Ministry of Education and Science of Ukraine

National University of "Kyiv-Mohyla Academy"

Department of Informatics of the Faculty of Informatics

Development of data pipeline for processing of streaming data on a cloud platform

Text part to the thesis in the specialty "Software Engineering" - F3

Thesis Supervisor:

Senior Lecturer

Dmytro Cherkasov

_____ (Signature)

"___" _____ 2026

Done by Student:

Volodymyr Materynskyi

"___" _____ 2026

Kyiv, 2026

Ministry of Education and Science of Ukraine
National University of "Kyiv-Mohyla Academy"
Department of Informatics of the Faculty of Informatics

Approved
Head of Department of Informatics,
Associate Professor, Ph.D.
S. S. Gorokhovskyi

_____ (Signature)

"___" _____ 2026

INDIVIDUAL TASK for thesis
of the student Volodymyr Materynskyi
4th year of the Faculty of Informatics
topic: Development of data pipeline for processing of streaming data on a cloud platform

The content of the PM to the thesis:
Abstract
Introduction
Related work
Approach
Solution
Evaluation
Further work and Conclusions
Bibliography

Date of issue: "___" _____ 2026

Supervisor: _____ (Signature)

Task received: _____ (Signature)

Research plan

Table 1: Thesis Work Schedule

No.	Thesis Stage	Deadline	Notes
1	Early analysis and research: defining scope and tools	02.11.2025	
2	Writing introduction to thesis	01.12.2025	
3	System design: creating high-level diagram of the application and dataflow	15.12.2025	
4	Implementation of designed system	05.01.2026	
5	Testing and optimization	02.02.2026	
6	Thesis writing	26.03.2026	
7	Review work and correct errors	01.04.2026	
8	Preliminary defense	03.04.2026	
9	Thesis defense	03.06.2026	

Student: Volodymyr Materynskyi
Thesis Supervisor: Dmytro Cherkasov
“____” _____ 2026

Abstract

Air quality monitoring is an important part of public health infrastructure worldwide, and in Ukraine it has become particularly important due to the combined impact of wartime fires and industrial emissions. Existing notification systems for Ukrainian cities operate on hourly readings with limited geographic coverage. This thesis presents the design, implementation, and evaluation of a streaming data pipeline that continuously ingests satellite readings from major Ukrainian cities, detects pollution events and visualises conditions on a live dashboard. The pipeline is organised as five loosely coupled stages spanning data ingestion, stream processing, aggregate storage, real-time visualisation, and email alerting. Load testing demonstrated sustained throughput of 414,000 records per minute with sub-minute latency, and an effective ceiling of approximately 1.79 million records per minute. The work contributes a reproducible reference implementation of streaming infrastructure for environmental monitoring, addressing the coverage and latency limitations of existing Ukrainian systems.

Table of Contents

Chapter 1

Introduction

1.1 Background

Air quality is one of the most important factors in human quality of life. Access to accurate and timely air quality data serves a broad range of purposes: individuals — particularly those with respiratory or cardiovascular conditions — can use it to make informed decisions about outdoor activity, health organizations can issue targeted public advisories during pollution events, and research institutions can incorporate it into environmental and epidemiological studies.

Figure 1.1: Satellite imagery of the Epicentr K warehouse in Chernihiv, Ukraine, ablaze following a Russian strike on 28 February 2022. Image credit: BlackSky [1].

Ukraine has historically faced air quality challenges attributable to two principal sources: emissions from heavy industrial facilities — which account for over 60% of total

national emissions [2] — and uncontrolled agricultural burning, including the burning of crop residues. Since the onset of the full-scale invasion in February 2022, a third significant source has emerged: fires caused by missile and drone strikes, Figure 1.1. Research covering the first two years of the conflict demonstrates that fire-related emissions became a major contributor to air pollution[3].

Numerous resources study air quality using a variety of approaches — from ground-level sensor networks to satellite technologies. The resulting data is used both in scientific research and in routine public notification systems. While deploying large-scale sensor networks was once technically demanding, advances in embedded systems and network infrastructure have made continuous, high-frequency monitoring across hundreds of stations increasingly accessible — generating data volumes and velocities that conventional processing approaches are ill-suited to handle within the narrow time window in which pollution data retains its actionable value, given that acute spikes in PM2.5 or NO_2 typically persist for hours rather than days.

1.2 Problem Statement

While applications such as Kyiv Digital and EcoCity include built-in notification systems for air quality degradation across various cities, none of the systems examined provides consistent coverage across all regions and at least district-level population centers, based on the publicly available features reviewed at the time of writing. Furthermore, all of these systems operate on the basis of hourly data readings, meaning that notifications may be delayed — an outcome that can affect public health, particularly for individuals with respiratory conditions or other vulnerabilities. Coverage and timeliness are the principal guarantees of effective early warning against negative health impacts.

1.3 Research Objectives

This work aims to address these gaps by designing and implementing a system for real-time air quality monitoring across 414 Ukrainian cities. The system must be capable of continuously ingesting high-throughput sensor readings, detecting pollution events and delivering alerts with sub-minute latency, providing live dashboard visualization of conditions across all monitored stations, and persisting both raw readings and aggregated results for historical analysis.

1.4 Methodology

The problem of real-time air quality monitoring can be decomposed into three sequential concerns: continuous collection of sensor readings from geographically distributed sta-

tions, timely processing of those readings to extract actionable information, and presentation of results to end users and decision-making systems. At small scale these concerns can be addressed by a single application; at the scale of hundreds of stations generating readings every minute or even every few seconds, each stage presents distinct engineering challenges. Data ingestion must accept concurrent streams without data loss, tolerating varying arrival rates. Processing must complete within a time frame that preserves the actionable value of the data — for air quality events, elevated concentrations may persist for only minutes, making sub-minute latency a practical requirement. Visualisation must expose both current conditions and historical context in a form accessible to health authorities and the general public.

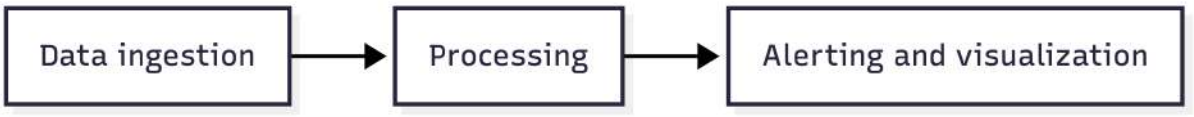

Figure 1.2: General data flow for an environmental monitoring system: sensor readings are collected, processed to extract actionable information, and presented through visualisation and alerting interfaces.

We considered several architectural approaches for implementing this system. A monolithic architecture, in which all concerns are handled within a single application is the natural first candidate. It offers simplicity but cannot scale individual stages independently — a bottleneck at any point degrades the entire system. A microservices architecture decomposes the system into independently deployable services but introduces significant coordination overhead for a data-intensive workload where the primary concern is throughput and latency rather than independent business logic. We chose the data pipeline architecture instead, decomposing the processing path into discrete, concurrently executing stages, each scalable in isolation, while keeping the data flow explicit and the operational model straightforward. Within the pipeline approach, streaming processing was chosen over batch: batch systems accumulate data over a fixed interval before acting on it, introducing an inherent latency floor that is incompatible with sub-minute alert delivery. A streaming pipeline evaluates each reading as it arrives, enabling continuous aggregation and real-time alerting without a fixed delay.

We followed a systematic research approach, beginning with high-level system architecture design covering all components from data-generating applications through visualization. Subsequently, we evaluated and selected implementation technologies, with emphasis on portability, open-source availability, and compatibility with managed cloud equivalents. The implementation phase involved system construction, iterative debugging, and performance testing. Finally, we analyzed the achieved throughput, latency measurements, and implementation challenges encountered during development.

1.5 Scope and Limitations

This research focuses specifically on data ingestion, processing, alerting and visualization infrastructure. The implementation is designed with cloud portability as a primary consideration. Amazon S3 is used for durable archival of raw sensor readings, and Amazon SES provides the email delivery infrastructure for real-time alerts. The remaining pipeline components — message broker, stream processing engine, storage layer, and visualization — are deployed as Docker containers, structured to enable straightforward migration to fully managed cloud equivalents (Amazon MSK, Amazon Managed Service for Apache Flink, and Amazon RDS) as operational scale demands it. This approach prioritizes development reproducibility and local testability while preserving the architectural patterns characteristic of production cloud deployments. The study deliberately excludes application business logic development, detailed examination of traditional relational database implementations, and batch processing methodologies, as these fall outside the core streaming architecture objectives.

