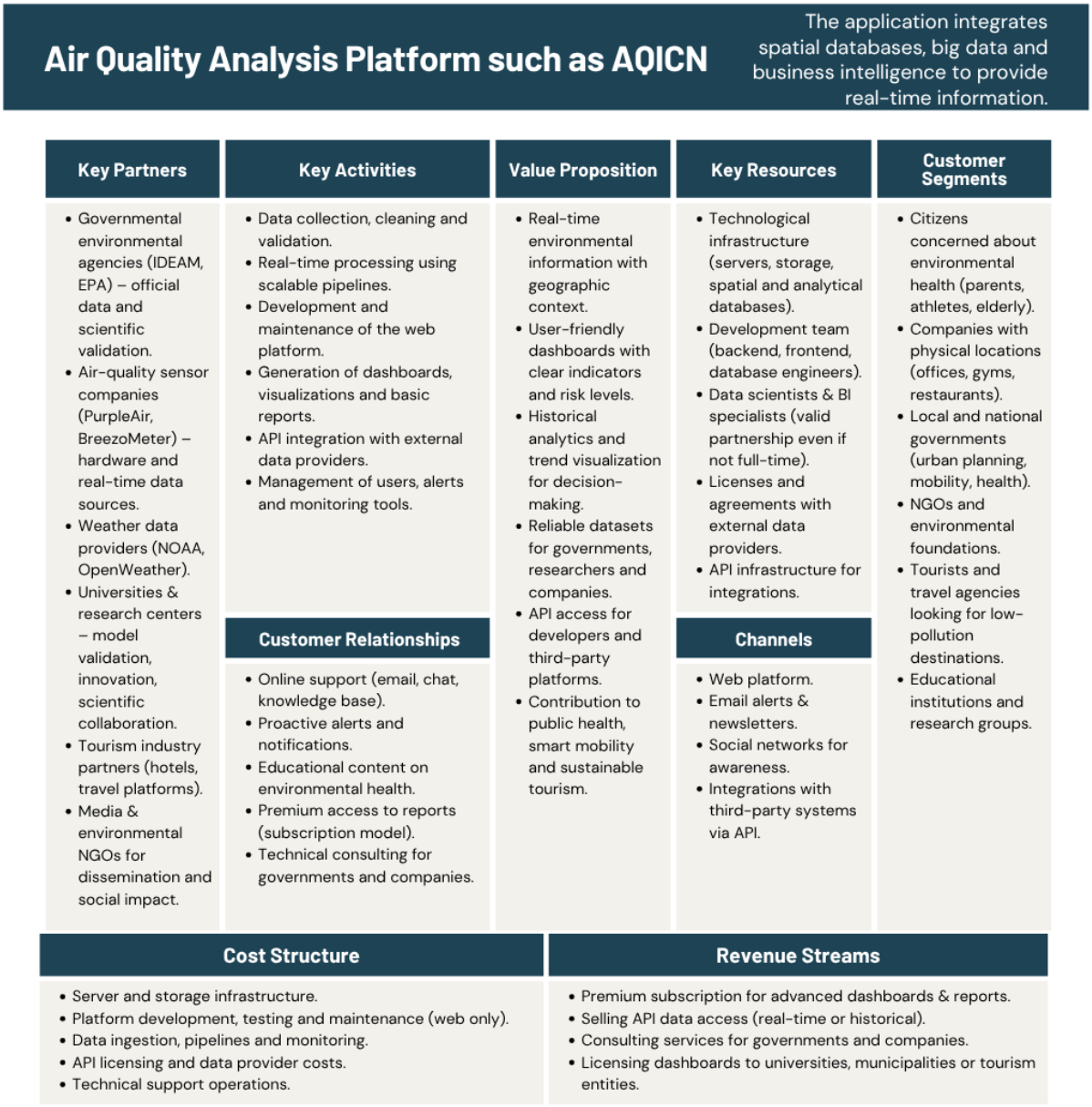

Figure 1: Business Model

How Does It Work?

The platform collects environmental, meteorological and geospatial data from official sources and third-party APIs. The information is cleaned, processed and stored in a hybrid architecture using relational and analytical databases. Users access the results through real-time dashboards, historical analyses, downloadable reports and interactive spatial visualizations.

Who Uses the Platform?

The system serves citizens monitoring exposure levels, government agencies issuing alerts or designing policies, companies and NGOs evaluating environmental impact, and academic institutions that require reliable datasets for analysis or research.

How Does It Generate Revenue?

Monetization focuses on premium access to advanced dashboards and reports, API data services, consulting for institutions and licensing analytical dashboards to organizations such as universities or municipalities.

What Technologies Does It Use?

The platform relies on a PostgreSQL-based relational database and a simple, scalable web architecture. This baseline implementation can be expanded in future iterations with partitioning, TimescaleDB extensions for time-series optimization, or business intelligence tools as data volume and user needs grow.

Air Quality Analysis Platform inspired by systems such as AQICN. The application integrates a relational database, scalable data pipelines and an API layer to provide frequent environmental information updates.

3 Requirements

ID	Type	Requirement	Associated User Stories
FR1	Functional	The system must periodically collect up-to-date air quality data from multiple external sources (e.g., APIs and monitoring stations) and store it for further processing.	US1, US13, US14
FR2	Functional	The system must allow users to query and visualize historical air quality data filtered by date range, location, and pollutant type.	US2, US7
FR3	Functional	The system must display air quality information in a consistent and clear format, independently of the original data source.	US1, US3
FR4	Functional	The system must display dashboards with key air quality indicators (e.g., AQI, main pollutant, daily trends) based on the latest available data.	US4
FR5	Functional	The system must generate custom reports with filters for date range, location, and selected indicators, and allow users to download these reports in standard formats.	US5
FR6	Functional	The system must present graphs showing the evolution of air quality over time, allowing users to interact with basic controls such as selecting date ranges or toggling pollutants.	US6
FR7	Functional	The system must allow users to export historical air quality data in standard formats such as CSV and JSON, based on the filters applied in the user interface.	US7
FR8	Functional	The system must provide simple, rule-based recommendations based on the user's location and current air quality conditions (e.g., avoiding outdoor exercise, using a mask).	US8
FR9	Functional	The system must send air quality alerts when configurable thresholds are exceeded, according to each user's notification preferences.	US9
FR10	Functional	*(Optional)* The system may suggest appropriate protective measures and product categories (e.g., certified masks, air purifiers) during high pollution periods, as informational guidance only.	US10
FR12	Functional	The system must support geographic search for air quality data by country, city, or region.	US15

ID	Type	Requirement	Associated User Stories
FR13	Functional	The system must provide a responsive web interface that works correctly on mobile, tablet, and desktop devices.	US16
FR14	Functional	*(Optional)* The system may allow users to share links to selected air quality views or reports on social media platforms, using simple shareable URLs.	US17
NFR1	Non-Functional	Queries on air quality data (for typical usage scenarios and expected data volumes) must execute in under 2 seconds for at least 95% of requests, under normal load conditions.	US3, US12
NFR2	Non-Functional	The system must support automated periodic ingestion of air quality data without manual intervention, according to a configurable schedule (e.g., every few minutes or hourly), instead of requiring continuous streaming 24/7.	US1, US14
NFR3	Non-Functional	Data storage must be dimensioned, indexed, and organized to handle the expected volume of readings and users over the project's time horizon, and to allow future partitioning if needed.	US1, US3
NFR4	Non-Functional	Customized reports with filters on date, location, and indicators must be generated in under 10 seconds for typical workloads.	US5
NFR5	Non-Functional	The recommendation logic (for simple rule-based recommendations) should be updated at least every 10 minutes, aligned with the update frequency of the external air quality APIs used by the system.	US8, US10
NFR6	Non-Functional	Air quality data views and visualizations must load in less than 2 seconds for at least 95% of user requests, under normal conditions.	US12
NFR7	Non-Functional	The system architecture must provide basic fault tolerance, ensuring that no permanent data loss occurs in case of a single node or instance failure, using regular backups and recovery procedures. Geographic redundancy across multiple regions is explicitly out of scope for the course project.	US14

