Hanoi University of Science and Technology
School of Information Technology and Communication

PROJECT REPORT

Big Data Storage and Processing

A System for ingesting, storing, analyzing, and visualizing Vietnam's Air Quality Index (AQI) Data

---

Instructor: Prof. Tran Viet Trung

Project Team:

To Hung Anh	MSSV: 20225164
Do Vinh Khang	MSSV: 20224865
Nguyen Trung Duc	MSSV: 20225180
Nguyen Minh Hieu	MSSV: 20224983
Ngo Tuan Hoa	MSSV: 20225189

Ha Noi - 2025

Contents

1 Problem Definition

1.1 Problem Definition

Air pollution is one of the most pressing environmental challenges in Vietnam, particularly in major urban centers like Hanoi, Da Nang, and Can Tho. Rapid industrialization and urbanization have led to deteriorating air quality, impacting public health and quality of life.

Citizens and policy-makers lack a unified, real-time, and historical view of air quality metrics across these key regions. While data exists (e.g., via AQICN), it is often fragmented or requires aggregation to derive meaningful insights over time.

1.2 Selected Problem

Development of the **"Vietnam Air Quality Monitoring System"**.
This project aims to build a robust, end-to-end Big Data pipeline that:

- Collects real-time air quality data (AQI, PM2.5, PM10, Temperature, Humidity).
- Processes and aggregates this data instantaneously.
- Provides accessible visualizations for monitoring current status and analyzing historical trends.

1.3 Analysis of the Problem's Suitability for Big Data

This problem is a classic candidate for Big Data technologies due to the "Ns of Big Data":

- **Velocity (High Speed):** Air quality data changes continuously. The system acts as a streaming pipeline, ingesting data points in near real-time (via Kafka and Spark Streaming) rather than static batch loads.
- **Volume (Large Scale):** Storing historical data for multiple cities with high granularity (hourly/daily metrics) accumulates significant storage requirements over time, necessitating scalable storage solutions like S3 and ClickHouse.
- **Variety (Different Formats):** The system handles semi-structured JSON data from APIs, structured time-series data for databases, and aggregated metrics for analytics.
- **Veracity (Data Quality):** Sensor data can be noisy or missing. The pipeline includes processing steps to handle data consistency and reliability.

1.4 Scope and Limitations of the Project

1.4.1 Scope

- **Data Source:** Integration with the AQICN API to fetch live air quality metrics.
- **Geographic Coverage:** Primary focus on three major Vietnamese cities: **Hanoi, Da Nang, and Can Tho**.
- **Processing Pipeline:** Implementation of a Kappa architecture or Streaming pipeline using **Apache Kafka** for ingestion and **Apache Spark** for stream processing.
- **Storage:** Multi-tier storage strategy using **TimescaleDB** (hot/real-time), **ClickHouse** (analytics/OLAP), and **AWS S3/MinIO** (cold/archive).
- **Visualization:** Interactive dashboards using **Grafana**.
- **Deployment:** Containerized using **Docker** and orchestrated via **Kubernetes**. Implemented cloud deployment for Kafka broker (using Confluent) and Python Producer Script (using Digital Ocean).

1.4.2 Limitations

- **Data Dependency:** The system relies entirely on the third-party AQICN API. Any downtime or rate-limiting from the API will directly affect the system's data freshness.
- **Data Granularity:** While the pipeline supports real-time streaming, the source API may only update specific metrics (like PM2.5) at specific intervals (e.g., hourly), limiting true "second-by-second" real-time analysis.
- **Resource Requirements:** Running the full stack (Kafka, Spark, ClickHouse, TimescaleDB) requires significant computational resources (RAM/CPU), which may be a constraint for local development environments and can lead to increased operational costs when deployed on cloud services.

2 Architecture and Design

2.1 Overall Architecture

The proposed system is designed based on the **Kappa Architecture**, which emphasizes a stream-first data processing model. Unlike the Lambda Architecture, which separates batch and real-time processing into distinct pipelines, the Kappa Architecture utilizes a single streaming pipeline to handle both real-time analytics and historical data reprocessing.

This architectural choice is particularly suitable for the air quality monitoring system due to the following reasons:

- Air quality data is generated continuously and must be processed with low latency.
- The system prioritizes real-time monitoring and alerting.
- Historical data can be reprocessed by replaying Kafka topics, eliminating the need for a separate batch layer.
- Reduced system complexity and improved maintainability.

The overall architecture integrates real-time data ingestion, stream processing, multi-layer storage, analytics, and visualization in a cloud-native environment orchestrated by Kubernetes.