1.6 Significance of the Study

This work contributes a practical reference implementation of a real-time monitoring infrastructure suitable for integration into a large-scale public air quality notification system. The architectural patterns and implementation details documented here provide a concrete foundation for building real-time data processing and analytics systems as monitoring infrastructure continues to develop. The work explicitly addresses the limitations of existing implementations — specifically their limited geographic coverage and hourly notification latency — and demonstrates how these gaps can be closed through modern streaming architecture.

Chapter 2

Related Work

2.1 Air Quality Research in Ukraine

Air quality in Ukraine has been the subject of scientific investigation across multiple dimensions over the past decade. Pre-conflict studies focused on establishing baseline pollution levels and identifying key emission sources[4, 5], while more recent work has examined the acute environmental impact of the ongoing military conflict[3]. A persistent challenge across all of these efforts is the reliable collection of high-quality sensor data at sufficient spatial and temporal resolution.

Svidzinska[5] demonstrated that citizen-deployed sensor networks — comprising 133 sensors across 7 independent projects in Kyiv, collecting readings at 1–10 minute intervals — can serve as a meaningful complement or partial substitute for state-operated monitoring infrastructure, which remains sparse and technologically outdated[2]. Shelestov et al.[4] proposed a conceptual infrastructure architecture for measuring PM2.5 and PM10 concentrations over Kyiv using satellite-derived data, ground-based observations, and atmospheric modelling. Zhurakovskyi et al.[6] proposed an IoT-based environmental monitoring system integrating machine learning for automated data collection and prediction of environmental state changes, with TinyML-based local inference to reduce dependence on cloud services.

Despite this body of work, the existing literature on Ukrainian air quality monitoring addresses individual components in isolation — sensor deployment, data collection, or predictive modelling — without integrating them into a pipeline architecture covering the full path from ingestion to alerting and visualization.

2.2 Streaming Pipeline Technologies

Building such an integrated pipeline requires a foundation of mature streaming technologies, and the maturity of that foundation directly shapes what is feasible at the system

level. As documented by Carbone et al.[7], the foundations of stream processing attracted the interest of database research communities as early as the 1990s. That decade witnessed the emergence of prototype systems such as TelegraphCQ and Stanford STREAM, which introduced foundational concepts including window aggregations, fault tolerance, and complex event processing[7]. Building on this research, the first commercial systems — including Esper and Oracle CQL — emerged in the 2000s. The 2010s marked a decisive shift toward widespread industrial adoption, with the introduction of Twitter Storm, Apache Spark Streaming, and Apache Flink, each targeting the growing demand for low-latency, large-scale data processing.

Stream data processing has since matured into one of the most widely adopted data processing paradigms. Modern use cases span IoT data aggregation, real-time fraud detection, and medical monitoring systems — all of which require minimal latency between data generation and actionable output. Kumar proposes a latency-based framework for architectural selection, noting that financial applications demand sub-50 ms response times while behavioural analytics can tolerate latencies measured in minutes[8]. At the technology level, Apache Kafka and Apache Flink have emerged as the dominant combination for production streaming pipelines, with properly configured Kafka clusters sustaining throughput exceeding one million messages per second at sub-10 ms latencies[8].

2.3 Streaming Data Processing Pipelines in Practice

The few publicly documented end-to-end pipeline implementations offer useful reference points for architectural decision-making. Kamal et al.[9] describe the deployment of a real-time visualization and analysis architecture at Cologne University Hospital's Medical Data Integration Center, leveraging the ELK stack for continuous ingestion and illustrating the applicability of streaming architecture in a regulated, data-intensive environment. Akanbi and Masinde[10] propose a distributed stream processing middleware framework for real-time analysis of heterogeneous sensor data, demonstrating its application specifically in the environmental monitoring domain.

Kumar et al.[11] propose a rule-based Complex Event Processing framework for air quality monitoring in smart cities, using Apache Kafka for stream ingestion and SPARQL queries over a knowledge graph for pattern-based alert detection. Sserunjogi et al.[12] documented the AirQo data pipeline — a production cloud-native system for AI-driven air quality monitoring in Africa. The ingestion layer is orchestrated using Apache Airflow while BigQuery serves as the analytical backend. Kafka is employed as a message broker to decouple the ingestion layer from downstream microservices, enabling asynchronous, fault-tolerant data distribution. This work illustrates that real-world deployments often combine streaming and batch processing depending on the operational constraints of the environment.

2.4 Summary

Existing research on air quality in Ukraine has produced valuable contributions across sensor deployment, atmospheric modelling, and conflict-driven pollution analysis[3–5]. However, these efforts address individual components in isolation and do not extend to integrated pipeline architectures capable of continuous ingestion, processing, and real-time notification.

In parallel, the stream processing landscape has matured from academic prototypes into production-grade infrastructure, and end-to-end pipelines built on that infrastructure are now routinely deployed in adjacent sensor-driven domains — from clinical monitoring and general environmental analytics to air quality systems abroad — demonstrating that the technical building blocks for the present work are no longer in question.

Taken together, these threads identify a clear gap: the availability of mature streaming technology and demonstrated domain need in Ukraine has not yet been bridged by a published pipeline architecture oriented toward sub-minute air quality notification. This work addresses that gap.

Chapter 3

Approach

The established approach to problems of data processing is to construct a data pipeline organised around three principal stages: extraction, transformation, and loading (ETL). Increasingly, a complementary pattern — ELT — defers transformation until after data has been loaded into the target system, allowing raw data to be preserved and reprocessed as requirements evolve. Both patterns promote loose coupling between pipeline stages, enabling each component to be scaled independently and preventing bottlenecks in any single subsystem from degrading the throughput of the pipeline as a whole. Stages execute concurrently, so the throughput of the pipeline is determined by the throughput of its individual stages rather than by a single shared execution unit. This is particularly valuable when stage throughputs differ by orders of magnitude — raw sensor readings may arrive at tens of thousands of records per minute, while alert events are generated only once every several minutes — as each stage can be scaled in isolation without affecting the others.

In this chapter we present the pipeline architecture for this work, justify the selection of technologies at each architectural layer, and detail the concrete implementation of each component.

3.1 Pipeline Architecture

The pipeline architecture is shaped by the functional and non-functional requirements of working with air quality sensor data. Readings arrive continuously from a large number of stations; their temporal order must be preserved; the volume of incoming data may be substantial; and processing must be completed as rapidly as possible to minimise the delay between a deterioration in air quality and the notification of affected populations.

The architecture we propose in this work, illustrated in Figure 3.1, organises the pipeline around five stages.

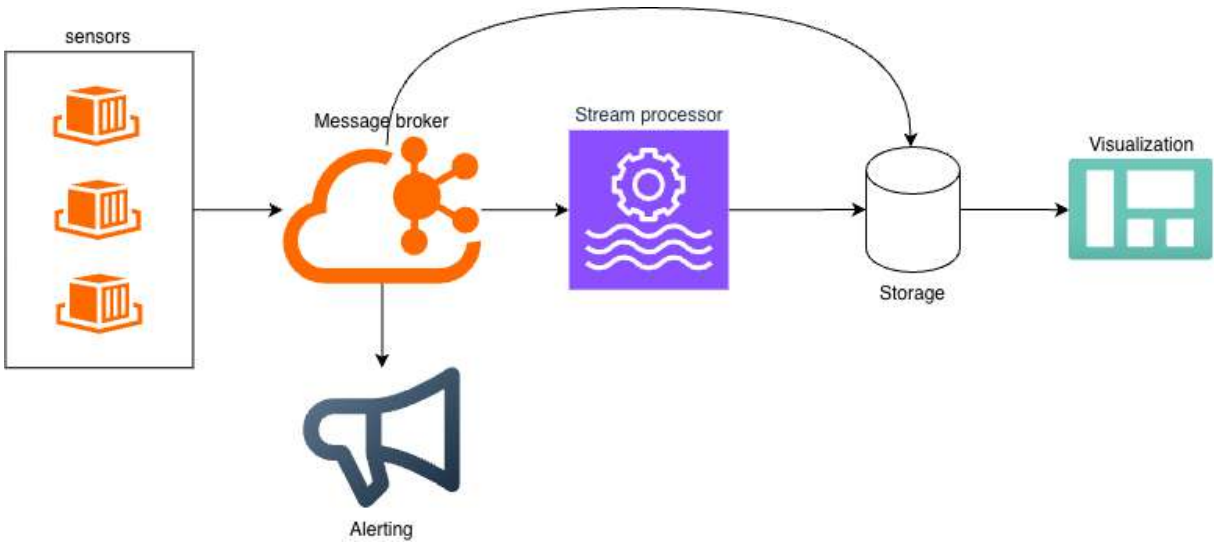

Figure 3.1: Architecture of the streaming data processing pipeline, illustrating the main system components and the direction of data flow.