ID	Type	Requirement	Associated User Stories
NFR8	Non-Functional	The system should be designed so that it can scale horizontally in the future (e.g., by adding read replicas or sharding), but a multi-node deployment is not required for the baseline course implementation (considered a future scalability goal).	US13, US14
NFR9	Non-Functional	The system's web user interface must function correctly on all major browsers and operating systems commonly used by the target audience, without critical errors.	US16
NFR10	Non-Functional	Data consistency and user personalization (e.g., preferences and dashboard configuration) must be preserved across all user devices, assuming that users log in with the same account.	US16

Table 1: Requirements

Non-Functional Requirements Justification

NFR2 (Automated periodic ingestion): Air quality monitoring requires regular automated data collection since pollution levels fluctuate throughout the day. Periodic ingestion (e.g., every few minutes or hourly) aligns with the update frequency of external APIs while maintaining data freshness for timely detection of hazardous conditions.

NFR4 (10 seconds report generation): Complex reports with multiple filters and large datasets require processing time, but 10 seconds maintains user engagement while allowing for comprehensive data analysis.

NFR5 (10 minutes update frequency): Air quality APIs from external sources typically update their information every 10 minutes to 1 hour, making more frequent updates unnecessary and resource-intensive.

NFR6 (Page and visualization load time): Critical safety information like air quality alerts must be delivered quickly through responsive UI loading to enable timely user decisions about outdoor activities and health precautions.

NFR7 (Basic fault tolerance): While multi-region geographic redundancy is beyond the scope of this course project, basic fault tolerance through regular backups and recovery procedures ensures data integrity and system reliability at a reasonable level for an academic implementation.

Future Functional Requirements

The following functional requirements are considered beyond the baseline scope of this course and may be implemented in future phases:

- **FR11:** In a future phase, the system may provide a basic map-based visualization that highlights areas with better or worse air quality using simple overlays on a web map. (Associated User Stories: US11)

4 User Stories

User stories describe the functional needs of the platform from the perspective of different stakeholder roles. Each story follows the standard *As a / I want to / So that* structure, and includes priority, effort estimates, and explicit acceptance criteria. The stories are aligned with the refined functional and non-functional requirements presented in the previous section, focusing on the realistic baseline scope of the project.

US1 – Automated Data Ingestion

As a	Technical Administrator
I want to	Collect up-to-date air quality data from external providers in an automated way
So that	The platform can provide accurate and current information on air quality without manual imports
Priority	Must
Effort	8
Acceptance criteria	Ingestion job runs on a configurable schedule (e.g., every 10-60 minutes); At least one external provider is ingested without manual intervention; Failed ingestions are logged with error details; Successful runs are logged with timestamps; Ingested readings appear in the database within the expected delay after the external API update

Table 2: User story US1

US2 – Filterable Historical Data Access

As a	Researcher/Analyst
I want to	Access historical air quality data filtered by city, date range, and pollutant
So that	I can perform longitudinal analysis and scientific research
Priority	Must
Effort	5
Acceptance criteria	User can select city, date range, and pollutant from the interface; System returns matching records from historical data; Results can be exported to CSV and JSON; At least 3 years of historical data are available for test cities

Table 3: User story US2

US3 – Efficient Analytical Queries

As a	Technical Administrator
I want to	Run queries over large volumes of air quality data without significant delays
So that	I can support analysts and dashboards without performance bottlenecks
Priority	Should
Effort	5
Acceptance criteria	Typical dashboard queries over AirQualityDailyStats complete in under 2 seconds for test datasets; Historical queries over several months complete in under 5 seconds; Database indexes for common filters (station, date, pollutant) are documented and enabled

Table 4: User story US3

US4 – Dashboards with Key Indicators

As a	Public Policy Manager
I want to	View dashboards with key air quality indicators for selected regions
So that	I can quickly understand current conditions and recent trends to inform decisions
Priority	Must
Effort	8
Acceptance criteria	Dashboard shows at least: current AQI, main pollutant, and daily trend for the selected city or station; Dashboard updates when the user changes city or station; Dashboard uses AirQualityDailyStats for historical trends and raw readings for current values

Table 5: User story US4

US5 – Custom Report Generation

As a	Researcher/Analyst
I want to	Generate custom reports with filters and download them
So that	I can use the data in external tools or include it in my own analyses
Priority	Must
Effort	5
Acceptance criteria	User can configure a report by choosing city/region, date range, and pollutants; System generates the report and confirms when it is ready; User can download the report in at least CSV format

Table 6: User story US5

US6 – Time-Series Graphs

As a	Citizen
I want to	See simple graphs of how air quality changes over time in my city
So that	I can understand whether conditions are improving or getting worse
Priority	Must
Effort	3
Acceptance criteria	User can select a city and a pollutant; System displays a time-series chart for the selected period; User can change the date range (e.g., last 7 days vs last 30 days) and the chart updates accordingly

Table 7: User story US6

US7 – Simple Rule-Based Recommendations

As a	Citizen
I want to	Receive simple recommendations based on current air quality at my location
So that	I can protect my health when air quality is poor
Priority	Should
Effort	5
Acceptance criteria	User can set a default city or location; When AQI exceeds defined thresholds, the system shows recommendations such as avoiding outdoor exercise or using a mask; Recommendations are based on simple, documented rules (no complex machine learning required)

Table 8: User story US7

US8 – Configurable Alerts

As a	Citizen
I want to	Configure alerts when air quality exceeds a certain threshold
So that	I can be notified when conditions become unhealthy
Priority	Must
Effort	5
Acceptance criteria	User can create an alert selecting city or station, pollutant, and AQI threshold; User can choose at least one notification channel (e.g., email or in-app notification); When AQI exceeds the configured threshold, an alert is recorded and shown; User can deactivate or delete existing alerts

Table 9: User story US8

US9 – Informational Protective Measures (Optional)

As a	Citizen
I want to	See informational suggestions about protective measures during high pollution episodes
So that	I can decide whether to use masks, air purifiers, or other measures
Priority	Could
Effort	3
Acceptance criteria	When AQI is above a defined level, the interface shows text with recommended protective measures; Suggestions are informational only and do not require e-commerce links

Table 10: User story US9

US10 – Geographic Search

As a	Citizen
I want to	Search air quality by country, city, or region
So that	I can compare air quality in different places
Priority	Must
Effort	3
Acceptance criteria	User can search by country and see available cities or stations; User can search directly by city name and open its dashboard; If regions are defined, the user can filter stations by region

Table 11: User story US10

US11 – Responsive Web Interface

As a	Citizen
I want to	Access the platform from different devices (mobile, tablet, desktop)
So that	I can check air quality whenever I need it, from any device
Priority	Must
Effort	5
Acceptance criteria	Core views (home, dashboard, alerts) are usable on mobile, tablet, and desktop browsers; Layout adapts without breaking text or hiding essential information; No native mobile app is required, the responsive web app is sufficient

Table 12: User story US11

US12 – Sharing Views (Optional)

As a	Citizen
I want to	Share air quality views with other people through social media or messaging
So that	I can raise awareness about air quality conditions
Priority	Could
Effort	3
Acceptance criteria	User can obtain a shareable link to the current view or report; Link opens the same view for other users without requiring them to reapply filters

