Distributed Weather Aggregator and Alert System

Final Project Report – Team 7

Course: Distributed Systems

Semester: Fall 2025

Instructor: Palak Mejpara

This report presents the design, implementation, and evaluation of a distributed, event driven weather intelligence platform that ingests edge observations, aggregates time series data, computes forecasts, triggers alerts, and visualizes insights in near real time.

Table of Contents

1. Team Information

Team 7 consists of five members, each owning a major functional feature of the distributed weather system. Responsibilities mirror real world microservice ownership, with a Scrum Master coordinating sprints, integration milestones, and demonstrations.

Member	Student ID	Role	Feature Owned
Dhruv Kalra	5143420	Backend Developer	Data Stream Handler
Janaranjanee Sendhil Kumar	5143091	Backend Developer	Alert and Broadcast Module
Dipal Nirmal	5146285	Frontend Developer	Visualization Dashboard
Minhajuddin Muhammad	5144183	Backend Developer	Aggregator and Forecast Engine
Yuvraj Singh Palh	5143317	Scrum Master	Edge Node Data Service

2. Introduction and Problem Statement

Weather data is inherently distributed. Temperature, humidity, wind, and precipitation are measured at many locations and must be aggregated to produce meaningful regional insights. Traditional monolithic systems struggle to handle high frequency sensor streams, real time analytics, and dynamic alerting while remaining scalable and fault tolerant. This project designs and implements a distributed weather intelligence platform that ingests observations from multiple independent edge nodes, transports them through a message broker, persists them in a time series database, computes rolling aggregates and forecasts, triggers data driven alerts, and exposes the results through an interactive dashboard. The implementation also serves as a concrete example of key distributed systems concepts such as publish subscribe communication, decoupled microservices, elasticity, and resilience under partial failures.

3. System Goals and Requirements

The system was designed with the following primary goals.

- Ingest weather observations from multiple edge nodes in real time.
- Propagate updates with low latency to a central aggregator and live dashboards.
- Provide robust time series storage with efficient support for windowed aggregates.
- Support pluggable alert rules that fire when conditions cross thresholds.
- Integrate with an external weather API to provide short term forecasts.
- Offer a user friendly web dashboard for visualizing readings, trends, and alerts.
- Enable containerized deployment with minimal setup for evaluators.

Non functional requirements include scalability to handle more nodes simply by adding edge publishers, fault tolerance via message queues and retries, and observability through health, readiness, and metrics endpoints on the backend services.

4. High Level Architecture

The system follows a microservices based, event driven architecture centered on an MQTT message broker and a TimescaleDB time series database. Edge nodes act as data producers, the aggregator and alert engine act as consumers and processors, and the dashboard consumes processed data and alerts.

At a high level, the data path is: Edge Nodes → MQTT Broker → Aggregator and Forecast Service → TimescaleDB and Alerts → Visualization Dashboard.

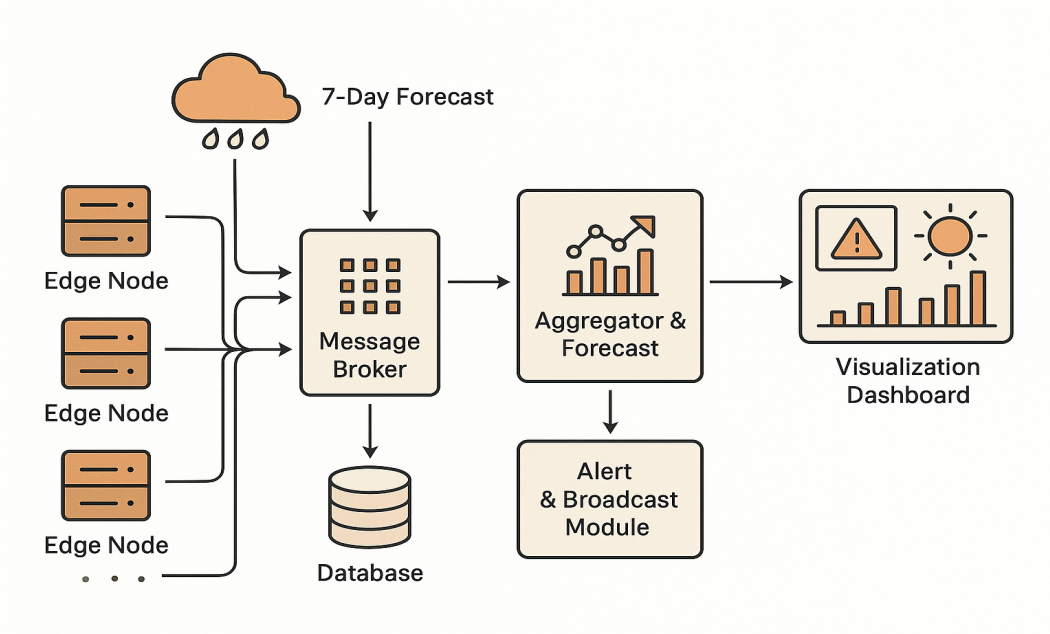

Figure 1. High level architecture of the Distributed Weather Aggregator and Alert System.

