Live Weather Data Aggregation System

Big Data Project Report

Group 15

Members:
Do Doan Hoang Du – 20220060
Le Gia Huy – 20225498
Nguyen Tien Thanh – 20225459
Tang Tieu Long – 20225507

December 21, 2025

Abstract

The rapid expansion of the Internet of Things (IoT) has generated massive streams of real-time data, necessitating advanced architectures for ingestion, processing, and storage. In the context of meteorological data, the ability to process information with low latency while simultaneously maintaining a historical archive for long-term trend analysis is critical. This project presents the design and implementation of an end-to-end **Data Lakehouse** system tailored for live weather data aggregation.

The system is architected around the **Lambda Architecture** paradigm, effectively decoupling the processing logic to serve two distinct needs: a *Hot Path* for real-time monitoring and a *Cold Path* for historical analytics. Data ingestion is managed by **Apache NiFi**, which polls external APIs and buffers data into **Apache Kafka** to ensure system resilience. The core processing engine, **Apache Spark Structured Streaming**, performs complex event-time transformations, including windowed aggregations, watermarking for late data handling, and custom business logic application via User Defined Functions (UDFs).

The storage layer implements a dual-path strategy: **MongoDB** serves as the real-time NoSQL store, providing sub-millisecond access to the latest weather states, while **MinIO** (S3-compatible storage) hosting **Apache Iceberg** tables serves as the analytical Data Lake. This Data Lake is governed by the **Hive Metastore**, enabling high-performance SQL querying via **Trino**. Finally, **Apache Superset** provides an interactive visualization layer, offering dynamic dashboards for both real-time metrics and historical trends. The entire infrastructure is fully containerized using Docker, ensuring portability, reproducibility, and ease of deployment.

Contents

Chapter 1

Introduction

1.1 Project Context

In the era of Big Data, weather forecasting and monitoring systems generate vast amounts of data at high velocity. Traditional batch processing systems are often insufficient for modern requirements, which demand both real-time alerts and deep historical analysis. A robust data pipeline must handle the "Three Vs" of Big Data: Volume (storing years of historical data), Velocity (processing incoming streams in real-time), and Variety (handling semi-structured JSON data from APIs).

This project addresses these challenges by building a comprehensive Data Lakehouse. Unlike a traditional Data Warehouse, which can be rigid and expensive, or a Data Lake, which can become a "swamp" of unmanaged files, a Lakehouse combines the best of both worlds: the low cost and flexibility of object storage with the management and ACID transaction capabilities of a database.

1.2 Objectives

The primary objectives of this project are defined as follows:

1. **Data Ingestion:** To design a reliable mechanism for fetching weather data from external APIs (OpenWeatherMap) and handling failures or API rate limits.

2. **Real-Time Stream Processing:** To implement a processing engine capable of cleaning, transforming, and aggregating data using event-time logic.

3. **Hybrid Storage Architecture:** To implement a Lambda Architecture that supports both low-latency lookups for mobile/web apps and high-throughput scans for analytical queries.

4. **Interactive Analytics:** To provide a SQL interface and visualization dashboard that allows users to explore weather patterns both in historical and real-time.

Chapter 2

Data Source Specification

2.1 OpenWeatherMap API

To simulate a realistic IoT sensor network and gather live meteorological data, this project utilizes the **OpenWeatherMap API** as the primary data source. Specifically, we employ the *Current Weather Data* endpoint, which provides frequent updates on weather conditions for over 200,000 cities worldwide. This API was selected for its reliability, extensive documentation, and standard JSON output format, which mirrors typical real-world data streams.

2.2 API Configuration

The system is configured to poll the API at regular intervals (every 15 minutes) to gather a time-series dataset. The ingestion layer (Apache NiFi) constructs HTTP GET requests using the following standard parameters to ensure consistent data retrieval:

- **Endpoint:** `https://api.openweathermap.org/data/2.5/weather`
- **Query Parameters:**
  - `q={city_name}`: The target city (e.g., Hanoi, Ho Chi Minh City, Da Nang).
  - `appid={API_KEY}`: The unique authentication credential used to access the service.
  - `units=metric`: Configures the API to return temperature in Celsius and wind speed in meters/sec, standardizing the units for downstream processing.