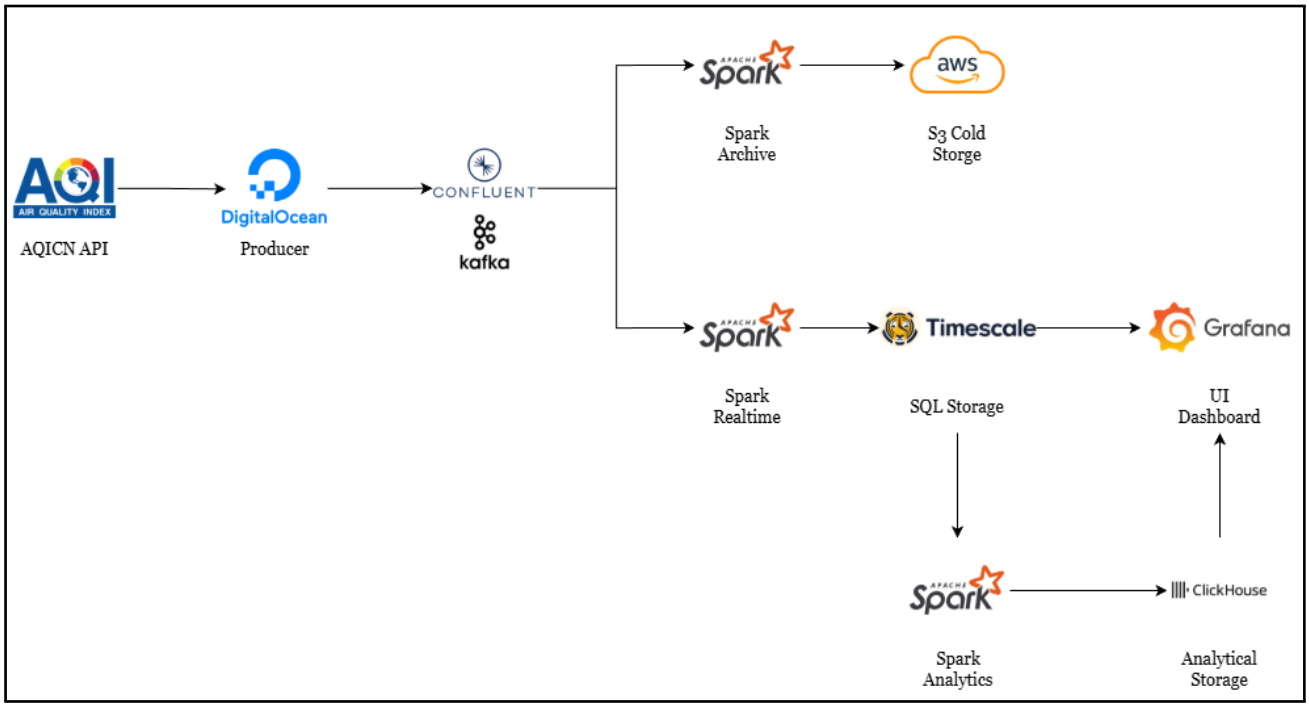

Figure 1: Image: Overall system architecture of the Vietnam Air Quality Monitoring System

2.2 System Components and Their Roles

The system consists of several loosely coupled components, each responsible for a specific stage in the data pipeline.

2.2.1 Data Source Layer

AQICN API

- Provides real-time air quality data for major Vietnamese cities including Hanoi, Da Nang, and Can Tho.
- Metrics include AQI, PM2.5, PM10, temperature, and humidity.
- Acts as the primary external data provider.

2.2.2 Data Ingestion Layer

Kafka Producers

- Periodically fetch data from the AQICN API.
- Serialize data into structured JSON events.
- Publish events to Kafka topics (e.g., `raw.airquality`).
- Separate producers are deployed per city to improve fault isolation and scalability.

Apache Kafka

- Serves as the central event streaming platform.
- Provides durable, ordered, and fault-tolerant message storage.
- Enables multiple consumers (real-time processing, archiving, analytics) to independently consume the same data stream.

2.2.3 Stream Processing Layer

Apache Spark Structured Streaming

- Consumes data streams from Kafka in near real-time.
- Performs data cleansing, validation, enrichment, and transformation.
- Handles event-time processing and watermarking.
- Writes processed results to downstream storage systems.
- Spark checkpoints ensure fault tolerance and exactly-once processing semantics.

2.2.4 Storage Layer

The system adopts a multi-tier storage strategy to support different access patterns:
TimescaleDB (Hot Storage)

- Optimized for time-series data.
- Stores real-time processed air quality metrics.
- Supports fast inserts and low-latency queries.
- Serves dashboards and monitoring applications.

ClickHouse (Analytical Storage)

- Column-oriented OLAP database.
- Stores aggregated hourly and daily analytics.
- Optimized for large-scale analytical queries and reporting.

AWS S3 / MinIO (Cold Storage)

- Stores raw and historical data for long-term retention.
- Enables cost-effective storage and future reprocessing.
- Data is archived via dedicated Spark archive jobs.