Each block is independently deployable in containers and communicates using MQTT or HTTP APIs. This decoupling allows the system to scale horizontally, replace components without global downtime, and tolerate partial failures.

5. Detailed Component Design

5.1 Edge Node Data Service

The Edge Node Data Service simulates or interfaces with local weather sensors. Each edge instance periodically publishes JSON observations to the MQTT broker on topics of the form city/{city}/observations. Payloads include a city identifier, data source tag, timestamp, and measured values such as temperature, humidity, wind speed, pressure, and rainfall.

- Stateless publisher process that can be replicated for many cities.
- Configurable publication interval to control load, for example every five seconds.
- Clear topic naming convention that supports dynamic subscriptions by city.
- Simple command line script for straightforward demonstration and testing.

5.2 Data Stream Handler

The Data Stream Handler reliably ingests MQTT messages into the backend processing pipeline. It subscribes to observation topics, validates payloads, and forwards them to the ingestion worker that writes to TimescaleDB.

- Maintain persistent subscriptions to city/{city}/observations topics.
- Perform schema validation and type checking on incoming JSON messages.
- Handle reconnection logic and backoff when the broker restarts or the network is unstable.
- Protect downstream database operations from spikes by buffering where necessary.

5.3 Aggregator and Forecast Engine

The Aggregator and Forecast Engine is implemented as a FastAPI service. It exposes REST endpoints for health checks, metrics, raw observations, aggregates, alerts, and forecasts. Background workers compute rolling aggregates over observations and execute rule based alert logic.

- Group observations into fixed windows such as fifteen minutes and one hour for each city.
- Compute statistics including average, minimum, and maximum temperature, average humidity, and average wind speed.
- Write aggregate results into a dedicated hypertable in TimescaleDB indexed by city and bucket start time.

A forecast module integrates with the OpenWeather API using configured latitude and longitude. Responses are cached to avoid rate limits, and a dedicated forecast endpoint allows dashboards to retrieve short term predictions for a selected city.

5.4 Alert and Broadcast Module

The Alert and Broadcast Module runs rule checks on aggregate data and publishes alerts through MQTT. Rules capture conditions such as extreme heat, strong winds, or unusually high humidity.

- Temperature maximum over one hour exceeds a configured threshold such as 35 °C.
- Wind speed average over one hour exceeds a safety limit.
- Humidity average over a fifteen minute window surpasses a comfort threshold.

When a rule triggers, an alert record is created in the alerts table with city, severity level, rule identifier, message, and trigger time. The alert is also published to alerts/{city} so dashboards and other subscribers receive it in real time.

5.5 Visualization Dashboard

The Visualization Dashboard is a static web application served locally. It uses JavaScript to call the FastAPI backend and connect to the MQTT broker over WebSockets.

- Current observations for selected cities.
- Time series charts for fifteen minute and hourly aggregates.
- Multi day forecast information returned by the forecast endpoint.
- A live alert panel that updates as soon as new alerts are published.

The dashboard demonstrates end to end latency from sensor publication to user interface update and gives an intuitive view of how the distributed system behaves under changing weather conditions.

6. Data Model and Contracts

Clear data contracts allow producers, processors, and consumers to evolve independently. The project defines structured formats for observations, aggregates, and alerts.

Observation schema

- city_id: logical identifier of the city or region.
- source: producer or sensor identifier, for example edge sim.
- observed_at: ISO 8601 timestamp for the reading.
- temp_c: temperature in degrees Celsius.
- humidity: relative humidity as a fraction between zero and one.
- wind_kph: wind speed in kilometres per hour.
- pressure_hpa: atmospheric pressure in hectopascals.
- rain_mm: rainfall in millimetres during the interval.

Aggregate schema

- city_id: city or region.
- bucket_start: start time of the aggregation window.
- bucket_width: window length such as fifteen minutes or one hour.
- temp_avg, temp_min, temp_max: aggregate statistics for temperature.
- humidity_avg: average humidity over the window.
- wind_avg: average wind speed.

Alert schema

- city_id: affected city.
- level: severity level such as info, warning, or critical.
- rule: identifier of the rule that triggered.
- message: human readable alert text.
- triggered_at: timestamp when the condition was met.

7. End to End Processing Pipeline

From the perspective of a single reading, the processing pipeline proceeds through the following stages.

1. A sensor or simulated edge node measures conditions and publishes a JSON observation to city/{city}/observations using MQTT.