Table 13: User story US12

US13 – Fast User Experience

As a	Citizen
I want to	Experience fast loading times when using the platform
So that	I do not abandon the platform due to slow responses
Priority	Must
Effort	5
Acceptance criteria	Under normal load, main dashboards load in under 2 seconds for test users; Pagination or lazy loading is used where necessary to avoid rendering very large lists at once

Table 14: User story US13

US14 – Monitoring Ingestion Jobs

As a	Technical Administrator
I want to	Monitor ingestion jobs and detect failures
So that	I can quickly react if external providers change or if ingestion stops
Priority	Should
Effort	5
Acceptance criteria	There is a view or log where the status of recent ingestion jobs is visible; For each job, the system stores timestamp, data source, and result (success or failure); Failure entries include a brief error message to guide debugging

Table 15: User story US14

5 Initial Database Architecture

Data Model Overview

The Air Quality Platform's data model is organized into four logical components that reflect different functional areas of the system. The relational schema handles structured, long-lived business data (stations, readings, users, alerts, recommendations, and reports), while a separate NoSQL store manages highly dynamic user-specific configurations. This separation ensures clean responsibilities and optimal performance for each data type.

The four main components are:

- **Geospatial & Monitoring:** Manages monitoring stations, pollutants, and raw air quality measurements.
- **Users & Access Control:** Handles user accounts, roles, and permissions.
- **Alerts & Recommendations:** Supports user-configured alerts and health-oriented suggestions.
- **Reporting & Analytics:** Provides operational reports and analytical aggregations for business intelligence.

Relational Model – ER Diagrams by Component

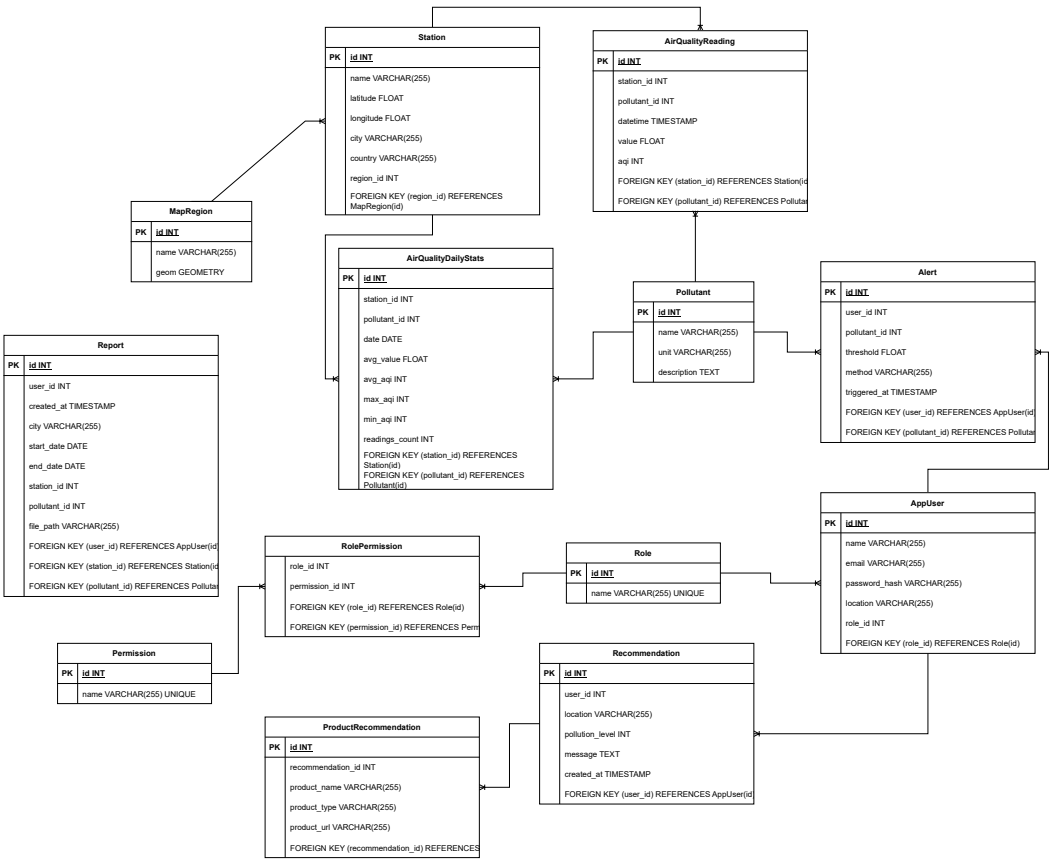

Figure 2: Complete ER diagram for the Air Quality Platform

Geospatial & Monitoring This component contains the core operational entities for air quality monitoring. The `Station` table stores metadata and geographic coordinates for each monitoring station. The `Pollutant` table catalogs different pollutants (e.g., PM2.5, NO_2, O_3) with their measurement units. The `AirQualityReading` table captures raw measurements from stations, linking each reading to a specific station and pollutant. The `MapRegion` table supports geospatial filtering and visualization, optionally using PostGIS geometry types for advanced mapping features.

Users & Access Control This component implements role-based access control (RBAC) for the platform. The `AppUser` table (renamed from `User` to avoid conflicts with PostgreSQL reserved words) stores registered users with basic profile information and a reference to their assigned role. The `Role` and `Permission` tables define distinct user roles (e.g., Admin, Analyst, Citizen) and granular permissions. The `RolePermission` junction table establishes a many-to-many relationship, allowing flexible permission assignment to roles.

Alerts & Recommendations This component supports personalized user engagement. The `Alert` table stores threshold-based alert configurations created by users (e.g., notify when PM2.5 exceeds 50 $\mu g/m^3$), with fields for the pollutant, threshold value, delivery method (email, SMS), and trigger timestamp. The `Recommendation` table contains health-oriented suggestions generated based on pollution levels and user location. The `ProductRecommendation` table links recommendations to certified protection products (e.g., face masks, air purifiers), providing actionable guidance during high pollution events.

Reporting & Analytics This component bridges operational and analytical workloads. The `Report` table stores metadata for user-generated reports, including parameters (city, date range, station, pollutant) and file paths for exported documents. The `AirQualityDailyStats` table is an analytical entity that contains pre-aggregated daily statistics per station and pollutant (average, maximum, minimum AQI values, and reading counts). This aggregation table supports efficient business intelligence queries and dashboard visualizations without repeatedly scanning the raw `AirQualityReading` table.

NoSQL Data Model for Preferences and Dashboards

To avoid storing semi-structured, frequently changing configuration data in the relational schema, the platform uses a separate NoSQL document store (e.g., MongoDB or Azure Cosmos DB) for two specific collections:

- **user_preferences:** Stores per-user settings such as UI theme (light/dark mode), default city for dashboard views, favorite pollutants to monitor, notification channels, language preferences, and other customizable options. This data changes frequently based on user interactions and does not require relational integrity constraints.
- **dashboard_configs:** Stores dashboard layout configurations, including widget positions, visibility settings, chart types, and time range preferences. This allows users to personalize their analytics dashboards without impacting the relational schema.

This design removes JSON fields from the relational model (which would complicate querying and schema evolution) and leverages the flexibility of NoSQL databases for schema-less, rapidly evolving configuration data. The relational database remains responsible for long-lived, structured business data with strong consistency requirements.