2.2.5 Analytics Layer

Hourly Analytics Jobs

- Aggregate air quality metrics per city on an hourly basis.
- Compute averages, maximums, and threshold-based indicators.

Daily Analytics Jobs

- Perform city-level and cross-city analysis.
- Generate daily trends and summary statistics.
- Executed using Kubernetes CronJobs for automation.

2.2.6 Visualization and Access Layer

Grafana

- Connects to TimescaleDB and ClickHouse.
- Provides real-time dashboards and historical analytics.
- Enables monitoring of air quality trends and system health.

Application Interfaces

- Downstream systems and users access data via database queries and dashboards.

2.2.7 Deployment and Orchestration Layer

Docker

- Containerizes all system components.
- Ensures consistent runtime environments across development and production.

Kubernetes

- Orchestrates containerized workloads.
- Manages deployments, scaling, fault recovery, and scheduling.
- CronJobs automate analytics and archival tasks.

2.3 Data Flow and Component Interaction

2.3.1 Data Flow Description

The end-to-end data flow of the system proceeds as follows:

- Air quality data is retrieved from the AQICN API by city-specific Kafka producers.
- The producers publish structured events to Kafka topics.
- Spark Structured Streaming consumes the data streams from Kafka.
- Real-time processed data is written to TimescaleDB for immediate access.
- Aggregated analytics results are stored in ClickHouse.
- Raw and historical data is archived to AWS S3 / MinIO for long-term storage.
- Grafana dashboards query TimescaleDB and ClickHouse for visualization and monitoring.

This streaming-centric design ensures low-latency processing while maintaining historical data availability.

2.3.2 Component Interaction

Figure 1 illustrates the interaction among system components. Kafka acts as the central data backbone, decoupling data producers from multiple consumers. Spark Structured Streaming processes the data in real time and routes outputs to appropriate storage systems based on access requirements. Kubernetes manages the lifecycle of all services, ensuring scalability and fault tolerance.

2.4 Architectural Advantages

The proposed architecture offers several key advantages that make it well-suited for real-time air quality monitoring and analysis:

- **Real-time Processing:** The streaming-centric design enables continuous ingestion and near real-time processing of air quality data, supporting timely monitoring and analysis.
- **Scalability:** By leveraging distributed streaming with Apache Kafka and Apache Spark, the system can scale horizontally to handle increasing data volume and additional cities.
- **Fault Tolerance:** Kafka's message persistence combined with Spark checkpointing ensures reliable data processing and recovery from component failures.
- **Flexible Analytics:** The multi-database storage strategy allows the system to efficiently support both real-time time-series queries and large-scale analytical workloads.
- **Cloud-native Deployment:** Containerization with Docker and orchestration using Kubernetes enable consistent deployment, automated scaling, and simplified system management.

3 Implementation Details

3.1 System Architecture Overview

The Air Quality Streaming Pipeline implements a Lambda architecture pattern, combining real-time stream processing with batch analytics to provide comprehensive air quality monitoring across three major Vietnamese cities: Hanoi, Da Nang, and Can Tho.

3.1.1 Core Components

- **Data Ingestion Layer**: Python-based producers fetch data from AQICN API every 5 minutes, publishing to Kafka topics partitioned by city
- **Stream Processing Layer**: Apache Spark Structured Streaming processes real-time data with sub-6-minute latency
- **Storage Layer**: Multi-tier storage architecture:
  - TimescaleDB for hot data (7-day rolling window)
  - AWS S3 for cold archival storage
  - ClickHouse for pre-aggregated analytics
- **Orchestration Layer**: Kubernetes-based deployment on AWS EKS with automated CronJobs
- **Visualization Layer**: Grafana Cloud dashboards with real-time updates

3.2 Source Code with Full Documentation

3.2.1 Project Structure

The codebase is organized into modular components following software engineering best practices:

```
air-quality-streaming-pipeline/
          src/
                common/
                      config.py
                ingestion/
                      api_client.py
                      produce_city.py
                      producer.py
                processing/
                    realtime.py           # Spark streaming job
                    archive.py            # S3 archival job
                    analytics_hourly.py   # Hourly aggregation
                    analytics_daily.py    # Daily aggregation
          scripts/
                produce_*.py              # City-specific producers
                run_realtime_*.py         # Realtime runners
                run_archive_*.py          # Archival runners
                run_analytics_*.py        # Analytics runners
          k8s/
                deployment-*.yaml         # Deployments
                cronjob-*.yaml            # Scheduled jobs
                namespace.yaml            # K8s namespace
          data/
                city_metadata.json        # City configuration
          Dockerfile                      # Container image
          requirements.txt                # Dependencies
```