2.3 Data Structure

The API returns data in a semi-structured JSON format. This nested structure necessitates specific transformation logic in the processing layer (using Jolt in NiFi and Schema definitions in Spark) to flatten and extract relevant metrics.

Below is a sample of the raw JSON payload received by the ingestion layer for a single city:

```json
{
  "coord": { "lon": 105.84, "lat": 21.02 },
  "weather": [
    {
      "id": 800,
      "main": "Clear",
      "description": "clear sky",
      "icon": "01d"
    }
  ],
  "base": "stations",
  "main": {
    "temp": 32.0,
    "feels_like": 35.5,
    "temp_min": 31.0,
    "temp_max": 33.0,
    "pressure": 1012,
    "humidity": 65
  },
  "wind": { "speed": 4.1, "deg": 120 },
  "dt": 1699945200,
  "sys": { "country": "VN", "sunrise": 1699916400, "sunset": 1699957200 },
  "timezone": 25200,
  "id": 1581130,
  "name": "Hanoi",
  "cod": 200
}
```

Listing 2.1: Sample OpenWeatherMap API Response

Key fields extracted for analysis include `main.temp` (Current Temperature), `main.humidity` (Humidity %), `wind.speed` (Wind Speed), and most importantly, `dt` (Unix epoch time), which serves as the event timestamp for windowed aggregations.

Chapter 3

System Architecture

3.1 Architecture Overview

The system follows a microservices architecture orchestrated by Docker Compose. Each component runs in an isolated container, communicating via a dedicated internal network. The data flow is linear and unidirectional, ensuring data lineage and simplifying debugging.