Entity Overview

The following table summarizes all entities in the data model, grouped by component:

Component	Entity	Description	Main Attributes	Type
Geospatial & Monitoring	Station	Physical air quality monitoring station	id, name, latitude, longitude, city, region_id	Operational
Geospatial & Monitoring	Pollutant	Catalog of pollutants and measurement units	id, name, unit, description	Operational
Geospatial & Monitoring	AirQualityReading	Raw air quality measurements	id, station_id, pollutant_id, datetime, value, aqi	Operational
Geospatial & Monitoring	MapRegion	Geographic regions for filtering and visualization	id, name, geom	Operational
Users & Access Control	AppUser	Registered platform user	id, name, email, password_hash, role_id	Operational
Users & Access Control	Role	User role definition	id, name	Operational
Users & Access Control	Permission	Granular permission definition	id, name	Operational
Users & Access Control	Role-Permission	Many-to-many role-permission mapping	role_id, permission_id	Operational
Alerts & Recommendations	Alert	Threshold-based user alerts	id, user_id, pollutant_id, threshold, method	Operational
Alerts & Recommendations	Recommendation	Health-oriented suggestions based on AQI	id, user_id, location, pollution_level, message	Operational
Alerts & Recommendations	Product-Recommendation	Suggested protection products	id, recommendation_id, product_name, product_type	Operational
Reporting & Analytics	Report	User-generated report metadata	id, user_id, city, start_date, end_date, file_path	Operational
Reporting & Analytics	AirQualityDailyStats	Daily aggregated air quality statistics	station_id, pollutant_id, date, avg_aqi, max_aqi	Analytical
Configuration (NoSQL)	user_preferences	Per-user UI and notification settings	theme, default_city, favorite_pollutants	Config
Configuration (NoSQL)	dashboard_configs	Dashboard layout and widget configurations	widgets, positions, visibility	Config

Table 16: Entity overview by component and type

The *Operational* entities handle transactional workloads with strong consistency re-

quirements, while the *Analytical* entity (`AirQualityDailyStats`) supports business intelligence queries. The *Config* entities reside in the NoSQL store and provide flexible schema evolution for user preferences and dashboard layouts.

6 Data System Architecture – Explanation

Architecture overview

The Air Quality Platform adopts a simplified, layered architecture designed to meet the project's requirements without introducing unnecessary complexity. The system is organized into five main layers:

Clients and web frontend A single responsive web application serves as the primary client interface, providing tailored views for three distinct user roles: citizens seeking real-time air quality information, researchers analyzing trends and patterns, and technical administrators managing the platform's operation. This unified frontend eliminates the need for separate mobile applications or external business intelligence tools as first-class components.

API layer The platform exposes two REST-based JSON-over-HTTP endpoints. The Public REST API serves citizen and researcher requests for air quality data, historical trends, and alerts. The Admin REST API provides administrative functionality for system configuration, user management, and operational monitoring. GraphQL was intentionally excluded to maintain simplicity and reduce the learning curve for the development team.

Data layer A single PostgreSQL instance functions as both the operational and analytical data store, hosting core entities such as stations, readings, users, alerts, recommendations, and reports. The `AirQualityDailyStats` table stores pre-aggregated daily statistics to accelerate dashboard and reporting queries. A small NoSQL document store complements PostgreSQL by managing user-specific configuration data, specifically `user_preferences` and `dashboard_configs`, which exhibit flexible schemas unsuited to rigid relational tables.

Batch jobs and background processing Three scheduled components handle data ingestion and transformation. The Ingestion Job periodically pulls raw measurements from external air quality APIs. The Normalizer and Validator component standardizes units, normalizes timestamps, and handles missing or invalid values before storing clean readings in PostgreSQL. Finally, the Daily Aggregation Job runs once per day to populate `AirQualityDailyStats` with aggregated metrics.

Observability A single Application Logs component collects structured logs from all services—Public API, Admin API, Ingestion Job, Normalizer/Validator, and Daily Aggregation Job. This lightweight approach replaces the need for complex log ETL pipelines or dedicated observability stacks like Elasticsearch and Grafana, which exceed the scope of this academic project.

Recommendations The recommendation module provides health-related suggestions based on current pollution levels (e.g., recommending masks or limiting outdoor activities). It does not support e-commerce functionality or product sales.

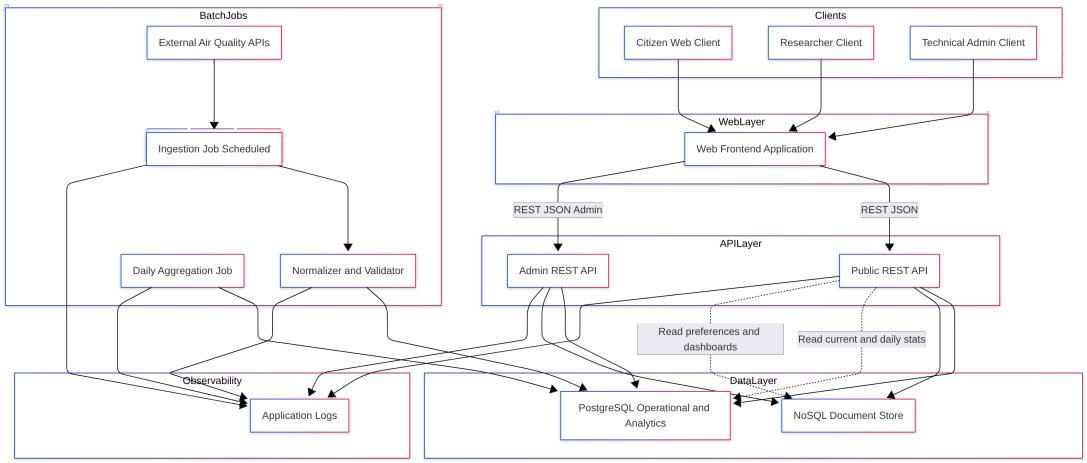

Figure 3: High-level architecture of the Air Quality Platform

The architecture diagram (Figure 3) illustrates the relationships between system components. Clients interact with the web frontend, which consumes the Public or Admin REST APIs. These APIs query both PostgreSQL and the NoSQL store. Meanwhile, batch jobs operate independently to ingest and transform data, writing structured logs to the Application Logs component for operational visibility.

Information flow and data transformations

The platform implements a structured data pipeline that transforms raw measurements from external providers into actionable insights for end users. This flow ensures data quality, supports both real-time and historical analysis, and maintains system performance within acceptable bounds.

The pipeline begins when the Ingestion Job executes on a configurable schedule—typically every few minutes to every hour, depending on the update frequency of external air quality APIs. The job polls these endpoints and retrieves raw JSON payloads containing pollutant concentrations, timestamps, and station metadata. These payloads pass immediately to the Normalizer and Validator component, which performs critical transformations:

- **Unit conversions**: Raw measurements may arrive in different units (e.g., µg/m^3 vs. ppm). The normalizer standardizes all readings to a consistent unit system.
- **Timestamp normalization**: External APIs may use different time zones or timestamp formats. The normalizer converts all timestamps to UTC and stores them in ISO 8601 format.
- **Missing and invalid value handling**: Missing fields are flagged with `NULL`, while readings outside physically plausible ranges (e.g., negative concentrations) are logged and rejected.

Once validated, clean readings are inserted into the `AirQualityReading` table in PostgreSQL as individual records. This table grows at approximately 14,400 rows per day (50

stations × 3 pollutants × 96 readings per day), as detailed in the performance estimates subsection.

The Daily Aggregation Job runs once per day—typically during off-peak hours—to compute per-day, per-station, per-pollutant aggregates. These aggregates (minimum, maximum, average, and 95th percentile values) are written into the `AirQualityDailyStats` table. This analytical table accelerates dashboard and reporting queries by eliminating the need to scan millions of raw readings for time-series visualizations or historical comparisons. The aggregation job adds approximately 150 rows per day to `AirQualityDailyStats`, yielding a total of 50,000–60,000 rows over a 3-year period.