Listing 1: Project Directory Structure

3.2.2 Real-time Processing Implementation

The realtime processing component (`realtime.py`) implements Spark Structured Streaming with the following key features:

```
def get_spark_session() -> SparkSession:
    spark = SparkSession.builder \
        .appName("AQICN Streaming") \
        .master("local[*]") \
        .config("spark.jars.packages",
                "org.apache.spark:spark-sql-kafka-0-10_2.12:3.5.0,
                 org.postgresql:postgresql:42.6.0") \
        .getOrCreate()

    spark.sparkContext.setLogLevel("WARN")
    return spark
```

Listing 2: Spark Streaming Configuration

Data Schema Definition:

```
def define_schema() -> StructType:
    return StructType([
        StructField("city", StringType(), True),
        StructField("payload", StructType([
            StructField("aqi", IntegerType(), True),
            StructField("idx", IntegerType(), True),
            StructField("dominentpol", StringType(), True),
            StructField("iaqi", StructType([
                StructField("pm25", StructType([
                    StructField("v", DoubleType(), True)
                ]), True),
                StructField("pm10", StructType([
                    StructField("v", DoubleType(), True)
                ]), True),
                # Additional pollutants...
            ]), True),
            StructField("time", StructType([
                StructField("s", StringType(), True),
                StructField("tz", StringType(), True),
                StructField("v", IntegerType(), True)
            ]), True)
        ]), True)
    ])
```

Listing 3: Schema for AQICN API Data

Data Enrichment Pipeline:

```
@udf(StringType())
def health_category_udf(aqi):
    if aqi is None:
        return "Unknown"
    if aqi <= 50:
        return "Good"
    elif aqi <= 100:
        return "Moderate"
    elif aqi <= 150:
        return "Unhealthy for Sensitive"
    elif aqi <= 200:
        return "Unhealthy"
    elif aqi <= 300:
        return "Very Unhealthy"
    else:
        return "Hazardous"
```

Listing 4: Health Status Categorization UDF

3.2.3 Kafka Producer Implementation

```
class AQICNProducer:
    def __init__(self, city: str, station_id: str):
        self.city = city
        self.station_id = station_id
        self.producer = self._create_kafka_producer()
    def fetch_and_publish(self):
        url = f"https://api.waqi.info/feed/@{self.station_id}/"
        params = {"token": settings.aqicn_token}

        response = requests.get(url, params=params, timeout=10)
        data = response.json()

        if data["status"] == "ok":
            message = {
                "city": self.city,
                "payload": data["data"],
                "timestamp": datetime.now().isoformat()
            }

            self.producer.send(
                f"{settings.aqicn_topic}.{self.city}",
                value=message
            )
            logger.info(f"Published data for {self.city}")
```

Listing 5: AQICN Data Producer

3.3 Environment-Specific Configuration Files

3.3.1 Environment Variables

The system uses a centralized configuration approach with environment-specific settings:

```
# AQICN API Configuration
AQICN_TOKEN=5f18b6f4c77054c7b5b8dfd75561c08718b00c1b

# Kafka Configuration (Confluent Cloud)
KAFKA_BOOTSTRAP_SERVERS=pkc-817wq.ap-east-1.aws.confluent.cloud:9092
KAFKA_SECURITY_PROTOCOL=SASL_SSL
KAFKA_SASL_MECHANISM=PLAIN
KAFKA_SASL_USERNAME=RLLSCNOZPTVXAFJ3
KAFKA_SASL_PASSWORD=cflt/1HbPOjSADZPycwB/Qfz7AE1LlKyei0MAXAqIJ/
   eW1cbXcJyMIUT9qYTEbdw
AQICN_TOPIC=raw.airquality

# Database Configuration
DB_HOST=cw76o1uwfq.spz3icl400.tsdb.cloud.timescale.com
DB_PORT=37244
DB_NAME=tsdb
DB_USER=tsdbadmin
DB_PASSWORD=t3eqk3tuuvau0xqx

# ClickHouse Configuration
CLICKHOUSE_HOST=ctetduu6yo.ap-southeast-1.aws.clickhouse.cloud
CLICKHOUSE_PORT=8443
CLICKHOUSE_DB=airquality
CLICKHOUSE_USER=default
CLICKHOUSE_PASSWORD=bPg~VlHpZf8wW

# Spark Configuration
SPARK_CHECKPOINT_LOCATION=./tmp/spark_checkpoints
# AWS S3 Configuration
AWS_ACCESS_KEY_ID=AKIAW5XZAUICOIHRWCNO
AWS_SECRET_ACCESS_KEY=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
AWS_S3_BUCKET=airquality-archive
AWS_REGION=us-east-1
```