Data producer publishes readings at a fixed interval to a message broker, which provides durable buffering and decouples the rate of data production from the rate of downstream consumption. The stream processing engine reads from the broker, performs two concurrent tasks: aggregates readings over sliding time windows and persists the resulting aggregates to the storage layer, and evaluates each window against anomaly detection rules, publishing alert events back to the broker for delivery to the alerting service. The storage layer serves as the data source for a real-time dashboard, enabling visualisation of current conditions, historical trends, and overall pipeline health.

3.2 Technology Selection

The selection of technologies for a data pipeline is non-trivial, as numerous tools exist for each architectural role with substantially overlapping feature sets. We evaluated the candidates at each layer against the following criteria: low end-to-end latency, support for local deployment and reproducible development environments, open-source availability with active community support, quality of documentation, and compatibility with managed cloud service equivalents to facilitate possible migration. Each subsection below presents the candidate technologies considered for that layer, evaluates them against these criteria, and states the selection made.

3.2.1 Message Queue

Since each sensor reading is a discrete event, the natural entry point for raw data is a message broker — a system designed specifically for durable storage and distribution of messages between producers and consumers. The message broker is among the most

performance-critical components of the pipeline: every sensor reading must pass through it, meaning it sustains the highest data volume of any single component. Three candidates were evaluated: RabbitMQ, Apache Pulsar, and Apache Kafka.

RabbitMQ is a general-purpose message broker built around the AMQP protocol. However, it operates as a task queue: messages are deleted once acknowledged, making replay after a crash or consumer restart impossible. This pipeline requires durable, replayable log storage, so RabbitMQ was excluded from consideration.

Apache Pulsar is a distributed messaging and streaming platform that supports both queuing and log-based consumption models. It offers multi-tenancy, geo-replication, and a tiered storage architecture. However, this feature set introduces significant operational complexity — Pulsar requires coordinating multiple separate components (broker, BookKeeper, ZooKeeper) even for a minimal deployment. Given that simplicity of local deployment is an explicit criterion for this work, and that the additional capabilities of Pulsar over Kafka are not required by this pipeline, Pulsar was excluded from further consideration.

Apache Kafka is a distributed commit log designed for high-throughput, durable, ordered event streaming. Kafka organises data into named topics, each partitioned across brokers to support parallel consumption. Messages are retained for a configurable period regardless of consumer state, allowing any consumer to replay or catch up on missed messages after a restart — directly addressing the durability requirements that disqualified RabbitMQ. Kafka's topic model also allows the pipeline to cleanly separate raw input, processed output, and alert events into independent streams, with each consumer operating at its own pace without affecting others. Kafka's partition-based model further allows the throughput of any topic to be scaled horizontally by increasing the number of partitions. It is open-source, well-documented, and has a managed cloud equivalent in Amazon MSK. For these reasons, Apache Kafka was selected as the message broker for this pipeline. The resulting topic structure, separating raw input from processed and alert streams, is illustrated in Figure 3.2.

Table 3.1: Comparison of message broker candidates

Property	**RabbitMQ**	**Apache Pulsar**	**Apache Kafka**
Consumption model	Push-based queue	Log + queue	Distributed log
Message retention	Until acknowledged	Configurable	Configurable
Multi-consumer replay	No	Yes	Yes
Local deployment	Simple	Complex	Simple
Cloud equivalent	Amazon MQ	—	Amazon MSK

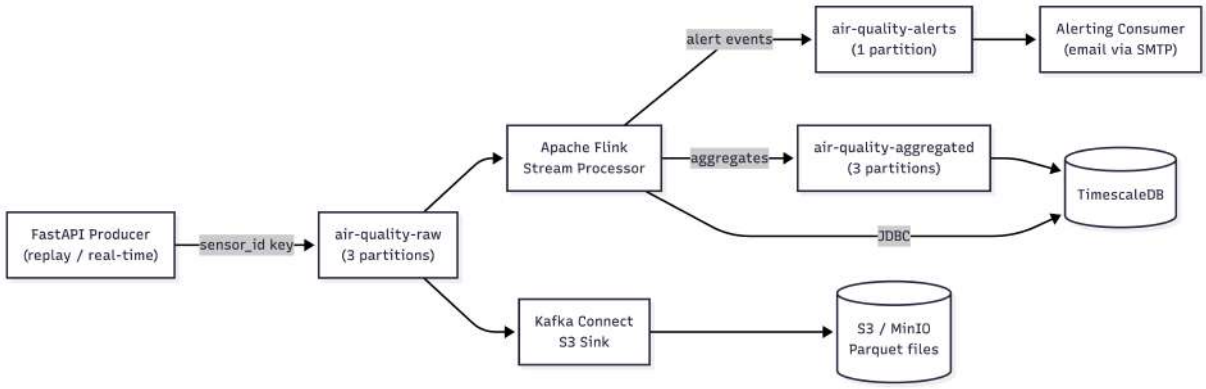

Figure 3.2: Kafka topic structure: data flow from producer through the three topics to Flink, S3 archive, and alerting consumer.

3.2.2 Stream Processing Engine

For processing the data we initially designed a microservice that polled batches of events from Kafka, but after it started failing under high load, stream processing was adopted as the core computation model. Unlike batch processing, which accumulates data over a fixed interval before operating on it, stream processing evaluates each record as it arrives, enabling continuous aggregation, correct ordered windowing, and near-real-time anomaly detection. Two candidate frameworks were evaluated: Apache Spark Streaming and Apache Flink.

Apache Spark Streaming operates on a micro-batch execution model: rather than processing records individually as they arrive, it accumulates incoming data into small batches and processes each batch as a discrete unit. This introduces an inherent minimum latency equal to the batch interval, which is incompatible with the sub-minute end-to-end latency requirement of this work.

Apache Flink is a distributed stream processing engine built on a true event-at-a-time execution model. It provides native support for event-time semantics, watermarking, stateful operators, and windowed aggregations — all of which are required by this pipeline for correct processing of out-of-order sensor readings. Empirical benchmarks by Joy [13] further confirm that Flink achieves lower latency and higher throughput than Spark Streaming. Flink is open-source, well-documented, and has a managed cloud equivalent in Amazon Managed Service for Apache Flink. For these reasons, Apache Flink was selected as the stream processing engine for this pipeline.

3.2.3 Data Storage

The storage layer of this pipeline must fulfil two distinct responsibilities. The first is raw data archival: every sensor reading published to Kafka must be durably persisted in its original form to support historical reconciliation, incident investigation, and fu-

ture reprocessing without re-ingestion. The second is serving aggregated results to the visualisation layer with low latency. These two responsibilities impose conflicting access patterns — append-only high-volume sequential writes for the archive versus concurrent indexed reads for the dashboard — and no single storage technology addresses both optimally. The pipeline therefore employs two separate storage tiers: a raw archive backed by Amazon S3, and a time-series database for processed aggregates.

Raw Archive: Amazon S3 with Kafka Connect. The raw archive must accept an unbounded stream of sensor readings with minimal operational overhead, retain them indefinitely at low cost, and make them available for batch reprocessing. Object storage is the natural fit for this role: it decouples write throughput from storage capacity, scales to arbitrary data volumes without re-partitioning, and stores data in open formats readable by any analytical tool.

Amazon S3 was selected as the object store. It offers eleven nines of durability, native integration with the broader AWS data ecosystem, and a cost model suited to write-heavy append workloads. We chose to write to S3 using the Kafka Connect S3 Sink connector, which reads directly from the Kafka topic and flushes batches of records as JSON files partitioned by ingestion date and hour. JSON was chosen for its zero-configuration compatibility with the schemaless records emitted by the producer; richer columnar formats such as Parquet require a Connect schema in every record, which would have demanded an additional schema-registry component for a marginal storage gain on this workload. This connector-based approach keeps the archival path entirely outside the Flink job: the stream processor is not burdened with raw persistence, and the two concerns remain independently deployable and scalable.

Processed Aggregates: TimescaleDB. The visualisation layer requires low-latency random reads over aggregated time-series data produced by the stream processor. The first architectural choice for storing aggregated data was ClickHouse, which is an OLAP database that enables continuous aggregation and has built-in integration with Kafka as a datasource. But after discovering its russian origins we decided to exclude it from consideration. Two candidates were evaluated as a substitute: DuckDB and TimescaleDB.