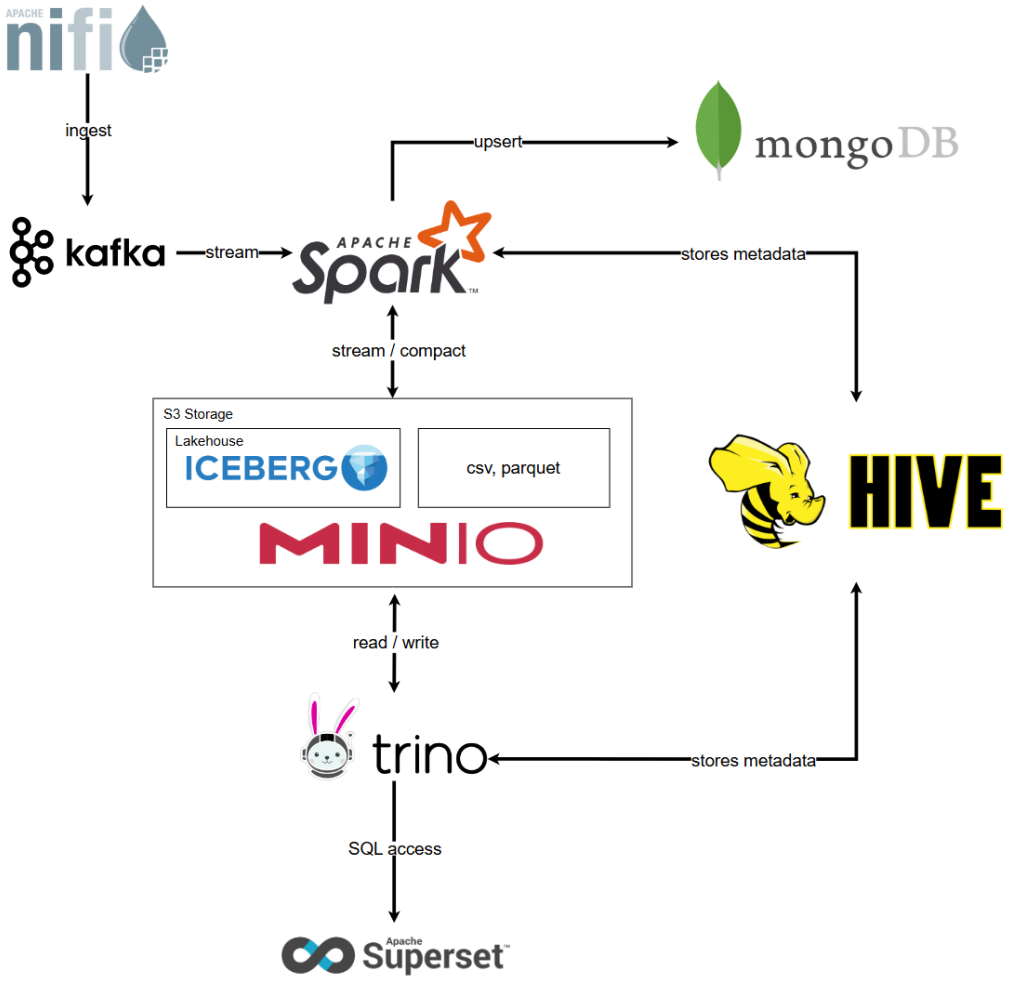


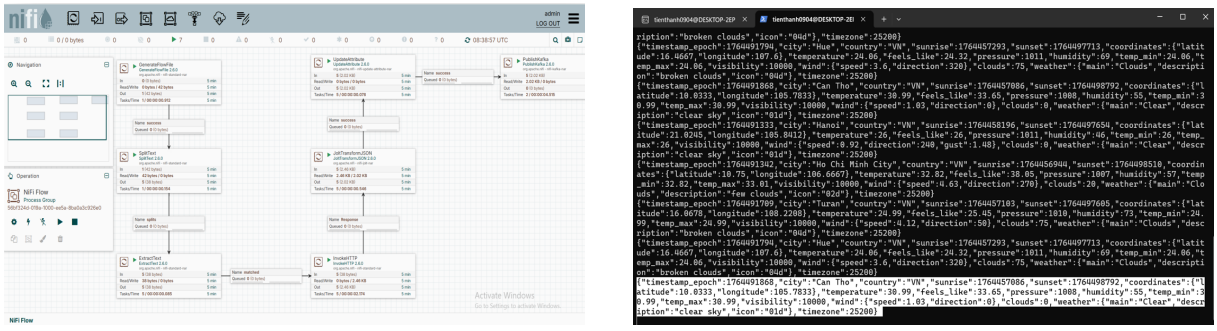
Figure 3.1: End-to-End Data Lakehouse Architecture

3.2 Detailed Component Breakdown

3.2.1 Ingestion Layer: NiFi & Kafka

The ingestion layer is responsible for decoupling the external data source from the internal processing logic.

- **Apache NiFi:** Apache NiFi serves as the data extraction and routing tool. It provides a visual interface to design data flows. In this project, NiFi is configured with a specific processor chain to automate data retrieval:
  - **GenerateFlowFile (Scheduler):** This processor acts as the trigger. It is configured with a Cron-driven scheduling strategy (e.g., every 15 minutes) to initiate the data collection workflow. It generates a FlowFile containing a list of target cities (e.g., Hanoi, Da Nang, Ho Chi Minh City).
  - **SplitText:** Since the initial FlowFile contains multiple cities, this processor splits the content line-by-line. This ensures that each city is processed as an individual event, allowing for parallel execution and better error handling per city.
  - **ExtractText:** This step extracts the city name from the FlowFile content and promotes it to a FlowFile Attribute (metadata). This attribute is then used dynamically in the subsequent API call.
  - **InvokeHTTP:** This is the core ingestion processor. It uses the city attribute to construct the dynamic URL for the OpenWeatherMap API and executes a GET request. The response body (raw JSON weather data) replaces the content of the FlowFile.
  - **JoltTransformJSON:** Before sending data downstream, NiFi performs light transformation using Jolt specifications. This step cleans the JSON structure, removes unnecessary fields, and ensures the timestamp format is consistent, preparing the data for the Spark schema.
  - **PublishKafka:** The final processor in the chain pushes the cleaned JSON data to the `weather-data` topic in Apache Kafka, making it available for the streaming engine.
- **Apache Kafka:** Apache Kafka acts as the distributed commit log and backbone of the architecture. Although configured with standard settings, its role is critical for system stability
  - **Decoupling:** Kafka completely decouples the ingestion rate from the processing rate. If NiFi fetches data faster than Spark can process (or if Spark is down for maintenance), Kafka buffers the messages on disk, preventing data loss.
  - **Backpressure Handling:** By serving as a buffer, Kafka naturally handles backpressure. It ensures that a burst of API data does not overwhelm the Spark cluster.
  - **Durability:** Even with default configurations, Kafka persists messages to disk. This provides a "replayability" feature, allowing us to re-process historical data streams if logic changes or errors occur in the processing layer.