The Public and Admin REST APIs serve as the primary interface for data access. Both APIs read from PostgreSQL, selecting either raw readings (for recent, high-resolution data) or daily aggregates (for dashboards, reports, and trend analysis). The NoSQL document store provides supplementary data for personalized experiences, such as user-configured dashboard layouts and alert preferences.

Finally, the web frontend consumes these REST APIs to render views for citizens, researchers, and administrators. Citizens view real-time air quality indices and receive threshold-based alerts. Researchers query historical trends and download datasets for offline analysis. Administrators monitor system health, configure stations, and review ingestion logs.

This end-to-end pipeline directly supports the platform's non-functional requirements: sub-second query performance (NFR1) is achieved through pre-aggregation and indexing, high data quality (NFR2) is ensured by the normalization and validation layer, and efficient report generation (NFR4, NFR6) leverages the `AirQualityDailyStats` table. The decision to use a single PostgreSQL instance with automated backups satisfies fault tolerance requirements (NFR7) without necessitating distributed databases or multi-region deployments, which remain outside the scope of this academic project (NFR8).

Performance and data estimates

This subsection provides concrete data volume and workload assumptions for the Air Quality Platform. These estimates support performance-related design decisions and align with the revised non-functional requirements, ensuring that the system remains scalable within the scope of the course project without requiring a multi-region or big-data infrastructure.

The platform's monitoring network is assumed to consist of approximately 50 stations, each tracking 3 pollutants (e.g., PM2.5, PM10, NO_2) with readings collected every 15 minutes. This configuration yields roughly 14,400 raw readings per day inserted into the `AirQualityReading` table. Over a 3-year operational period, the system accumulates approximately 15.8 million raw readings. In contrast, the `AirQualityDailyStats` table—which stores pre-aggregated daily statistics—grows at a much slower rate of 150 rows per day (one per station per pollutant), totaling approximately 50,000–60,000 rows over several years. The user base is estimated at around 1,000 registered accounts (`AppUser`), with 2,000–3,000 configured alerts.

Table 17 summarizes the estimated volumes for the main operational entities.

Table 17: Data volume estimates for key entities over a 3-year period

Entity	**Approx. volume (3 years)**	**Growth per day**	**Notes**
AirQualityReading	∼15.8M rows	14,400 rows	Raw sensor readings
AirQualityDailyStats	∼50–60k rows	150 rows	Aggregated daily statistics
AppUser	∼1,000 users	Low	Registered user accounts
Alert	∼2–3k alerts	Low	Threshold-based notifications

The workload exhibits a read-heavy pattern, with dashboard and analytical queries dominating system usage. Writes are moderate and predictable, consisting primarily of continuous ingestion into `AirQualityReading` and periodic batch updates to `AirQualityDailyStats`. To maintain sub-second query performance (NFR1) and support efficient report generation (NFR4, NFR6), composite indexes are applied to frequently queried columns: (`station_id`, `datetime`) on `AirQualityReading` and (`station_id`, `date`) on `AirQualityDailyStats`. Most dashboard and reporting queries leverage the aggregated statistics table, minimizing expensive full-table scans on the raw readings. This indexing strategy, combined with the use of a single PostgreSQL instance with automated backups, satisfies the fault tolerance requirements (NFR7) without necessitating a distributed or multi-region deployment.

Should data volumes grow significantly beyond current projections, time-based partitioning (e.g., monthly or yearly partitions on `AirQualityReading`) or read replicas can be introduced incrementally. This approach aligns with NFR8, which treats horizontal scalability as a future design goal rather than an immediate requirement, ensuring that the architecture remains extensible without overengineering for the present scope.

7 Information Requirements

2.1 Organizational Information Requirements

OIR	Description
District-level air quality indicators	To meet sustainability, regulatory compliance and policy formulation objectives.
Historical trends and longitudinal analysis	To evaluate the impact of environmental public policies over time.
System performance metrics (uptime, latency, scalability)	To ensure service continuity and operational quality.
Aggregate user behavior data	To optimize the digital experience, educational campaigns and loyalty strategy.
Information on successful and unsuccessful integrations with third parties (API partners)	For technological or commercial improvement and expansion decisions.
Premium feature usage statistics	To evaluate revenue models and retention strategies.

2.2 Asset Information Requirements

AIR	Description
Server and database specifications (PostgreSQL, clusters)	To ensure processing and storage capacity.
System health status logs (processes, microservices, containers)	For predictive maintenance and stability monitoring.
Network configurations, load balancers, geographic redundancy	To ensure availability and fault tolerance.
Technical information from sensors and external data sources (frequency, format, authentication)	To maintain integrity and consistency of data ingestion.
Active and current licenses (third-party APIs, BI software like Metabase)	To ensure legal and technical software compliance.
Details of internal and external APIs, technical documentation, endpoint versioning	For maintenance and integration with new systems.

2.3 Project Information Requirements

PIR	Description
Deployment schedules for new modules (recommendations, BI, alerts)	For release planning and team coordination.
User story acceptance and validation criteria	For functional testing and compliance verification.
Backend technical specifications (APIs, pipelines)	For modular and scalable development.
Testing data, staging and validation environments	For automated testing and quality control before deployment.
Alerts and dashboards configuration documentation	To ensure that institutional customers receive the contracted features.
Evidence of compliance with legal and regulatory requirements (environmental, privacy)	To ensure regulatory adherence during development.

2.4 Exchange Information Requirements

EIR	Description
Data ingestion from external APIs (AQICN, Google, IQAir)	JSON, every 10 minutes, with integrity check and validation.
Historical data export for researchers	CSV and JSON formats; must allow filters by date, zone and pollutant.
PDF or email reports for governments and businesses	Customizable with selected indicators and automatic graphs.
API access for clients (tourism, health, mobility apps)	Authentication tokens, consumption limits, clear documentation.
Customizable alerts by mail, app or push notification	Based on user-defined thresholds or local regulation.
Embeddable visualization for educational institutions or municipalities	Responsive dashboards with easy integration via iframe or script.

8 Query Proposals

This section describes the purpose of each example query proposed in the architecture, linking them to the functional and performance requirements of the system. The examples below illustrate both SQL queries over the relational schema and NoSQL queries over the configuration store. SQL is used for air quality readings, alerts and recommendations, while NoSQL is used for user preferences and dashboard layouts.

1. Latest Air Quality Readings per Station

Code 1 Query Latest Air Quality

```sql
WITH latest_readings AS (
  SELECT
    aqr.id,
    aqr.station_id,
    aqr.pollutant_id,
    aqr.value,
    aqr.aqi,
    aqr.datetime,
    ROW_NUMBER() OVER (
      PARTITION BY aqr.station_id, aqr.pollutant_id
      ORDER BY aqr.datetime DESC
    ) AS rn
  FROM AirQualityReading aqr
  JOIN Station s ON aqr.station_id = s.id
  WHERE s.city = 'Bogota'
)
SELECT
  s.name         AS station_name,
  s.latitude,
  s.longitude,
  p.name         AS pollutant_name,
  lr.value,
  lr.aqi,
  lr.datetime    AS last_updated
FROM latest_readings lr
JOIN Station s   ON lr.station_id = s.id
JOIN Pollutant p ON lr.pollutant_id = p.id
WHERE lr.rn = 1
ORDER BY s.name, p.name;
```

Purpose: This query retrieves the most recent air quality readings for all pollutants measured in Bogotá stations. It supports real-time display of air quality cards on the dashboard and fulfills user stories requiring fast access to current conditions (e.g., US1, US4, US9).