DuckDB is an analytical database designed for single-node deployments. It employs a columnar storage format with vectorised query execution, delivering competitive analytical performance for datasets that fit within the capacity of a single machine [14]. However, DuckDB operates as an embedded library rather than a standalone server — it does not expose a network endpoint, meaning that multiple independent processes cannot connect to it concurrently. Since the stream processing engine and the visualisation layer are separate processes that must access the storage layer simultaneously, this architectural constraint disqualifies DuckDB for this use case.

TimescaleDB is an open-source time-series database implemented as an extension to PostgreSQL. It introduces hypertables — automatically partitioned tables optimised for time-ordered inserts and range queries — while retaining full SQL compatibility and the complete PostgreSQL tooling ecosystem. This compatibility is particularly valuable for this pipeline: the stream processing engine can write to TimescaleDB using a standard JDBC connector, and the visualisation layer can query it through a native PostgreSQL datasource without any additional plugins or drivers. TimescaleDB can be deployed identically in both local Docker-based environments and cloud infrastructure, satisfying the local reproducibility criterion. For these reasons, TimescaleDB was selected as the processed-aggregates store for this pipeline. The two-tier storage architecture is illustrated in Figure 3.3.

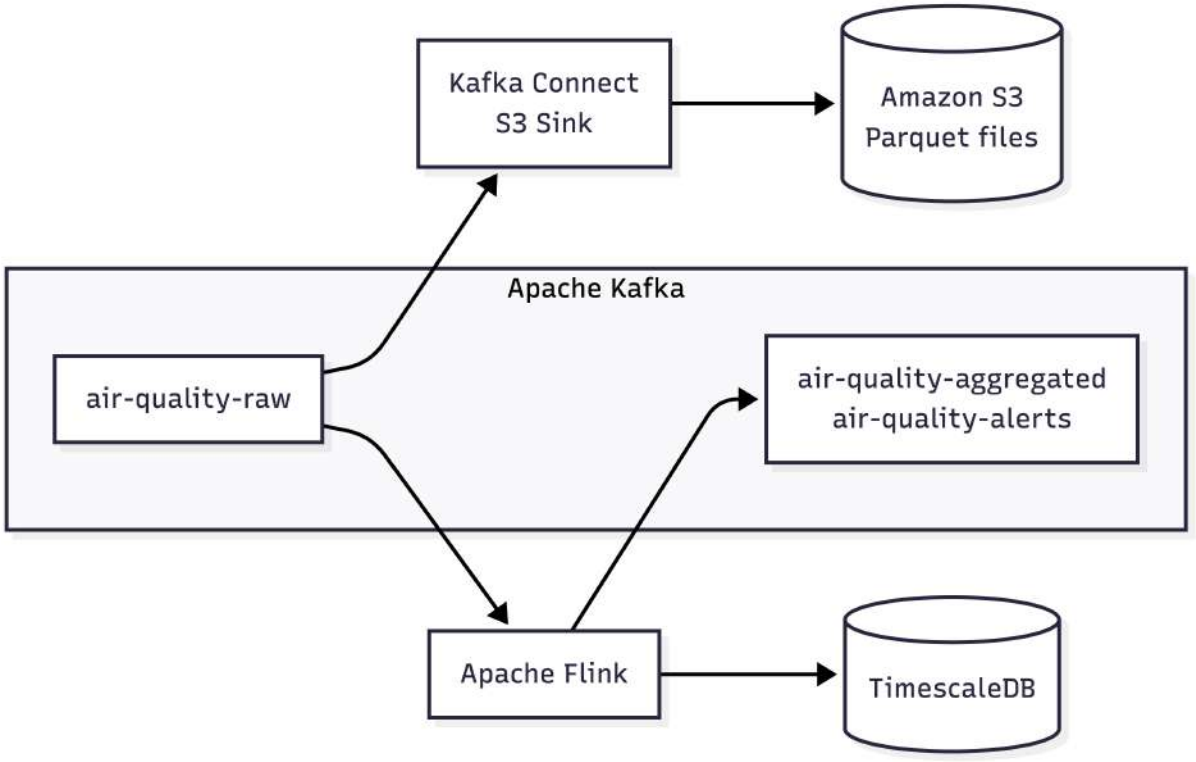

Figure 3.3: Two-tier storage architecture: raw sensor readings archived to Amazon S3 via Kafka Connect; processed aggregates persisted to TimescaleDB by the Flink job.

3.2.4 Visualization

Visualisation is an important but not the most performance-critical component of the pipeline. The requirements here are less strict than for the message broker or stream processor: the tool must be straightforward to configure, produce clear and interpretable dashboards, and connect natively to TimescaleDB without requiring custom plugins. Two candidates were evaluated: Apache Superset and Grafana.

Apache Superset is an open-source business intelligence platform that supports a broad

range of chart types and SQL-based data exploration. However, it is designed primarily for analytical reporting over historical datasets rather than real-time monitoring. Its dashboard refresh model is pull-based and not oriented towards sub-minute update intervals. Superset also requires a non-trivial deployment stack — a metadata database, a caching layer, and a Celery worker — which adds operational overhead disproportionate to the visualisation requirements of this pipeline.

Grafana is an open-source monitoring and observability platform designed specifically for time-series data and real-time metrics. It ships with a native PostgreSQL datasource that connects to TimescaleDB without additional plugins, supports sub-minute dashboard refresh intervals, and is widely used in conjunction with Kafka-based data pipelines. Critically for this project, Grafana supports fully declarative provisioning: datasources and dashboards are defined as YAML and JSON configuration files that are loaded automatically on startup, enabling the entire visualisation layer to be version-controlled and reproduced identically across environments. Grafana is open-source and has a managed cloud equivalent in Grafana Cloud, preserving the option of future migration. For these reasons, Grafana was selected as the visualisation layer for this pipeline.

Table 3.2: Comparison of visualisation candidates

Property	**Apache Superset**	**Grafana**
Local deployment	Yes, complex	Yes, simple
Real-time refresh	Limited	Native
Native TimescaleDB support	Via PostgreSQL	Via PostgreSQL
Declarative provisioning	No	Yes
Open-source	Yes	Yes
Cloud equivalent	—	Grafana Cloud

3.2.5 Data Producer

Since this work does not involve deploying physical sensors, we developed a lightweight simulation service to generate sensor readings and publish them to Kafka. We implemented two modes for it: a replay mode that replays historical readings at a configurable speed to reproduce realistic load conditions during evaluation, and a real-time mode that fetches live air quality measurements from an external API.

FastAPI was selected as the framework for this service. FastAPI is an asynchronous Python web framework built on top of Starlette and Pydantic. Its native support for Python's `asyncio` is directly useful here: both operation modes are implemented as async generators that yield `SensorReading` objects, and the FastAPI lifespan mechanism starts the selected simulation mode as a background task on service startup. FastAPI also exposes health and statistics endpoints at no additional cost, which are used to monitor producer throughput during pipeline evaluation. The framework is lightweight, well-

documented, and imposes no deployment overhead beyond a single container.

3.2.6 Alerting

Grafana was initially considered for alerting, as it is already present in the stack and requires no additional service. However, Grafana alerting is limited to stateless threshold comparisons: it cannot track whether elevated conditions persist across multiple consecutive windows or distinguish between the onset of a new event and a continuing one. The alert logic in this pipeline requires this kind of multi-window stateful reasoning — generating an onset notification when degradation first appears and a recovery notification when it clears — to suppress repeated alerts for the same ongoing event.

The design adopted here separates alert generation from delivery. Flink publishes structured alert events to a dedicated Kafka topic; a lightweight Python consumer reads from that topic and delivers email notifications via SMTP. This keeps the stream processor free of delivery concerns and allows the delivery channel to be changed independently. Example alert emails produced by this design are shown in Figure 3.4.