(a) NiFi Flow

(b) Kafka Consumer Output

Figure 3.2: Data Ingestion Workflow

3.2.2 Processing Layer: Apache Spark

Apache Spark Structured Streaming (v3.5.1) is the computational heart of the system. It operates in a Master-Worker cluster configuration to ensure distributed processing capabilities.

- **Structured Streaming:** Spark treats the live data stream as a table that is being continuously appended to. This allows us to express complex streaming computations using standard SQL-like semantics.
- **Fault Tolerance:** Spark utilizes checkpointing stored in MinIO. This records the read offsets from Kafka. In the event of a failure, Spark restarts and resumes processing exactly where it left off, guaranteeing exactly-once processing semantics.

3.2.3 Storage Layer: The Lambda Architecture

To satisfy conflicting requirements for speed and volume, the system splits the processed stream into two paths:

- **Cold Path (Analytics):** Data is written to **MinIO** (an S3-compatible object store) using the **Apache Iceberg** table format. Iceberg provides ACID transactions, preventing readers from seeing partial writes. The data is partitioned by 'City' and stored as Parquet files, which are highly compressed and optimized for analytical queries.
- **Hot Path (Serving):** Data is written to **MongoDB**, a NoSQL document store. Spark uses an "Upsert" mechanism here. For every city, only the most recent weather record is kept. This allows downstream applications to query the current status of any city with sub-millisecond latency. The collection structure is `weather_db.current_weather`.

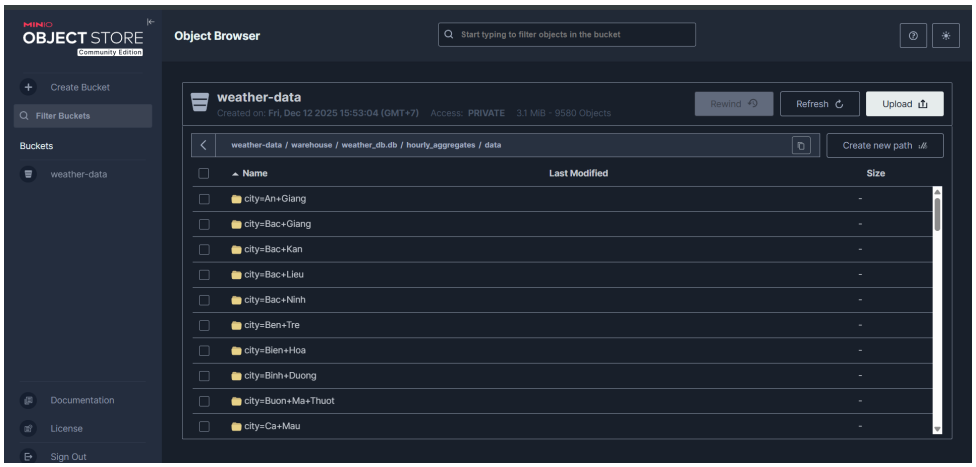


Figure 3.3: Cold Storage Layer Implementation

3.2.4 Metadata & Serving Layer

- **Hive Metastore:** Backed by a PostgreSQL database, this component tracks the location and schema of the Iceberg tables stored in MinIO. It acts as the bridge between the storage (MinIO) and the query engine (Trino).
- **Trino (formerly PrestoSQL):** A distributed SQL query engine. It connects to the Hive Metastore to locate data and then reads Parquet files directly from MinIO. This allows users to run complex ANSI SQL queries (JOINs, aggregations) on the data lake without moving the data.
- **Apache Superset:** A modern, enterprise-ready Business Intelligence (BI) web application.
  - **Visualization:** Connects to Trino via the SQLAlchemy driver to visualize historical trends.
  - **SQL Lab:** Provides a rich SQL IDE for data analysts to explore raw data, debug queries, and preview datasets before building charts.
  - **Semantic Layer:** Allows defining metrics (e.g., "Avg Temp") and calculated columns virtually, simplifying chart creation for non-technical users.

Chapter 4

Implementation Details

4.1 Spark Processing Logic

The application logic is encapsulated in `spark_streaming_app.py`. The following sections detail the critical technical decisions made during implementation.