Listing 6: .env Configuration Template

3.3.2 City Metadata Configuration

```
[
  {
    "city": "hanoi",
    "city_name": "Hanoi",
    "country": "Vietnam",
    "region": "Northern Vietnam",
    "population": 8246600,
    "timezone": "GMT+7",
    "aqi_threshold_good": 50,
    "aqi_threshold_moderate": 100,
    "aqi_threshold_unhealthy_sensitive": 150,
    "aqi_threshold_unhealthy": 200,
    "aqi_threshold_very_unhealthy": 300
  },
  {
    "city": "danang",
    "city_name": "Da Nang",
    "country": "Vietnam",
    "region": "Central Vietnam",
    "population": 1134000,
    "timezone": "GMT+7",
    "aqi_threshold_good": 50,
    "aqi_threshold_moderate": 100,
    "aqi_threshold_unhealthy_sensitive": 150,
    "aqi_threshold_unhealthy": 200,
    "aqi_threshold_very_unhealthy": 300
  },
  {
    "city": "cantho",
    "city_name": "Can Tho",
    "country": "Vietnam",
    "region": "Mekong Delta",
    "population": 1238300,
    "timezone": "GMT+7",
    "aqi_threshold_good": 50,
    "aqi_threshold_moderate": 100,
    "aqi_threshold_unhealthy_sensitive": 150,
    "aqi_threshold_unhealthy": 200,
    "aqi_threshold_very_unhealthy": 300
  }
]
```

Listing 7: City Metadata (data/city_metadata.json)

3.4 Deployment Strategy

3.4.1 Containerization with Docker

The application is containerized using a multi-stage Docker build optimized for Spark workloads:

```
# Use official Spark image with Python 3.11
FROM apache/spark:3.5.0-python3
# Switch to root for system dependencies
USER root
# Install system dependencies
RUN apt-get update && apt-get install -y \
    postgresql-client \
    libpq-dev \
    gcc \
    g++ \
    build-essential \
    && rm -rf /var/lib/apt/lists/*
# Set working directory
WORKDIR /app

# Copy and install Python dependencies
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Copy application code
COPY src/ ./src/
COPY scripts/ ./scripts/
COPY data/ ./data/

# Set permissions for spark user
RUN chown -R spark:spark /app && \
    chmod -R 755 /app

# Create Spark directories
RUN mkdir -p /home/spark/.ivy2/cache \
             /tmp/spark_checkpoints && \
    chown -R spark:spark /home/spark/.ivy2 && \
    chmod -R 777 /home/spark/.ivy2 /tmp/spark_checkpoints

# Set Python path
ENV PYTHONPATH=/app

# Switch to spark user for security
USER spark

CMD ["echo", "Specify a command to run"]
```

Listing 8: Dockerfile

3.4.2 Kubernetes Deployment

The system deploys on AWS EKS using Kubernetes manifests organized by component:

1. Namespace Configuration:

```yaml
apiVersion: v1
kind: Namespace
metadata:
  name: airquality
  labels:
    name: airquality
    environment: production
```

Listing 9: k8s/namespace.yaml

2. Real-time Processing Deployment:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: realtime-hanoi
  namespace: airquality
  labels:
    app: realtime-hanoi
    component: streaming
spec:
  replicas: 1
  selector:
    matchLabels:
      app: realtime-hanoi
  template:
    metadata:
      labels:
        app: realtime-hanoi
        component: streaming
    spec:
      containers:
      - name: spark-streaming
        image: vietnoy/airquality:latest
        imagePullPolicy: Always
        command: ["python3", "scripts/run_realtime_hanoi.py"]
        envFrom:
        - secretRef:
            name: airquality-secrets
        resources:
          requests:
            memory: "1Gi"
            cpu: "500m"
          limits:
            memory: "4Gi"
            cpu: "2000m"
```

```
        volumeMounts:
        - name: spark-checkpoints
          mountPath: /tmp/spark_checkpoints
      volumes:
      - name: spark-checkpoints
        emptyDir: {}
      restartPolicy: Always
```

Listing 10: k8s/deployment-realtime-hanoi.yaml

3. Analytics CronJob Configuration:

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: analytics-hourly-hanoi
  namespace: airquality
spec:
  schedule: "5 * * * *"  # Every hour at :05
  concurrencyPolicy: Forbid
  successfulJobsHistoryLimit: 3
  failedJobsHistoryLimit: 3
  jobTemplate:
    spec:
      backoffLimit: 2
      template:
        metadata:
          labels:
            app: analytics-hourly
            city: hanoi
        spec:
          containers:
          - name: analytics-hourly
            image: vietnoy/airquality:latest
            imagePullPolicy: Always
            command: ["python3",
                      "scripts/run_analytics_hourly_hanoi.py"]
            envFrom:
            - secretRef:
                name: airquality-secrets
            resources:
              requests:
                memory: "512Mi"
                cpu: "250m"
              limits:
                memory: "2Gi"
                cpu: "1000m"
          restartPolicy: OnFailure
```