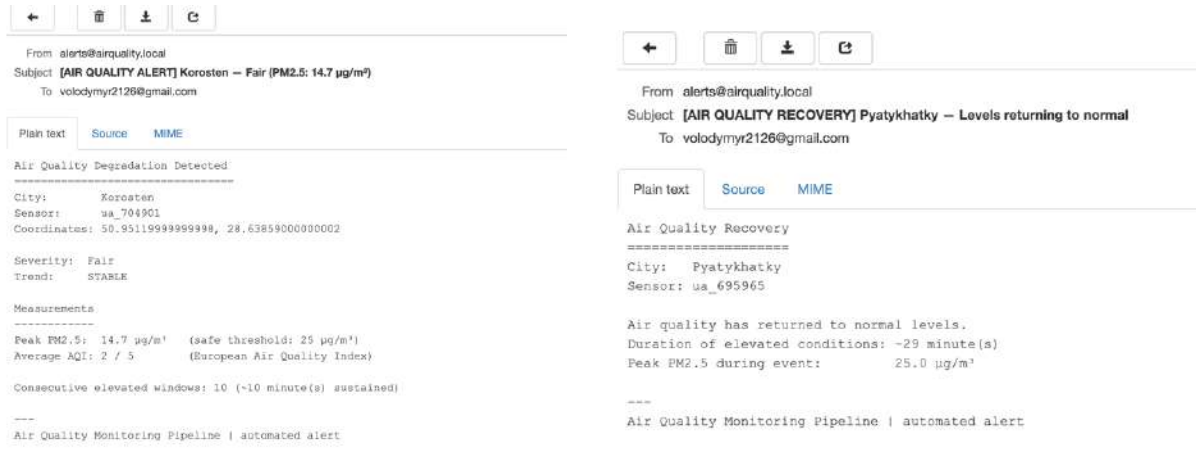

Figure 3.4: Example alert emails delivered by the alerting consumer: onset notification (left) and recovery notification (right).

3.3 Implementation

This section describes the implementation of the proposed pipeline. Each subsection covers a distinct component of the architecture — data generation, stream processing logic, storage schema, dashboard configuration, and deployment — and explains the concrete design decisions made to satisfy the requirements identified in Section 3.1.

3.3.1 Data Generation

First, a data source for the project had to be selected. We tested several public APIs: aqicn, OpenAQ, and OpenWeatherMap. OpenWeatherMap was chosen as it offered the largest number of monitoring locations, a convenient data format, and the most straightforward API access.

We first developed a real-time mode that fetches data live while respecting the OWM API rate limit of no more than 60 requests per minute. Upon realising that this rate limit prevented high-load testing, we developed a replay mode to demonstrate system stability under sustained load. Using it, we conducted a load test of the pipeline, ingesting over 24,000 records per minute. The historical dataset used for replay was collected in advance via a separate loader script that polls the OWM Air Pollution API for all monitored stations. As historical readings arrive once per hour, we augmented the dataset using a normal distribution to interpolate per-minute readings, and stored the results as Parquet files in S3, partitioned by sensor identifier and date.

Files for one day are loaded in parallel and sorted by timestamp, after which the producer emits records at a pace scaled by a configurable speed multiplier: the original inter-event intervals are preserved but compressed in time, so the pipeline receives a realistic distribution of sensor activity rather than an artificial uniform stream. The dataset is processed one day at a time to keep memory consumption flat regardless of how much historical data is available, and the replay loops indefinitely to sustain load throughout an evaluation run.

Both modes emit records using the same `SensorReading` schema: a unique reading identifier, sensor identifier, station name, city and geographic coordinates, a UTC timestamp, and other air quality related fields. This shared schema means the rest of the pipeline — the Kafka topic, the Flink job, and the storage layer — is entirely agnostic to which mode the producer is running in. The two modes are compared in Table 3.3.

Table 3.3: Comparison of data producer operation modes

Property	**Replay Mode**	**Real-time Mode**
Data source	Historical Parquet files (S3)	OpenWeatherMap Air Pollution API
Rate control	Configurable speed multiplier	OWM rate limit (60 req/min)
Primary purpose	Load testing and evaluation	Live air quality monitoring
Coverage	414 cities (historical set)	414 cities (live stations)
Output schema	`SensorReading`	`SensorReading`

3.3.2 Stream Processing Logic

The stream processor is implemented as a single PyFlink job that subscribes to the `air-quality-raw` Kafka topic as its sole input and fans out to four sinks: two Kafka topics for downstream consumers (`air-quality-aggregated` and `air-quality-alerts`) and

two TimescaleDB tables for persistence (`sensor_aggregates` and `air_quality_alerts`). All sources and sinks are declared as Flink Table API tables with JSON or JDBC connectors, and a single `StatementSet` submits the four inserts together so that they share the same execution graph and checkpoint barriers.

The source table parses each Kafka record's JSON payload into the `SensorReading` schema, derives an `event_time` column from the record's UTC timestamp, and declares a watermark of `event_time - INTERVAL '5' SECOND`. A 5-second idle-source timeout is set so that watermarks continue to advance when a partition is briefly silent, which would otherwise stall the windowed aggregation. The overall data flow through the job is shown in Figure 3.5.

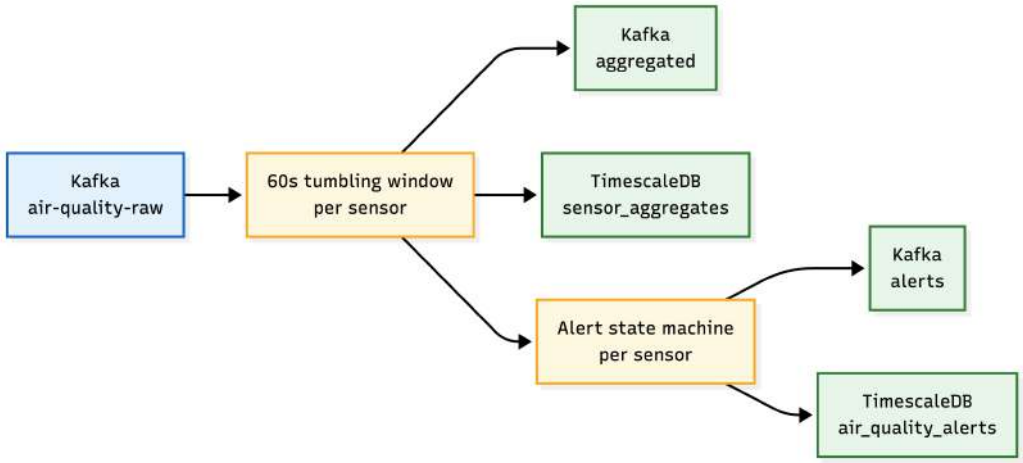

Figure 3.5: Stream processing job: a single Kafka source feeds a 60-second tumbling window aggregation that fans out to a Kafka topic and TimescaleDB; the aggregated stream is then keyed by sensor and processed by an alert state machine that emits onset/recovery events to a second pair of sinks.

On top of this source the job performs two concurrent tasks. The first is a windowed aggregation expressed in SQL through the Table API: readings are grouped by sensor identifier into 60-second tumbling windows over `event_time`, producing per-window averages and maxima for PM2.5, PM10, NO_2, O_3, SO_2, and AQI together with a reading count. The resulting table is inserted into both the aggregated Kafka sink and the TimescaleDB JDBC sink in parallel.

The second task is alert detection, which operates on the aggregated stream and maintains per-sensor state across consecutive windows. Because Flink SQL does not expose the kind of cross-window stateful reasoning required here, the aggregated table is converted to a `DataStream`, keyed by sensor identifier, and processed by a `KeyedProcessFunction` that holds two pieces of managed state per key: a `ValueState` carrying the current alert status and counters for consecutive elevated, pending, and recovery windows, and a `ListState` acting as a rolling 72-window history buffer used for trend and peak computation. Anomaly detection uses a Z-score approach: each window's PM2.5 and AQI

values are compared against a rolling baseline of the previous 72 windows, and a window is flagged as elevated if either metric exceeds two standard deviations above the mean. Per-sensor baseline statistics are pre-computed from historical data and loaded from a JSON file at job startup, avoiding a cold-start period during which no baselines would be available. When no baseline exists for a sensor, fixed fallback thresholds are applied instead; these are not defined by us but taken from the European Air Quality Index band boundaries published by the European Environment Agency[15]. To suppress alert flapping, an onset event is emitted only after 10 consecutive elevated windows, and a recovery event only after 10 consecutive normal ones. Emitted alerts are converted back into a Table API row, then written to the alerts Kafka topic and to the `air_quality_alerts` table in TimescaleDB through the same `StatementSet`. The resulting state machine is illustrated in Figure 3.6.

The Flink job is configured with exactly-once processing semantics and a 10-second checkpoint interval. Checkpoints are written to a persistent volume, allowing the job to resume from the last consistent state after a restart without reprocessing or losing in-flight window aggregates. Parallelism is set to one for the experiments reported in Chapter 4; the design does not depend on this choice, as Kafka partitioning of the source topic and the per-sensor key in the alert function would allow parallelism to be increased without changes to the logic.