4.1.1 Schema Enforcement and Data Quality

Streaming data is often messy. To prevent pipeline crashes, we enforce a strict schema at the ingestion point. We also implemented a robust strategy to handle missing timestamps. Since the API sometimes returns null values for the event time, we use a 'coalesce' strategy: prioritize the API timestamp, but fallback to the Kafka ingestion time if necessary.

```
# Robust timestamp extraction strategy
weather_df = weather_df.withColumn(
    "event_timestamp",
    coalesce(
        to_timestamp(col("timestamp")),          # Priority 1: ISO String
        to_timestamp(from_unixtime(col("dt"))),  # Priority 2: Unix Epoch
        col("kafka_timestamp")                   # Priority 3: Ingestion Time
    )
)
```

Listing 4.1: Timestamp Handling with Coalesce

4.1.2 Custom Business Logic (UDF)

To make the data more consumable for end-users, we implemented a Python User Defined Function (UDF) to calculate a "Comfort Index." This logic transforms raw temperature and humidity numbers into human-readable categories like "Scorching," "Muggy," or "Comfortable."

```
def determine_comfort_level(temp, humidity):
    if temp is None or humidity is None: return "Unknown"
    # Business logic for weather classification
    if temp >= 35: return "Scorching"
    elif temp >= 30 and humidity > 70: return "Muggy"
```

```
    elif temp < 15: return "Chilly"
    elif 20 <= temp < 30: return "Comfortable"
    else: return "Moderate"

# Registering the function for Spark SQL
comfort_udf = udf(determine_comfort_level, StringType())
```

Listing 4.2: Comfort Index UDF Implementation

4.1.3 Windowing and Watermarking

For the analytics path, raw data is aggregated into 1-hour tumbling windows. To handle the distributed nature of the system where data might arrive out of order, we apply a 10-minute watermark. This instructs Spark to keep the window state open for 10 minutes past the window end time, allowing late data to be included in the aggregation before finalizing the result file.

```
hourly_agg = weather_df \
    .withWatermark("event_timestamp", "10 minutes") \
    .groupBy(
        window(col("event_timestamp"), "1 hour"),
        col("city"), col("country")
    ) \
    .agg(
        avg("temperature").alias("avg_temperature"),
        max("temperature").alias("max_temperature"),
        min("temperature").alias("min_temperature"),
        avg("humidity").alias("avg_humidity"),
        count("*").alias("record_count")
    )
```

Listing 4.3: Windowed Aggregation Logic

4.2 Infrastructure Automation

To ensure reliability, we implemented an "Infrastructure as Code" approach for the Hive Metastore and Iceberg catalog. A dedicated initialization container (`init-iceberg`) runs a specialized script (`init_warehouse.py`) upon system startup. This script connects to Hive, cleans up any stale metadata, and ensures the database schema exists before the Spark streaming job attempts to write data. This prevents race conditions and "Table Not Found" errors.

Chapter 5

Results and Visualization

5.1 Data Lake Storage (Iceberg)

The system successfully writes data to MinIO. The data is organized in a hierarchical structure: `warehouse/weather_db.db/hourly_aggregates/data/`. Inside, data is physically partitioned by city (e.g., `city=Hanoi`), which optimizes read performance for location-based queries.

5.2 Query Performance (Trino)

Trino serves as the interactive query layer. We can execute complex SQL queries across the distributed dataset with low latency. Below is an example query used to analyze daily average temperatures from the aggregated hourly data.

```
trino> SELECT * from iceberg.weather_db.hourly_aggregates order by window_start;
       window_start        |         window_end         |      city      | country |  avg_temperature   | max_temperature | min_temperature
---------------------------+----------------------------+----------------+---------+--------------------+-----------------+-----------------
2025-12-03 00:00:00.000000 UTC | 2025-12-03 01:00:00.000000 UTC | Turan          | VN      |              22.99 |           22.99 |           22.99
2025-12-03 00:00:00.000000 UTC | 2025-12-03 01:00:00.000000 UTC | Hue            | VN      |              23.06 |           23.06 |           23.06
2025-12-03 00:00:00.000000 UTC | 2025-12-03 01:00:00.000000 UTC | Can Tho        | VN      |              25.99 |           25.99 |           25.99
2025-12-03 00:00:00.000000 UTC | 2025-12-03 01:00:00.000000 UTC | Ho Chi Minh City | VN    |              27.01 |           27.01 |           27.01
2025-12-03 00:00:00.000000 UTC | 2025-12-03 01:00:00.000000 UTC | Hanoi          | VN      |               22.0 |            22.0 |            22.0
2025-12-03 01:00:00.000000 UTC | 2025-12-03 02:00:00.000000 UTC | Hue            | VN      |              23.06 |           23.06 |           23.06
2025-12-03 01:00:00.000000 UTC | 2025-12-03 02:00:00.000000 UTC | Turan          | VN      | 23.101111111111113 |           23.99 |           22.99
2025-12-03 01:00:00.000000 UTC | 2025-12-03 02:00:00.000000 UTC | Hanoi          | VN      |               22.0 |            22.0 |            22.0
2025-12-03 01:00:00.000000 UTC | 2025-12-03 02:00:00.000000 UTC | Can Tho        | VN      | 28.101111111111113 |           28.99 |           27.99
2025-12-03 01:00:00.000000 UTC | 2025-12-03 02:00:00.000000 UTC | Ho Chi Minh City | VN    |  27.88111111111111 |           28.37 |           27.82
```