Output: Returns station name, location, pollutant type, measured value, AQI, and timestamp of the latest measurement for each station in the given city.

2. Monthly Historical Averages by Pollutant and City

Code 2 Query Monthly Historical

```sql
SELECT
  date_trunc('month', stats.date) AS month,
  AVG(stats.avg_value)            AS avg_value,
  AVG(stats.avg_aqi)              AS avg_aqi
FROM AirQualityDailyStats stats
JOIN Station s   ON stats.station_id   = s.id
JOIN Pollutant p ON stats.pollutant_id = p.id
WHERE s.city = 'Medellin'
  AND p.name = 'PM2.5'
GROUP BY month
ORDER BY month DESC
LIMIT 36;
```

Purpose: This query supports longitudinal trend analysis by computing average air quality values per month for a specific pollutant. It leverages the analytical table `AirQualityDailyStats` to compute monthly averages efficiently without scanning all raw readings. This enables researchers and public policy analysts to track pollution evolution over time (e.g., US2, US5, US7).

Output: Returns a time series of average pollutant concentrations and AQI values by month for the last three years in the specified city.

3. Active User Alerts and Configurations

Code 3 Query User Alert Configurations

```sql
SELECT
  u.name        AS user_name,
  p.name        AS pollutant_name,
  COUNT(*)      AS trigger_count,
  MIN(aqr.datetime) AS first_triggered_at,
  MAX(aqr.datetime) AS last_triggered_at
FROM Alert a
JOIN AppUser u         ON a.user_id      = u.id
JOIN Station s         ON a.station_id   = s.id
JOIN Pollutant p       ON a.pollutant_id = p.id
JOIN AirQualityReading aqr
  ON aqr.station_id    = a.station_id
 AND aqr.pollutant_id  = a.pollutant_id
 AND aqr.aqi           >= a.threshold_aqi
 AND aqr.datetime      >= NOW() - INTERVAL '7 days'
WHERE a.is_active = TRUE
GROUP BY u.name, p.name
ORDER BY trigger_count DESC;
```

Purpose: This query analyzes alert patterns by counting how many times alerts were triggered in the last 7 days per user and pollutant. It joins user alert configurations with actual air quality readings that exceeded the configured AQI thresholds, helping administrators understand which pollution thresholds are most frequently triggered and how users interact with the alert system (e.g., US9, US14).

Output: Returns user name, pollutant name, trigger count, and first/last trigger timestamps for active alerts in the last 7 days, ordered by frequency.

4. Station Coverage and Data Completeness

Code 4 Query Station Coverage Analysis

```sql
SELECT
  s.city,
  s.country,
  COUNT(DISTINCT s.id)          AS station_count,
  COUNT(DISTINCT aqr.pollutant_id) AS pollutant_types_monitored,
  MAX(aqr.datetime)             AS latest_reading,
  COUNT(aqr.id)                 AS readings_last_24h
FROM Station s
LEFT JOIN AirQualityReading aqr
       ON s.id = aqr.station_id
      AND aqr.datetime >= NOW() - INTERVAL '24 hours'
GROUP BY s.city, s.country
ORDER BY s.country, s.city, readings_last_24h DESC;
```

Purpose: This query analyzes station coverage and data completeness across different geographic regions, supporting system monitoring and ensuring data quality for all covered areas (e.g., US1, US13, US14).

Output: Returns station count, monitored pollutant types, latest reading timestamp, and reading volume for each city and country.

5. User Recommendation History

Code 5 Query User Recommendation Analysis

```sql
SELECT
  u.name  AS user_name,
  r.location,
  r.pollution_level,
  r.suggestion,
  r.created_at,
  COUNT(pr.id) AS product_recommendations_count
FROM Recommendation r
JOIN AppUser u
  ON r.user_id = u.id
LEFT JOIN ProductRecommendation pr
  ON r.id = pr.recommendation_id
WHERE r.created_at >= NOW() - INTERVAL '30 days'
GROUP BY
  u.name,
  r.location,
  r.pollution_level,
  r.suggestion,
  r.created_at
ORDER BY
  r.created_at DESC,
  product_recommendations_count DESC;
```

Purpose: This query analyzes the recommendation engine's output and user engagement with suggested actions and products, helping optimize the personalization algorithms (e.g., US8, US10, US11).

Output: Returns user recommendation history including location, pollution level, suggestions, and associated product recommendations from the last 30 days.

6. Retrieving user preferences from the NoSQL store

Code 6 Query User Preferences (NoSQL)

```
{
  "user_id": "<USER_ID>",
  "notifications.enabled": true
}
```

Purpose: This query retrieves the preference document for a given user from the `user_preferences` collection, including notification settings and default city. It demon-

strates how the NoSQL store is used to manage user-specific configuration that changes frequently and has a flexible schema.

Output: Returns the configuration document for that user, which is used by the web frontend to personalize alerts and default views.

7. Retrieving dashboard configurations from the NoSQL store

Code 7 Query Dashboard Configurations (NoSQL)

```
{
  "user_id": "<USER_ID>",
  "widgets": {
    "$elemMatch": {
      "type": "pollutant_trend",
      "pollutant": "PM2.5",
      "enabled": true
    }
  }
}
```

Purpose: This query finds dashboard configurations where the user has enabled a pollutant trend widget for a specific pollutant. It illustrates how the NoSQL store supports flexible widget configurations and user interface customization.

Output: Returns the corresponding `dashboard_configs` documents, which drive the layout and widgets rendered in the web frontend.

9 Improvements to Workshop 2

Following the instructor's feedback, we revised Workshop 2 to bring the Air Quality Platform into a realistic scope for the course. The main issues identified were overly ambitious technical claims (strict real-time streaming, multi-region infrastructure, complex map-based features) and incomplete documentation of the data model, architecture, and expected workloads. We reorganized all deliverable components to address these concerns and ensure full traceability between requirements, architecture, data model, and performance assumptions.

Requirements. We rewrote the functional and non-functional requirements to eliminate unrealistic expectations. For FRs, we replaced strict "real-time" guarantees (FR1, FR4) with a *periodic ingestion* model aligned with typical external API update frequencies (every few minutes or hourly). We clarified that recommendations (FR8) are simple, rule-based suggestions tied to AQI thresholds, not complex ML-driven systems, and that product suggestions (FR10) are informational health guidance, not e-commerce. Advanced features such as map-based navigation (FR11) were moved to future work, and the responsive web interface (FR13) was explicitly scoped as a web app, not native mobile apps. All requirements are now classified as baseline, optional, or future work, giving us a clear and achievable roadmap.

For NFRs, we tied performance targets to plausible workloads instead of arbitrary thresholds on massive datasets. We replaced the "continuous 24/7 streaming" requirement (NFR2) with automated periodic ingestion, removed the distributed big-data infrastructure claim (NFR3), and dropped multi-region redundancy (NFR7) in favour of basic fault tolerance via backups. Horizontal scalability (NFR8) is now a design goal for the future, not a mandatory multi-node deployment for the baseline.

Architecture. We simplified the architecture to focus on what is achievable within the course: a single responsive web app as the only client, two REST APIs (Public and Admin), a single PostgreSQL instance for both operational and analytical data, and a small NoSQL document store limited to user preferences and dashboard layouts. We removed GraphQL, external BI tools, time-series databases, Elasticsearch, Grafana, and complex log ETL pipelines from the baseline design, keeping only application logs for observability. We documented the information flow explicitly—ingest from external APIs, validate and normalize data, store raw readings in PostgreSQL, aggregate them daily into `AirQualityDailyStats`, and serve dashboards and reports via REST—and tied this flow to the batch jobs that run on a schedule.