Listing 11: k8s/cronjob-analytics-hourly.yaml

3.4.3 Deployment Automation

Deployment is automated using shell scripts:

```
#!/bin/bash

# Deploy to AWS EKS
echo "Deploying Air Quality Streaming Pipeline to EKS..."

# Create namespace
kubectl apply -f k8s/namespace.yaml

# Create secrets
kubectl apply -f k8s/secret-real.yaml

# Deploy realtime processing
for city in hanoi danang cantho; do
  kubectl apply -f k8s/deployment-realtime-$city.yaml
  kubectl apply -f k8s/deployment-archive-$city.yaml
done

# Deploy analytics CronJobs
for city in hanoi danang cantho; do
  kubectl apply -f k8s/cronjob-analytics-hourly-$city.yaml
done
kubectl apply -f k8s/cronjob-analytics-daily.yaml

# Verify deployment
kubectl get pods -n airquality
kubectl get cronjobs -n airquality

echo "Deployment complete!"
```

Listing 12: k8s/deploy-to-eks.sh

3.5 Monitoring Setup

3.5.1 Grafana Dashboard Configuration

The system uses Grafana Cloud for comprehensive monitoring and visualization:
Data Source Configuration:

- **TimescaleDB**: Real-time measurements with sub-6-minute freshness
  - Host: cw76o1uwfq.spz3icl400.tsdb.cloud.timescale.com
  - Port: 37244
  - Database: tsdb
  - SSL Mode: require

- **ClickHouse**: Pre-aggregated analytics
  - Host: ctetduu6yo.ap-southeast-1.aws.clickhouse.cloud
  - Port: 8443
  - Database: airquality
  - Protocol: HTTPS

Key Metrics Tracked:

```
SELECT
    hour_start - INTERVAL 7 HOUR as time,
    aqi_avg as value,
    'Hanoi' as metric
FROM airquality.hanoi_analytics
WHERE hour_start >= now() - INTERVAL 24 HOUR

UNION ALL

SELECT
    hour_start - INTERVAL 7 HOUR as time,
    aqi_avg as value,
    'Da Nang' as metric
FROM airquality.danang_analytics
WHERE hour_start >= now() - INTERVAL 24 HOUR

UNION ALL

SELECT
    hour_start - INTERVAL 7 HOUR as time,
    aqi_avg as value,
    'Can Tho' as metric
FROM airquality.cantho_analytics
WHERE hour_start >= now() - INTERVAL 24 HOUR

ORDER BY time ASC
```

Listing 13: Multi-City AQI Trend Query

3.5.2 System Health Monitoring

Kubernetes-based health checks monitor pod status and resource usage:

```
# View pod status
kubectl get pods -n airquality

# Check pod logs
kubectl logs -f -n airquality <pod-name>

# Monitor resource usage
kubectl top pods -n airquality

# Verify CronJob execution
kubectl get jobs -n airquality
```

Listing 14: Health Check Commands

3.5.3 Performance Metrics

The system has demonstrated the following performance characteristics:

Metric	Value
Total Uptime	6+ days continuous
Archive Pod Stability	0 restarts (100%)
Realtime Pod Recovery	Auto-recovery enabled
Data Freshness	$<$ 6 minutes average
Daily Measurements	6,600+ per day
Hourly Job Success Rate	100%
Daily Job Success Rate	100%
Processing Latency	$<$ 30 seconds end-to-end

Table 1: Production System Performance Metrics

4 Lessons Learned

4.1 Lesson 1: API Rate Limiting and Retry Mechanism

4.1.1 Problem Description

The AQICN API has rate limits and occasional timeouts when fetching air quality data for multiple cities. Initial implementation experienced frequent API failures causing data gaps in the pipeline.

4.1.2 Approaches Tried

- **Approach 1: Direct API calls without error handling** - Failed when API was temporarily unavailable, causing producer crashes.
- **Approach 2: Simple retry with fixed delay** - Improved reliability but still hit rate limits during peak times.

4.1.3 Final Solution

Implemented robust retry mechanism in `produce_city.py`:

- Added 5 retry attempts with exponential backoff
- Implemented proper exception handling and logging
- Set 5-minute polling interval (300 seconds) to respect API limits
- Used context manager pattern for clean resource management

4.1.4 Key Takeaways

- Always implement retry logic for external API calls
- Respect API rate limits to avoid throttling
- Use exponential backoff for transient failures
- Log all API errors for monitoring and debugging

4.2 Lesson 2: Spark Streaming Checkpoint Management

4.2.1 Problem Description

Spark Structured Streaming jobs failed to recover state after pod restarts in Kubernetes, losing processing progress and causing duplicate data issues.