Figure 5.1: Trino Query Execution

```sql
SELECT
    city,
    date_trunc('day', window_start) as report_date,
    avg(avg_temperature) as daily_avg_temp,
    max(max_temperature) as daily_max_temp
FROM iceberg.weather_db.hourly_aggregates
GROUP BY 1, 2
ORDER BY report_date DESC;
```

Listing 5.1: Sample Analytical Query

5.3 Real-time Dashboard (Superset)

Apache Superset connects to Trino to visualize the data. We have successfully created a dashboard that provides a comprehensive view of the weather data.

The dashboard includes:

- **Time-Series Charts:** Line charts showing temperature and humidity trends over time for selected cities.
- **Country Map:** A geospatial map of Vietnam visualizing weather conditions across different provinces.
- **Gauge Charts:** Speedometer-style gauges displaying current humidity levels in percentage.
- **Latest Data Table:** A dynamic table fetching data from the MongoDB "Hot Path", showing the absolute latest weather metrics and comfort indices for each city to enable immediate operational awareness.

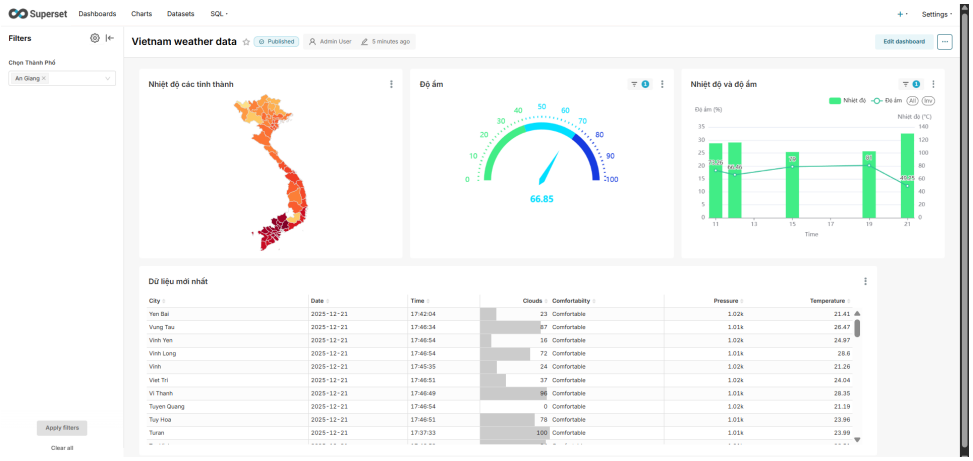


Figure 5.2: Real-time Weather Dashboard

Chapter 6

Conclusion

This project successfully demonstrates the construction of a modern, scalable Data Lakehouse using entirely open-source technologies. By implementing the **Lambda Architecture**, the system effectively balances the needs for real-time operational monitoring (via MongoDB) and deep historical analysis (via Iceberg).

The use of Docker for containerization ensures that the complex ecosystem of services (NiFi, Kafka, Spark, Hive, Trino, Superset) operates cohesively and can be easily deployed on any environment. The solutions implemented for data quality (Timestamp Fallback) and system reliability (Distributed Checkpointing) prove the robustness of the pipeline for production-grade scenarios. This architecture serves as a solid foundation for future expansions, such as integrating Machine Learning models for weather prediction or expanding the ingestion layer to IoT sensor networks.