Data model. We redesigned the ER diagrams and entity overview to reflect the revised schema. We renamed `User` to `AppUser` to avoid SQL reserved-word conflicts, introduced `AirQualityDailyStats` as an analytical table to support efficient historical queries, and removed JSON fields from relational tables by moving user preferences and dashboard configurations to a separate NoSQL store (`user_preferences` and `dashboard_configs`). Instead of a single crowded ER diagram, we now provide ER diagrams per component (Geospatial & Monitoring, Users & Access Control, Alerts & Recommendations, Reporting & Analytics) for readability. We also documented the split between operational (raw readings, users, alerts) and analytical data (daily aggregates), which is key to our performance strategy.

Performance and data estimates. We added concrete estimates for the expected data volumes and workloads—approximately 50 stations, 3 pollutants per station, 15-minute reading intervals yielding around 14,400 readings per day and 15.8 million rows

over 3 years, plus around 1,000 registered users and typical concurrency of 20–50 active users at peak. These numbers justify the use of a single PostgreSQL instance with proper indexing and the introduction of `AirQualityDailyStats` for efficient aggregation queries. They also support our non-functional requirement targets for query performance (under 2 seconds for 95% of requests) and report generation (under 10 seconds).

Queries. We updated the query examples to match the revised schema and to demonstrate the new design decisions. The SQL snippets now use `AirQualityDailyStats` for historical analysis, reference the renamed `AppUser` entity, and illustrate threshold-based alerts. We also added two NoSQL examples showing how the configuration store (`user_preferences` and `dashboard_configs`) is queried in practice, addressing the instructor's request for concrete NoSQL usage.

User stories. We rewrote all 14 user stories using the standard format (*As a / I want to / so that*), adding priority (Must, Should, Could), effort estimates, and explicit acceptance criteria for each. The new user story set is aligned with the refined functional and non-functional requirements and focuses on the realistic baseline scope (data ingestion, dashboards, reports, alerts, simple recommendations, and responsive web access), while clearly marking advanced features (multi-region, complex maps) as future work.

Overall, these changes give us a more realistic, coherent baseline for the database and architecture analysis developed in Workshop 3.

10 Monitoring Strategy

Key Monitoring Requirements for the Air Quality Platform

Given the frequent ingestion schedule of our pipeline (periodic jobs running every 10 minutes) and the read-intensive behavior of users, monitoring focuses primarily on operational health, ingestion performance, and error detection. The following table summarizes the core metrics that must be tracked to guarantee platform stability.

Metric	**Description**	**Reason for Monitoring**
Execution time per ingestion job	Each ingestion cycle (every 10 minutes) must complete within an acceptable time window.	Detects performance degradation, API delays, or processing bottlenecks.
Ingestion delays	Measures the difference between expected ingestion timestamps and actual processed time.	Identifies late data insertion, network issues, or queue congestion.
API errors (4xx / 5xx)	Tracks client-side and server-side errors from external providers (AQICN, Google, IQAir).	Quickly identifies provider failures, quota exhaustion, or malformed requests.
Log retention (7 days)	All ingestion, processing, and pipeline logs are stored for one week.	Ensures traceability for incident analysis without excessive storage usage.

Table 18: Key Monitoring Requirements of the Platform

Monitoring Scenarios Aligned with Current Usage Patterns

Since current users only perform read operations and there is no multi-user editing, concurrency-related monitoring is minimal. Instead, monitoring focuses on ingestion consistency, system stability, and availability.

- **Ingestion every 10 minutes:** The ingestion pipeline must be monitored to ensure that no job exceeds the allowed duration or becomes desynchronized with the schedule.
- **Read-only dashboard usage:** All interactions with dashboards and reports depend on the consistency of recent ingestions. Monitoring query performance ensures users always access fresh and reliable AQI data.
- **Single-client configuration:** Because ingestion, reporting, and visualization are centralized, the risk of concurrent modifications is negligible. Monitoring therefore focuses on operational continuity.

Potential Operational Problems and Examples

- **Slow Ingestion Job:** If an ingestion cycle takes longer than 10 minutes, subsequent jobs may overlap, causing delays and cascading failures.
- **High API Error Rates:** A spike in 4xx errors may indicate quota exhaustion or invalid parameters; 5xx errors may signal temporary outages from external APIs.
- **Delayed Data Availability:** If ingestion is delayed, dashboards may show outdated AQI values, reducing reliability of alerts and recommendations.
- **Insufficient Log Retention:** If logs rotate too early, incident diagnosis becomes impossible; if they persist too long, storage cost increases unnecessarily.

Proposed Monitoring Controls and Tools

Metric	Control Mechanism	Justification
Execution time per ingestion job	Track duration via cron monitoring or job instrumentation (Prometheus + Grafana)	Detect anomalies early and prevent overlap between ingestion cycles.
Ingestion delays	Compare expected timestamps vs. actual ingestion completion time	Ensures that dashboards and reports operate on timely and accurate data.
API errors (4xx / 5xx)	Implement error counters and alert thresholds	Enables rapid detection of external provider issues and reduces downtime.
Log retention (7 days)	Configure log rotation policies (e.g., Loki or ELK)	Provides sufficient data for audits while keeping storage usage efficient.

Table 19: Monitoring Metrics and Control Mechanisms

Fit with Existing Architecture

The monitoring strategy fits naturally with the PostgreSQL-based architecture and can leverage additional optimizations as the platform evolves:

- **PostgreSQL with optional partitioning or TimescaleDB:** The platform can adopt time-based partitioning or TimescaleDB extensions in the future to improve query performance and enable more efficient ingestion monitoring as dataset size grows.
- **Materialized views for dashboards:** The architecture can utilize materialized views to accelerate dashboard queries; monitoring their refresh times would ensure smooth user experience and prevent stale visualizations.
- **Hybrid storage model (PostgreSQL + lightweight NoSQL):** Logs and ingestion traces can be stored efficiently and accessed independently of analytical workloads.
- **Containerized ingestion pipeline:** Each component exposes clear metrics and logs, enabling granular monitoring.
- **Centralized log retention (7 days):** Ensures consistent access to operational data without overwhelming storage resources.

11 Parallel and Distributed Database Design

Context and Scope of This Section

This section provides an analytical and forward-looking exploration of how the Air Quality Analysis Platform could evolve into a parallel and distributed database model as its operational scale increases. Currently, the platform relies on a centralized architecture that combines PostgreSQL with TimescaleDB extensions and a complementary NoSQL store. This setup is sufficient for its present data volume, geographic coverage, and analytical needs.

However, as the number of monitored locations grows and long-term retention of real-time geospatial measurements increases, the underlying infrastructure may require structural adjustments. The purpose of this section is therefore not to describe the current implementation, but to outline possible future design paths that ensure long-term scalability, reliability, and performance.

Rationale for Future Distributed or Parallel Architectures

Several projected developments of the platform may demand a transition toward distributed or parallel database mechanisms:

- **Increasing ingestion frequency and scale:** Data is ingested every 10 minutes from multiple external APIs. As more stations and cities are added, the volume of incoming measurements will increase considerably, generating higher write throughput.

- **Growth of historical time-series datasets:** Storing several years of air quality and geospatial data results in large, continuously expanding tables, intensifying analytical workloads over time.
- **More complex analytical and BI queries:** Public agencies, researchers, and institutions may require long-range trend evaluations, spatial joins, or aggregated models that can benefit from parallel execution.
- **Geographic expansion of the system:** Integrating additional regions introduces challenges such as increased latency, larger query spans, and potential regulatory constraints regarding data locality.
- **High availability and operational continuity:** Since the platform supports real-time dashboards, notifications, and decision-making applications, uninterrupted service becomes essential, encouraging the adoption of replicated and fault-tolerant environments.