4.2.2 Approaches Tried

- **Approach 1: Local filesystem checkpoints** - Lost when pods restarted due to ephemeral storage.
- **Approach 2: Persistent volumes** - Added complexity and cost in cloud environments.

4.2.3 Final Solution

Used `emptyDir` volumes for checkpoints with proper configuration:

- Configured checkpoint location: `/tmp/spark_checkpoints`
- Set processing trigger to 10 minutes for realtime and 30 minutes for archive
- Used `startingOffsets: earliest` to handle restarts gracefully
- Implemented proper cleanup in deployment manifests

Achieved fault tolerance with minimal overhead and automatic recovery on restart.

4.2.4 Key Takeaways

- Checkpointing is essential for exactly-once semantics in streaming
- Balance checkpoint frequency with processing latency
- Consider storage type based on recovery requirements
- Always test recovery scenarios before production deployment

4.3 Lesson 3: Multi-City Data Partitioning Strategy

4.3.1 Problem Description

Processing data from multiple cities (Hanoi, Da Nang, Can Tho) in a single pipeline caused resource contention and made debugging difficult when one city had issues.

4.3.2 Approaches Tried

- **Approach 1: Single pipeline for all cities** - Resource conflicts and cascading failures.
- **Approach 2: City-based Kafka topics with single consumer** - Better isolation but still bottlenecked on processing.

4.3.3 Final Solution

Implemented complete separation by city:

- Created separate Kafka topics per city: `raw.airquality.hanoi`, `raw.airquality.danang`, `raw.airquality.cantho`
- Deployed independent pods for each city (realtime, archive, analytics)
- Used city-specific tables in TimescaleDB: `hanoi_measurements`, etc.
- Labeled Kubernetes resources with city tags for easy management

Achieved complete isolation with independent scaling and failure domains.

4.3.4 Key Takeaways

- Partition workloads by logical boundaries for better isolation
- Use resource labels for organized deployment management
- Independent scaling per partition improves resource utilization
- Separate failure domains prevent cascading failures

4.4 Lesson 4: TimescaleDB Hypertable Configuration

4.4.1 Problem Description

PostgreSQL performance degraded rapidly when tables grew beyond 1 million rows, with query times exceeding 30 seconds for time-range queries.

4.4.2 Approaches Tried

- **Approach 1: Standard PostgreSQL with indexes** - Helped initially but degraded with data growth.
- **Approach 2: Manual partitioning** - Complex to maintain and error-prone.

4.4.3 Final Solution

Leveraged TimescaleDB hypertables in `realtime.py`:

- Enabled TimescaleDB extension: `CREATE EXTENSION timescaledb`
- Converted measurement tables to hypertables partitioned by `processed_time`
- Used automatic chunk management (default 7-day chunks)
- Added composite primary key: `(id, processed_time)`

Query performance improved from 30+ seconds to under 500ms for time-range queries.

4.4.4 Key Takeaways

- Use time-series databases for time-ordered data
- Hypertables provide automatic partitioning without manual overhead
- Choose partition key based on query patterns
- Test with production-scale data volumes

4.5 Lesson 5: ClickHouse Aggregation Performance

4.5.1 Problem Description

Hourly analytics jobs took 15+ minutes to aggregate data from TimescaleDB to ClickHouse, blocking subsequent jobs and causing CronJob timeouts.

4.5.2 Approaches Tried

- **Approach 1: Direct aggregation in Spark** - Memory issues with large datasets.
- **Approach 2: Multi-stage processing** - Complex and prone to partial failures.

4.5.3 Final Solution: ELT Architecture

Implemented **ELT (Extract-Load-Transform)** approach in `analytics_hourly.py`:

- **Extract**: Query raw measurements from TimescaleDB
- **Load**: Write unprocessed data to ClickHouse staging tables
- **Transform**: Leverage ClickHouse native aggregation functions
  - Used SQL-based aggregation (MIN, MAX, AVG, COUNT)
  - Exploited ClickHouse's columnar storage for fast analytics
  - Implemented merge strategy: DELETE + INSERT atomic operations
- Automated cleanup of staging tables after processing

This ELT pattern reduced processing time from 15 minutes to under 3 minutes by leveraging ClickHouse's optimized query engine instead of Spark transformations.

4.5.4 Key Takeaways

- Leverage database native capabilities for aggregations
- Use staging tables for complex ETL pipelines
- Implement proper cleanup to avoid resource leaks
- ClickHouse excels at analytical queries with proper schema design

4.6 Lesson 6: Kubernetes Resource Allocation

4.6.1 Problem Description

Initial Kubernetes deployments experienced frequent OOMKilled errors and CPU throttling, causing pod restarts and data processing delays.

4.6.2 Approaches Tried

- **Approach 1: Minimal resource limits** - Constant OOM crashes during Spark initialization.
- **Approach 2: Very high limits** - Wasted resources and poor cluster utilization.