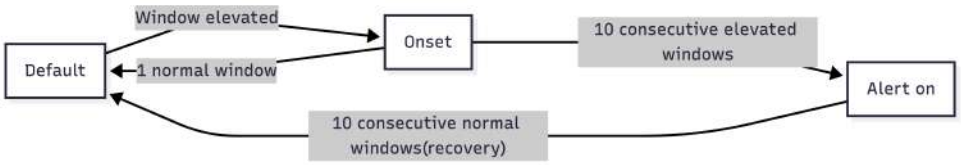

Figure 3.6: Alert state machine: onset notification emitted after 10 consecutive elevated windows; recovery notification emitted after 10 consecutive normal windows.

3.3.3 Storage Schema

The stream processor writes its output to two destinations in parallel. Aggregated window results and alert events are both published to dedicated Kafka topics — `air-quality-aggregated` and `air-quality-alerts` respectively — making them available for any downstream consumer without coupling to the storage layer. The same data is simultaneously persisted to TimescaleDB for long-term storage and dashboard queries. We proposed two separate tables for this purpose: `sensor_aggregates` and `air_quality_alerts`.

`sensor_aggregates` is a hypertable partitioned on `window_start`. It stores one row per sensor per 60-second window, containing the window boundaries, sensor identity and location, aggregate pollutant metrics, and a reading count. An `inserted_at` column records when each row was written to the database; the difference between `inserted_at` and `window_end` serves as the end-to-end latency measure for the pipeline. Composite

indexes on `(sensor_id, window_start)` and `(city, window_start)` support the time-range queries issued by the Grafana dashboard.

`air_quality_alerts` is also a hypertable, partitioned on `created_at`. It stores onset and recovery events with sensor identity, severity level, peak PM2.5 concentration, trend direction, and event duration. Indexes on sensor, city, and event type allow the dashboard to filter and aggregate alert history efficiently.

3.3.4 Visualization Dashboards

A single Grafana dashboard was developed to serve two purposes simultaneously: real-time visibility into air quality conditions across monitored stations, and operational monitoring of the pipeline itself. The dashboard is defined entirely as a provisioned JSON configuration file, loaded automatically on startup without any manual setup.

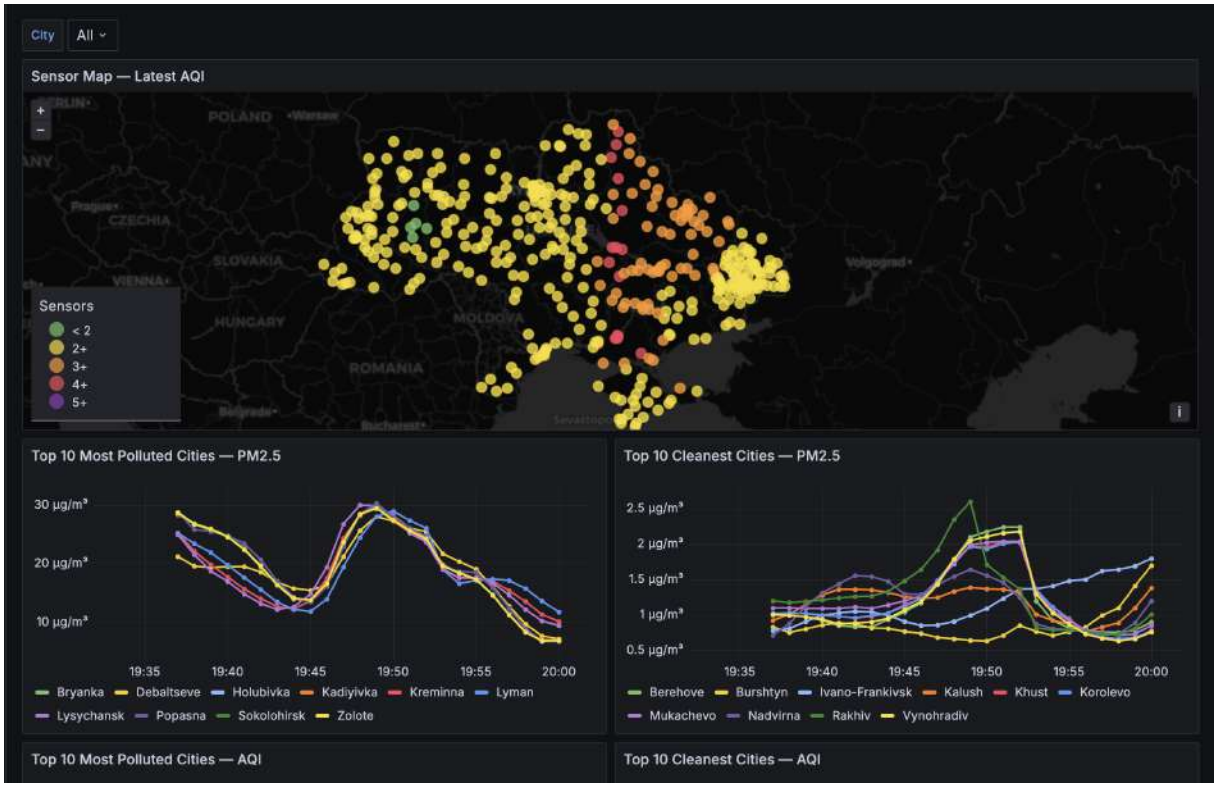

Figure 3.7: Grafana dashboard showing the Ukraine sensor map and top polluted and cleanest cities by PM2.5.

The top of the dashboard displays a map of Ukraine with all active monitoring stations marked by location. Below it, four panels show the cleanest and most polluted cities ranked by PM2.5 and AQI respectively, giving an immediate overview of current conditions across the country. Further down, two panels track the alert state — active onset and recovery events by city and sensor. The bottom row contains two system health panels: one showing the number of readings processed per window, and one showing

the average end-to-end latency computed as `inserted_at` − `window_end` for each row in `sensor_aggregates`. This latency panel is the primary instrument for verifying the sub-minute processing guarantee that is the central claim of this work. The dashboard refreshes every 30 seconds. The main dashboard view is shown in Figure 3.7; the pipeline health panels are shown in Figure 3.8.

Figure 3.8: Pipeline health panels: throughput (events per 60-second window) and end-to-end latency (window close to TimescaleDB insert).

3.3.5 Deployment

The entire pipeline is orchestrated locally using Docker Compose, which defines all components as services on a shared bridge network. Startup ordering is enforced through health checks and dependency conditions: Kafka must pass a broker API version check before the producer or Flink job are allowed to start, TimescaleDB must pass a `pg_isready` check before Flink connects, and the Flink JobManager must be reachable before the TaskManager registers and the job is submitted.

The stack comprises the following services: a KRaft-mode Kafka broker with a topic initialisation container that creates the three topics on first run; the FastAPI producer connected to S3 for historical Parquet data; a Flink JobManager and TaskManager pair with a job submitter container that deploys the stream processor; TimescaleDB with schema initialised from a SQL file mounted at startup; the alerting consumer connected to MailHog for local email capture and SES for production; and Grafana with provisioned datasource and dashboard configuration. Kafka UI is also included for inspecting topic contents during development.

Sensitive configuration — AWS credentials, SMTP settings, replay speed, and producer mode — is passed through environment variables, allowing the same Compose file to be used for both local evaluation and cloud-connected runs without code changes.

3.4 Conclusions

This chapter described the design and implementation of a streaming data pipeline for real-time air quality monitoring. The pipeline is organised around five loosely coupled stages — data generation, message brokering, stream processing, storage, and visualisation — each implemented with a technology selected to satisfy the two principal requirements of the work: sub-minute end-to-end latency and full local reproducibility.

Apache Kafka was selected as the message broker for its durable, replayable log model; Apache Flink as the stream processor for its native event-time semantics and stateful windowing; TimescaleDB as the time-series store for its PostgreSQL compatibility and hypertable partitioning; and Grafana as the visualisation layer for its native TimescaleDB support and declarative provisioning. Raw sensor data is archived to Amazon S3 via the Kafka Connect S3 Sink connector, providing an immutable data lake that also serves as the source for a historical replay. The entire stack is deployable with a single Docker Compose command, with startup ordering enforced through health checks.

The implemented pipeline demonstrated the ability to sustain over 24,000 records per minute under load while maintaining end-to-end latency — measured as the delay between window close and database insertion — within the sub-minute threshold.

Chapter 4

Evaluation

This chapter evaluates the implemented pipeline against the two primary claims stated in the research objectives: the ability to sustain high-throughput ingestion across all monitored cities, and the delivery of end-to-end latency below one minute from sensor reading generation to stored aggregate.