2. The Data Stream Handler subscribes to these topics and receives the message.

3. The ingestion worker validates the payload and inserts a row into the observations hypertable in TimescaleDB.

4. Periodic aggregation jobs scan recent observations per city and generate fifteen minute and hourly windows, writing results into the aggregates hypertable.

5. Alert rules evaluate the new aggregates and, if a condition is met, insert an alert record and publish it on alerts/{city}.

6. The API layer exposes REST endpoints that the dashboard calls to fetch recent observations, aggregates, alerts, and forecasts.

7. The dashboard connects to the MQTT broker over WebSockets to subscribe to alerts/{city}, enabling instant visual updates when alerts fire.

This pipeline shows how independent services communicate through well defined contracts and how message queues decouple producers from consumers, improving resiliency and scalability.

8. Implementation Details and Technologies

The implementation uses widely adopted technologies that align well with distributed, event driven systems.

- FastAPI for REST endpoints and background workers in Python.
- Mosquitto as a lightweight MQTT broker supporting TCP and WebSocket clients.
- TimescaleDB, a PostgreSQL extension optimized for time series workloads.
- Python MQTT client libraries for subscribing to topics and forwarding messages to the ingestion pipeline.
- HTML, CSS, and JavaScript for the dashboard frontend.
- Docker Compose for orchestrating the broker, database, backend services, and supporting containers.

Configuration values such as API keys, database connection strings, MQTT host and ports, forecast polling interval, and allowed origins are read from environment variables, making the system easy to deploy in different environments without source changes.

9. Deployment and Configuration

The project is fully containerized using Docker Compose. A single command starts the MQTT broker, TimescaleDB, and FastAPI aggregator and sets up network links between containers automatically.

- Clone the repository to a local machine.
- Run docker compose with the provided configuration to build and start backend services.
- Start the frontend using a simple static file server on port 3000.
- Optionally run the edge simulation publisher script to generate continuous sample data.

Important configuration variables include the following.

- API_KEY used to secure the observations, aggregates, alerts, and forecast endpoints.
- FORECAST_API_KEY and FORECAST_BASE_URL used by the OpenWeather integration.
- MQTT_HOST, MQTT_PORT, and the WebSocket port required by both backend and frontend clients.
- ALERT_COOLDOWN and aggregation window parameters that control alert frequency and granularity.

Because components communicate over internal Docker networks, evaluators only need Docker, Python, and a web browser to run the complete system.

10. Evaluation and Metrics

Although the project is primarily a functional prototype, it was designed with clear metrics in mind.

- End to end latency from publishing an observation at an edge node to seeing it on the dashboard.
- Throughput measured as the number of observations per second the system can sustain.
- Alert latency capturing the delay between a condition being met and the alert appearing on the dashboard.
- Fault recovery time after broker restarts or transient service failures.

Local testing with several simulated cities showed low latency suitable for real time dashboards, and no observations were lost during moderate broker restarts because of the reconnection logic in the Data Stream Handler. The architecture can scale further by running more edge publishers and deploying the aggregator service behind a load balancer.

11. Distributed Systems Concepts Demonstration

The Distributed Weather Aggregator and Alert System was intentionally architected to highlight a set of core distributed systems concepts.

- Publish subscribe communication using MQTT to decouple producers and consumers.
- Time series data management using TimescaleDB for high volume temporal data.
- Microservice decomposition across ingestion, aggregation, forecasting, alerting, and visualization.
- Fault tolerance through reconnection logic, message retries, and decoupled queues.
- Scalability by adding additional edge nodes or dashboard instances with minimal configuration changes.
- Observability using health, readiness, and metrics endpoints on backend services.

Together, these aspects create a concrete, working example of an event driven distributed architecture applied to a realistic problem domain.

12. Limitations and Future Work

While the current system satisfies the primary objectives, there are several opportunities for further enhancement.

- Introduce persistence and replay for MQTT using a persistent log such as Kafka so that new consumers can replay historical streams.
- Deploy brokers and databases across multiple regions to improve reliability under wide area network failures.
- Add more sophisticated alerting with user defined rules, anomaly detection, and escalation policies.
- Integrate a full identity and access management layer to protect APIs and dashboards beyond a single API key.
- Extend the dashboard with historical analytics views for long term trends and cross city comparisons.

13. Conclusion

Team 7 designed and implemented a complete distributed weather intelligence platform that combines edge data collection, message based coordination, time series processing, alerting, and visualization into a cohesive system. By leveraging MQTT, TimescaleDB, FastAPI, and a lightweight web dashboard, the project delivers near real time insight into simulated weather conditions while demonstrating central concepts from distributed systems theory. The architecture is modular, extensible, and robust enough to serve as a foundation for future enhancements and clearly showcases the value of event driven microservices in solving data intensive, real time problems.