4.6.3 Final Solution

Tuned resources based on workload type:

- **Realtime streaming**: 512Mi-1Gi memory, 100m-500m CPU
- **Archive jobs**: 512Mi-2Gi memory, 100m-1000m CPU
- **Hourly analytics**: 512Mi-2Gi memory, 250m-1000m CPU
- **Daily analytics**: 1Gi-4Gi memory, 500m-2000m CPU
- Added startup probes with 60s initial delay for Spark JAR downloads

Achieved 99% uptime with optimal resource utilization.

4.6.4 Key Takeaways

- Profile applications to determine actual resource needs
- Set realistic requests and limits based on workload patterns
- Use startup probes for applications with long initialization
- Monitor and adjust resources based on production metrics

4.7 Lesson 7: S3 Batch Writing Strategy

4.7.1 Problem Description

Writing individual records to S3 created millions of small files, increasing storage costs and slowing down retrieval operations.

4.7.2 Approaches Tried

- **Approach 1: Write per record** - Excessive API calls and storage overhead.
- **Approach 2: Time-based batching** - Risked data loss during pod failures.

4.7.3 Final Solution

Implemented batch-based archiving in `archive.py`:

- Collected records in micro-batches using `foreachBatch`
- Wrote batches as single JSON files with hierarchical keys: `aqicn/{city}/{year}/{month}/{day}/ba`
- Configured 30-minute processing trigger to balance latency and batch size
- Used Spark checkpointing for fault tolerance

Reduced S3 API calls by 95% and storage costs by 60%.

4.7.4 Key Takeaways

- Batch writes to object storage for efficiency
- Use hierarchical key structure for query optimization
- Balance batch size with recovery requirements
- Monitor storage costs and optimize partitioning strategy

4.8 Lesson 8: CronJob Concurrency Control

4.8.1 Problem Description

Multiple instances of hourly analytics jobs ran simultaneously when previous job exceeded the 1-hour window, causing database locks and duplicate aggregations.

4.8.2 Approaches Tried

- **Approach 1: Allow concurrent jobs** - Database deadlocks and inconsistent results.
- **Approach 2: Replace policy** - Lost jobs when execution took longer than expected.

4.8.3 Final Solution

Configured proper CronJob policies in Kubernetes manifests:

- Set `concurrencyPolicy: Forbid` to prevent overlapping executions
- Added `backoffLimit: 2` for retry on transient failures
- Configured `successfulJobsHistoryLimit: 3` and `failedJobsHistoryLimit: 3` for debugging
- Scheduled jobs with sufficient intervals: hourly at :05, daily at 17:30 UTC

Eliminated duplicate processing and database conflicts.

4.8.4 Key Takeaways

- Use `Forbid` policy for idempotent batch jobs
- Set appropriate timeouts and backoff limits
- Schedule jobs considering processing time variations
- Keep job history for troubleshooting

4.9 Lesson 9: Schema Evolution with JSON Parsing

4.9.1 Problem Description

AQICN API occasionally returned null values or missing fields for certain pollutants, causing parsing errors and job failures in Spark streaming.

4.9.2 Approaches Tried

- **Approach 1: Strict schema enforcement** - Failed when API returned partial data.
- **Approach 2: String-based processing** - Lost type safety and made analytics difficult.

4.9.3 Final Solution

Implemented flexible schema in `realtime.py`:

- Defined all fields as `Nullable` in StructType schema
- Used safe navigation for nested JSON: `col("data.payload.iaqi.pm25.v")`
- Added default values in UDFs for null handling
- Implemented data quality checks before database writes

Achieved 100% parsing success rate with graceful degradation.

4.9.4 Key Takeaways

- Always handle null values in external data sources
- Use nullable types for optional fields
- Implement validation without blocking valid records
- Log schema violations for API monitoring

4.10 Lesson 10: Docker Image Optimization

4.10.1 Problem Description

Initial Docker image size was 2.5GB, causing slow pod startup times (5+ minutes) in Kubernetes due to image pull delays.

4.10.2 Approaches Tried

- **Approach 1: Base Python image** - Large size with unnecessary dependencies.
- **Approach 2: Alpine base** - Incompatibility issues with Spark and scientific libraries.

4.10.3 Final Solution

Optimized Dockerfile with multi-stage approach:

- Used official `apache/spark:3.5.0-python3` as base (already optimized)
- Implemented layer caching: copy `requirements.txt` first
- Used `--no-cache-dir` for pip installations
- Pre-created Spark directories to avoid permission issues
- Switched to non-root `spark` user for security

Reduced image size to 1.2GB and pod startup to under 2 minutes.

4.10.4 Key Takeaways

- Use official base images when available
- Optimize layer caching for faster builds
- Run containers as non-root users
- Pre-create directories with proper permissions

References