4.1 Experimental Setup

All experiments were conducted on a single host machine (Apple M-series, 16 GB RAM) running all pipeline services via Docker Compose. The only external dependency was Amazon S3, used as durable storage for the historical Parquet archive that the producer replays from; all processing, storage, and visualisation components ran locally. The data source was a historical Parquet file replayed at configurable speed by the FastAPI producer, simulating continuous readings from 414 Ukrainian cities. The speed multiplier was increased stepwise across nine test runs, with each run sustained until Kafka consumer lag and Flink checkpoint duration stabilised, confirming the pipeline was keeping pace with incoming data.

End-to-end latency was instrumented via the `inserted_at` timestamp column written to the `sensor_aggregates` hypertable in TimescaleDB at the moment each aggregate record is persisted. Latency is defined as `inserted_at` − `window_end`: the elapsed time between the close of a 60-second tumbling window and the moment the resulting aggregate becomes queryable. This metric captures processing time through Flink, serialisation, and database write, and directly reflects the delay between a pollution event occurring and its data being available for alerting and visualisation. The producer overwrites historical timestamps with the current wall-clock time on emit, so the measurement reflects real elapsed time rather than compressed replay time.

4.2 Throughput

The producer replay rate was increased stepwise to determine the maximum sustainable ingestion rate. Table 4.1 presents the complete set of tested configurations. The pipeline ingested data without observable backpressure or checkpoint degradation from 1× through 1,000×, covering a range from 414 to 414,000 records per minute. At 10,000× the configured rate of 4.14 million records per minute could not be sustained; the producer itself became the bottleneck, as the rate at which it can read and re-emit records from the S3 Parquet archive is bounded by object storage throughput and local I/O. Effective throughput plateaued at approximately 1.79 million records per minute, as visible in Figure 4.1.

Table 4.1: Pipeline throughput and end-to-end latency across tested replay speeds

Speed multiplier	Records / min	p95 latency (s)	p99 latency (s)
1×	414	37.10	37.60
5×	2,070	8.04	8.69
20×	8,280	7.39	7.86
60×	24,760	5.34	5.96
100×	41,400	5.90	6.35
200×	82,800	5.42	5.68
500×	207,000	5.27	5.88
1,000×	414,000	5.43	5.66
10,000×	1,790,000	13.00	13.65

Figure 4.1: Grafana pipeline health panel at 10,000× replay speed: throughput plateaus at approximately 1.79 million records per minute, indicating that the data producer — not the Kafka or Flink layers — is the binding bottleneck at this configuration.

4.3 End-to-End Latency

The latency results in Table 4.1 reveal a clear pattern across the tested range. At 1× replay speed (414 records per minute), p95 latency is 37.1 seconds — substantially higher than at any other tested rate. This is explained by the watermark mechanism in the Flink job. The pipeline uses a 5-second event-time watermark to tolerate late arrivals. At 1× speed, each of the 414 sensors emits approximately one reading per minute; the resulting stream is sparse enough that individual partitions frequently go quiet between

events, causing the watermark to stall until the next reading arrives from a lagging sensor. A stalled watermark delays window closure, which in turn delays the aggregate write to TimescaleDB. This effect disappears as the replay rate increases and the stream becomes dense enough to advance the watermark continuously.

From 5× onwards, latency stabilises at approximately 5 to 9 seconds and remains within that band through the entire range up to and including 1,000×. This floor corresponds closely to the configured 5-second watermark delay plus the time required for Flink to perform the aggregation, serialise the result, and write it to TimescaleDB. All values in this range are well within the sub-minute threshold stated in Section 1.3.

At 10,000×, even with the producer-bound effective rate of 1.79 million records per minute, p95 latency rose only to 13 seconds and p99 to 13.65 seconds — indicating the onset of resource contention on the shared host, but still well within the one-minute requirement.

4.4 Alert Detection

Alert correctness was verified by replaying a sequence of readings with PM2.5 concentrations set to two standard deviations above the sensor baseline, sustained for more than ten consecutive 60-second windows. The Flink anomaly detection operator correctly identified the onset condition and dispatched an alert email after the tenth consecutive elevated window. Upon return to baseline readings, a recovery notification was issued after the corresponding ten-window recovery period. The individual onset and recovery email templates are shown in Figure 3.4; Figure 4.2 shows the MailHog inbox after a full evaluation run, confirming that multiple sensors triggered and resolved alerts independently without interference.

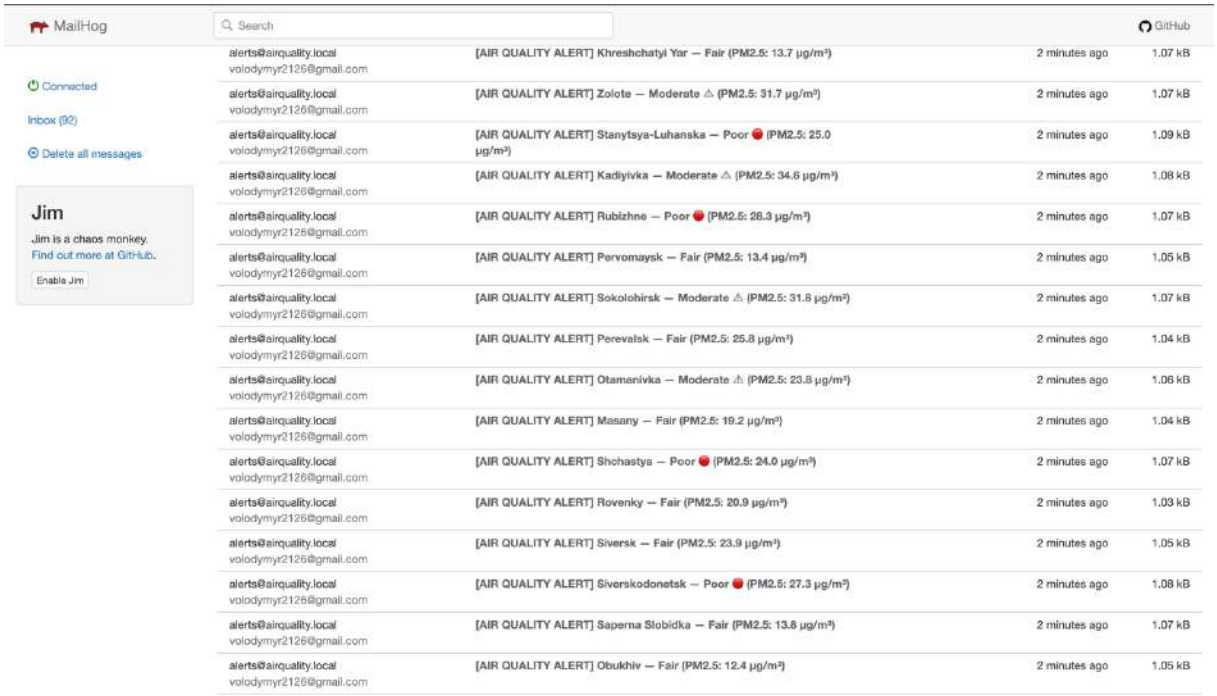

Figure 4.2: MailHog inbox after a full evaluation run, showing onset notifications dispatched independently for multiple sensors.

4.5 Discussion

The results demonstrate that the implemented pipeline meets both primary objectives under the experimental conditions described. End-to-end latency remained below one minute across all nine tested configurations, including the 10,000× run where resource contention raised p95 to 13 seconds; and alert detection operated correctly, suppressing spurious notifications through the ten-window onset and recovery requirements.

Several limitations of the evaluation should be noted. First, the data source is a replay of historical sensor readings rather than live hardware sensors, meaning network jitter and sensor failure scenarios are not represented. Second, all services were co-located on a single host, which eliminates network latency between components present in a distributed cloud deployment; the elevated latency at 10,000× reflects single-host saturation rather than a fundamental limit of the architecture. At production scale — with ingestion distributed across Amazon MSK, processing on Amazon Managed Service for Apache Flink, and storage on Amazon RDS — latency characteristics would differ, though the architectural design preserves the same processing guarantees. Fault tolerance under node failure and cold-start recovery time were not measured and remain directions for future evaluation.

Chapter 5

Future Work

Several directions remain open for future development of the pipeline.

Cloud deployment. The local Docker Compose stack is architecturally equivalent to a fully managed cloud deployment, but the migration itself has not been performed. Deploying the pipeline to Amazon MSK, Amazon Managed Service for Apache Flink, and Amazon RDS would validate that the architecture sustains the same latency and throughput guarantees under real network conditions and distributed resource allocation.