In this context, adopting a distributed or parallel database model in the future would support:

- **Horizontal scalability** for both ingestion and analytical processes.
- **Lower latency** through geographically localized data nodes.
- **Separation of workloads** between ingestion, storage, and business intelligence layers.
- **Higher availability** through replication and automated failover mechanisms.

Potential Distribution and Fragmentation Approaches

The following strategies describe how the platform's data could be partitioned or distributed across multiple nodes as the system grows:

Temporal Fragmentation (Time-Based Partitioning Across Nodes)

Air quality data is inherently chronological. A natural distribution approach would be to fragment data by time ranges (e.g., monthly or quarterly segments), placing older partitions on separate nodes. This approach supports:

- Faster queries over recent data (typically most requested).
- Distributed processing for long historical analyses.
- Efficient archival and data-retention policies.

Geographical Fragmentation (Region-Based Distribution)

Since the system manages air quality readings by city and country, data could be distributed by geographic zones. Benefits include:

- Lower latency for regional queries.
- Improved performance for geospatial filters and location-based analytics.
- Compliance with potential future regional data regulations.
- Reduced load on the primary cluster during regional peak usage.

Hybrid Fragmentation Model (Time + Geography)

A combined model could be applied when the density of readings increases significantly. For example:

- Partitions by country distributed across regional nodes.
- Within each region, temporal partitions optimized for time-series operations.

This hybrid model aligns with the platform's long-term objective of supporting multi-city and multi-country audiences efficiently.

Replication for High Availability

To maintain uninterrupted service, read replicas could be introduced for:

- Dashboards and public-facing visualizations.
- Analytical workloads executed by BI tools or researchers.

This ensures that ingestion processes do not compete with visualization or reporting tasks, improving both reliability and performance.

Parallel Execution for Analytical Queries

As historical datasets grow, analytic operations—such as aggregations, pollutant correlation analysis, and geospatial pattern detection—could benefit from:

- Parallel execution plans within PostgreSQL clusters.
- Distributed compute nodes dedicated to research-intensive workloads.

These enhancements strengthen the platform's ability to support institutional and scientific stakeholders.

Summary

Although the platform currently operates under a centralized database architecture, projected growth in data volume, geospatial coverage, and analytical requirements makes it necessary to anticipate more sophisticated storage and processing models. The distribution strategies outlined—temporal, geographic, hybrid, and workload-based—provide feasible migration paths that maintain alignment with the platform's objectives of scalability, high availability, and low-latency access.

12 Performance Improvement Strategies

The goal is not to introduce heavy distributed technologies prematurely, but to identify realistic optimizations that match the system's workload, data characteristics and projected growth.

Partitioning of Operational and Analytical Data

Rationale

The platform stores high-frequency air quality readings and generates daily analytical summaries. Although the system currently operates on a single PostgreSQL instance, partitioning can significantly improve query latency and maintenance operations as the dataset grows.

Strategy: Temporal Partitioning

The `AirQualityReading` table can be divided into monthly partitions. This approach is aligned with the ingestion pattern (continuous, time-ordered data) and the most common dashboard filters (daily or monthly views).

- Recent months stay in fast storage for dashboards and near-real-time analytics.
- Older partitions can be compressed, moved to cheaper storage, or archived.

Strategy: Geographic Partitioning (Future)

If the platform expands beyond a single city or region, additional partitions can be introduced by geographic area (`country`, `city`). This hybrid model (time + geography) reduces scan ranges and minimizes contention on the operational store.

Benefits

- Faster query execution for dashboards filtered by city or date.
- Simplified retention policies, since old partitions can be dropped or archived directly.
- Reduced bloat and improved vacuum performance on hot partitions.

Trade-offs

- Requires discipline in query design to ensure proper use of filters.
- Excessive partitions (hundreds) may degrade planning time if not managed.

Read Replicas for Workload Separation

Rationale

Even in a modest-scale platform, citizen dashboards, research queries and administrative reports can generate substantial read traffic over historical data. These long-running scans may interfere with ingestion jobs or analytical batch operations.

Strategy

Introduce one or more asynchronous read replicas that serve:

- Citizen dashboards and maps.
- Researcher queries over historical records.
- Public-facing visualization endpoints.

The primary node focuses exclusively on:

- ingesting new measurements,
- running validation and normalization steps,
- generating daily aggregates.

Benefits

- Eliminates competition between ingestion and analytical reads.
- Reduces latency during peak pollution events when dashboard traffic spikes.
- Provides a pathway toward high availability (automatic failover).

Trade-offs

- Replication lag introduces slight delays in dashboards (a few seconds).
- Additional monitoring is required to detect replica staleness.

Parallel Execution for Analytical Queries

Rationale

The `AirQualityDailyStats` table is accessed by researchers and public organizations to run long-range analyses. As the table grows (multiple cities, multi-year retention), single-threaded query plans may become slow.

Strategy

Enable PostgreSQL's parallel query execution mechanisms for:

- large temporal scans (multi-month or multi-year),
- computations of pollutant averages or correlations,
- geospatial filters (bounding boxes or radial searches).

Parallel plans distribute segments of the scan across multiple CPU cores, reducing total execution time without modifying the database topology.

Benefits

- Significant reductions in analytical query runtime.
- Zero changes to application logic.

Trade-offs

- Requires tuning of PostgreSQL parameters (e.g., `max_parallel_workers_per_gather`).
- Benefits depend on server CPU availability.

Optimized Batch Ingestion and Daily Aggregation

Rationale

The ingestion workflow (external APIs → ingestion job → normalizer → PostgreSQL) is executed on a scheduled basis. Although not real-time, it must remain predictable and resilient against API delays or inconsistent data.

Strategy

- Use bulk inserts instead of row-by-row ingestion.
- Apply batching windows aligned with partition boundaries.
- Run daily aggregation during low-traffic periods.
- Use indexes optimized for ingestion (e.g., partial indexes on active partitions only).

Benefits

- Lower write amplification during ingestion.
- Guaranteed freshness of daily analytical data.
- Stable ingestion latency regardless of traffic patterns.

Caching of Hot Data at the API Layer

Rationale

Certain values are accessed extremely frequently—such as the latest AQI per station, daily recommendations and alert levels—while being updated only every few minutes.

Strategy

Use a short-lived in-memory cache (within the API server) to store:

- latest readings per station,
- precomputed recommendation text,
- daily aggregates for the current day.

Benefits

- Reduces repetitive database queries.
- Improves responsiveness for high-traffic endpoints.

Summary of Proposed Strategies

Strategy	Key Benefit	Main Trade-off
Temporal and geographic partitioning	Fast, selective scans and easier retention	Requires careful query design and partition management
Read replicas	Offloads dashboards and analytics	Replication lag and additional monitoring
Parallel execution in PostgreSQL	Faster analytical queries with no redesign	Requires parameter tuning and CPU availability
Optimized batch ingestion	Predictable ingestion and cleaner partitions	Requires ingestion window coordination
API-level caching	Reduces repeated DB access	Only useful for short-lived, stable data

References

Group, P. G. D. (2025). Postgresql documentation [Accessed Sep 22, 2025]. https://www.postgresql.org/docs/

Inc., M. (2025). Minio [Accessed Sep 22, 2025]. https://min.io/

Inc., T. (2025). Timescale documentation [Accessed Sep 22, 2025]. https://docs.timescale.com/tutorials/latest/real-time-analytics-transport/

Labs, G. (2025). Grafana [Accessed Sep 22, 2025]. https://grafana.com/