Live sensor integration. All evaluation in this work used replayed historical data. Integrating live sensor hardware — or a larger set of real-time API sources — would introduce network jitter, sensor dropout, and calibration drift that the replay mode cannot reproduce. Evaluating pipeline behaviour under these conditions, including the effect of the watermark on genuinely late-arriving readings, remains an important step toward production readiness.

Fault tolerance. The Flink job is configured with exactly-once semantics and periodic checkpointing, but recovery from node failure was not evaluated. Measuring cold-start recovery time — the delay between a TaskManager crash and the resumption of window processing from the last checkpoint — would quantify the practical availability of the pipeline under failure.

Anomaly detection quality. The current Z-score detector uses a static per-sensor baseline loaded at job startup. A rolling baseline updated from the live stream would adapt to gradual shifts in sensor output caused by seasonal variation or sensor degradation. More sophisticated approaches — including machine learning models trained on historical patterns — could reduce false positives further, particularly for stations near industrial sites with predictable periodic emission profiles.

Broader geographic coverage. The pipeline was designed and tested for 414 Ukrainian monitoring stations. Extending coverage to additional countries or integrating satellite-derived air quality estimates alongside ground-level sensors would increase the practical value of the system as a public health notification infrastructure.

Targeted alert routing. The current alerting consumer delivers all notifications

to a single endpoint regardless of the geographic origin of the event. A more practical design would route alerts based on the location of the affected station — notifying regional health authorities, municipal services, or individual subscribers with a direct interest in a given area. This could be implemented through a publish-subscribe model in which consumers subscribe to alerts filtered by region or city, eliminating irrelevant notifications and reducing response latency for affected parties.

Chapter 6

Conclusions

6.1 Summary

This work set out to design and implement a real-time air quality monitoring pipeline capable of continuously ingesting sensor readings from 414 Ukrainian cities, detecting pollution events and delivering alerts with sub-minute latency, providing live dashboard visualisation, and persisting both raw readings and aggregated results for historical analysis. All four objectives stated in Section 1.3 were achieved.

The pipeline is organised around five loosely coupled stages. A FastAPI-based data producer publishes sensor readings to Apache Kafka, which provides durable, replayable buffering between production and consumption. An Apache Flink job reads from Kafka and performs two concurrent tasks: windowed aggregation over 60-second tumbling windows and Z-score-based anomaly detection with a ten-window onset and recovery state machine. Aggregated results are persisted to TimescaleDB, which serves as the data source for a Grafana dashboard providing real-time visualisation of conditions across all monitored stations. Raw readings are archived to Amazon S3 via the Kafka Connect S3 Sink connector, forming an immutable data lake for future reprocessing. A lightweight alerting consumer delivers onset and recovery notifications by email when the stream processor detects sustained anomalies.

The entire stack is reproducible with a single Docker Compose command, with startup ordering enforced through health checks and sensitive configuration passed through environment variables.

6.2 Evaluation Results

The pipeline was evaluated through a series of load tests using a historical dataset replayed at configurable speed. End-to-end latency was measured as the elapsed time between window close and database insertion — the delay between a pollution event occurring

and its data becoming queryable.

At near-real, production replay rates (from 5× to 1,000×, which are from 2,070 to 414,000 records per minute), p95 latency was approximately 5 to 9 seconds, which is within the sub-minute threshold. The latency floor corresponds to the configured 5-second watermark delay plus Flink processing and database write time. At 1× speed, sparse inter-event intervals caused watermark stalls that raised p95 to 37 seconds; this is a direct consequence of the single-reading-per-minute regime rather than a limitation of the pipeline at realistic load. At 10,000× the data producer became the bottleneck, reaching effective throughput at approximately 1.79 million records per minute and stopping there; even at this rate p95 latency remained at 13 seconds, well which still satisfies the one-minute requirement.

Alert detection operated correctly across all tests: onset notifications were dispatched after ten consecutive elevated windows and recovery notifications after ten consecutive normal ones, confirming that the state machine suppresses spurious alerts for sustained events without introducing false negatives.

Acknowledgement of LLM Use

Large language models were used as a writing and engineering aid during the preparation of this thesis. Anthropic's Claude Sonnet 4.6 was used to: translate Ukrainian text into English and improve the clarity of prose in this thesis; review code during pipeline components development; help with locating secondary literature, all references to which were verified against their primary sources. All content that was generated was reviewed by the author, and all decisions, evaluation, and wording still are the responsibility of the author.

Bibliography

[1] M. Wall. *Russian attack sets Ukrainian home-improvement store ablaze (satellite photo)*. https://www.space.com/russia-ukraine-war-warehouse-fire-satellite-photo. Space.com. Accessed: 2026-05-14. Feb. 2022.

[2] D. Pehchevski. *The Impacts of Ukraine's Energy Sector on Air Quality*. Tech. rep. CEE Bankwatch Network and Ecoaction, Nov. 2020. URL: https://bankwatch.org/wp-content/uploads/2020/10/2020-10-19_Ukraine-air-quality-mapping_final2.pdf.

[3] L. Malytska et al. "The impact of fires on air quality in Ukraine during two years of military conflict (2022–2023): Analyzing satellite, ground-based observations of NO_2, CO, and aerosols". In: *Atmospheric Research* 338 (2026), p. 108959. ISSN: 0169-8095. DOI: 10.1016/j.atmosres.2026.108959.

[4] A. Shelestov et al. "Essential Variables for Air Quality Estimation". In: *International Journal of Digital Earth* 13.3 (2020), pp. 278–298. DOI: 10.1080/17538947.2019.1620881.

[5] D. Svidzinska. "Exploring the Potential of Citizen Sensing for Urban Air Quality Monitoring on the Example of Kyiv, Ukraine". In: *Proceedings of the XIV International Scientific Conference "Monitoring of Geological Processes and Ecological Condition of the Environment"*. 2020. DOI: 10.3997/2214-4609.202056043.

[6] B. Zhurakovskyi, O. Pliushch, V. Saiko, and V. Shutenko. "Machine Learning-based Environmental Monitoring and Analysis System". In: *CPITS 2025: Workshop on Cybersecurity Providing in Information and Telecommunication Systems*. 2025, pp. 183–192. URL: https://ceur-ws.org/Vol-3991/paper14.pdf.

[7] P. Carbone, M. Fragkoulis, V. Kalavri, and A. Katsifodimos. "Beyond Analytics: The Evolution of Stream Processing Systems". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. SIGMOD '20. Portland, OR, USA, 2020, pp. 1–8. DOI: 10.1145/3318464.3383131.

[8] S. Kumar. "The Evolution of Real-Time Data Streaming: Architectures, Implementations, and Future Directions in Distributed Computing". In: *World Journal*

of Advanced Research and Reviews 26.2 (2025), pp. 1004–1012. DOI: 10.30574/wjarr.2025.26.2.1746.

[9] M. M. Kamal, E. Kutafina, and O. Beyan. "Real-Time Visualization and Analysis Architecture for Data Integration Processes at Cologne University Hospital's Medical Data Integration Center". In: *Studies in Health Technology and Informatics*. 2024. DOI: 10.3233/SHTI240341.

[10] A. Akanbi and M. Masinde. "A Distributed Stream Processing Middleware Framework for Real-Time Analysis of Heterogeneous Data on Big Data Platform: Case of Environmental Monitoring". In: *Sensors* 20.11 (2020), p. 3166. DOI: 10.3390/s20113166.

[11] S. S. Kumar, R. Chandra, and S. Agarwal. *Rule Based Complex Event Processing for an Air Quality Monitoring System in Smart City*. 2024. arXiv: 2403.14701 [cs.DB]. URL: https://arxiv.org/abs/2403.14701.

[12] R. Sserunjogi et al. *Design and Evaluation of a Scalable Data Pipeline for AI-Driven Air Quality Monitoring in Low-Resource Settings*. 2025. arXiv: 2508.14451 [cs.SE].

[13] N. Joy. "Scalable Data Pipelines for Real-Time Analytics: Innovations in Streaming Data Architectures". In: *International Journal of Emerging Research in Engineering and Technology* 5.1 (2024), pp. 8–15. DOI: 10.63282/3050-922X/IJERET-V5I1P102.

[14] M. Raasveldt and H. Mühleisen. *DuckDB: An Embeddable Analytical Database*. https://duckdb.org. Proceedings of the 2019 ACM SIGMOD International Conference on Management of Data. 2019.

[15] European Environment Agency. *European Air Quality Index*. https://www.eea.europa.eu/themes/air/air-quality-index. Accessed: 2026-05-14. 2024.