Capstone Project
IT4043E : Big Data Storage and Processing

Flight Delays and Cancellations Monitoring System

Students :
Pham Anh Tu 20225463
Florian DORCHIES 20250072S
Eliott MOSKOWICZ 20250075S
Etienne FONTBONNE 20250006S

Contents

1 Abstract

This project details the implementation of a scalable Big Data solution for the real-time processing of flight data. The primary objective is to develop a comprehensive streaming pipeline capable of ingesting high-volume flight records, transforming raw operational data into immediate, actionable insights, and historical metrics. This system is designed to establish a robust data flow for continuous stream analytics, ultimately providing statistics for both operational monitoring and long-term historical analysis.

The core of this architecture is a stream-based pipeline that leverages industry-standard big data technologies for continuous ingestion, transformation, and aggregation. Our processing layer performs crucial data enrichment and executes complex real-time calculations to generate key performance indicators for air travel, such as carrier performance, detailed delay breakdowns, and route efficiency. The resulting analytical intelligence is stored in a multi-layered persistent store, catering to both low-latency demands for instant querying and a durable file system for long-term archival needs.

2 Introduction

2.1 Problem Statement

The air travel industry is a complex, high-stakes environment where efficiency and performance metrics are in constant flux. While vast quantities of operational data are generated every second, detailing flight movements, delays, cancellations, and causal factors, a critical challenge lies in transforming this volume and velocity of raw data into readily accessible and comparative intelligence. Existing, predominantly batch-processing architectures fail to keep pace, leaving operational analysts, business stakeholders, and service managers without the current, comparative performance statistics necessary for immediate, tactical decision-making.

This deficiency prevents key users from rapidly answering fundamental operational questions. For example, without real-time analytics, it is impossible to instantly determine which airline is currently operating most efficiently, identify the specific flight routes experiencing the longest average delays, or pinpoint which causal factor (e.g., weather, air system, maintenance) is the leading contributor to delays across the network right now. This lag between data generation and metric availability forces stakeholders to rely on outdated reports, hindering their ability to identify underperforming routes or carriers and implement timely corrective measures. The inability to offer an easily navigable, up-to-the-minute comparison of performance metrics such as on-time records, delay rankings, and efficiency scores is a significant barrier to maintaining competitive advantage and high service levels.

Therefore, the core problem is the absence of a real-time, user-centric data system capable of performing continuous comparative analysis on high-volume flight data and presenting immediate, actionable rankings and metrics to stakeholders. This project is dedicated to solving this problem by designing and implementing a robust, Kubernetes-deployed Big Data streaming pipeline that specifically focuses on generating and visualizing easily comparable, low-latency performance statistics, empowering users to make data-driven decisions that enhance operational effectiveness and overall service quality.

2.2 Scope and Limitations

3 Architecture and Design

3.1 Data description

The data used in this project originates from the "2015 Flight Delays and Cancellations" public dataset available on Kaggle. This dataset was originally published by the U.S. Department of Transportation's (DOT) Bureau of Transportation Statistics.

The dataset is distributed across three distinct files, which facilitates a relational modeling approach (joins) during the processing phase :

1. `flights.csv` : The main file containing detailed information for each specific flight.
2. `airline.csv` : A reference table linking the IATA code (e.g., "AA") to the full airline name (e.g., "American Airlines").
3. `airports.csv` : A reference table containing geographical details for airports (Name, City, State, Latitude, Longitude), identified by their IATA code (e.g., "JFK").

The main file contains 31 variables that are ingested and processed by our pipeline. These can be categorized into 4 groups :

Category	**Variables Included**
Temporal Information	YEAR, MONTH, DAY, SCHEDULED_DEPARTURE, DEPARTURE_TIME
Flight Identification	AIRLINE, FLIGHT_NUMBER, TAIL_NUMBER, ORIGIN_AIRPORT, DESTINATION_AIRPORT
Performance Metrics	DEPARTURE_DELAY, ARRIVAL_DELAY, AIR_TIME, DISTANCE
Status & Causes	CANCELLED, DIVERTED, CANCELLATION_REASON, *_DELAY (Weather, Security, etc.)

Table 1: Key Variables by Category

3.2 Overall Architecture

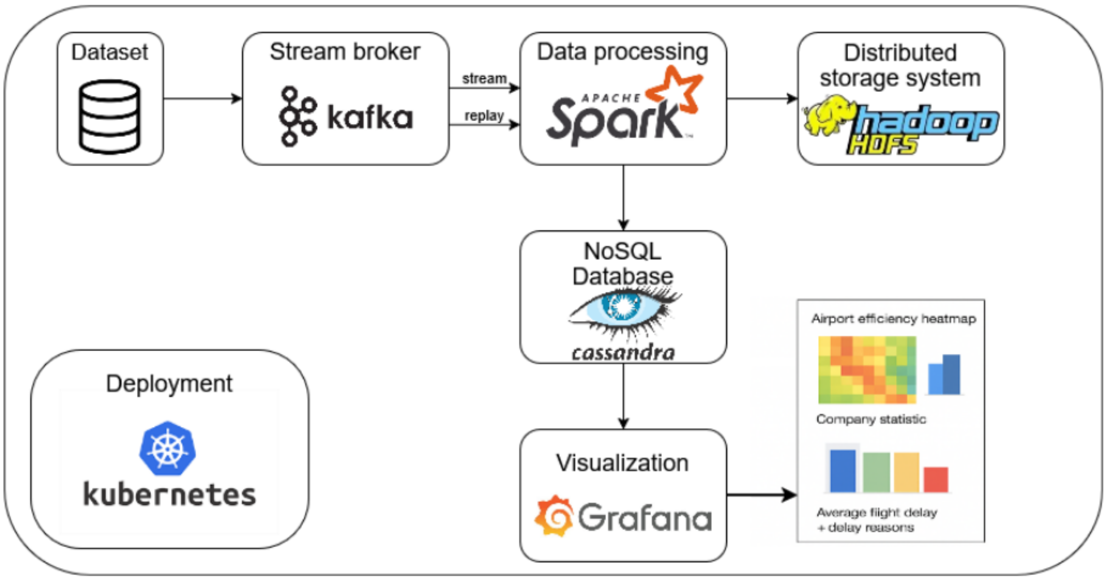


Figure 1: Overall system architecture for real-time flight delay analytics

As shown in Figure 1, the system follows a Kappa Architecture with Apache Kafka as the streaming backbone, Apache Spark for real-time processing, HDFS for distributed storage, Cassandra for analytical queries, and Grafana for visualization. Kappa Architecture is designed for stream-centric data processing, where all data is treated as a continuous flow and processed through a single streaming pipeline. Unlike Lambda Architecture, Kappa Architecture removes the need for a separate batch processing layer, relying entirely on streaming systems for both real-time and historical data processing. Historical analysis and reprocessing are achieved by replaying data from the streaming platform. A detailed description of the data flow is presented in Section 3.4.

3.2.1 Why Kappa Architecture is Suitable for This Project

- **Real-Time Requirements:** The project focuses on analyzing flight delays as data arrives, enabling near real-time insights into airline performance, route congestion, and delay patterns. Kappa Architecture supports continuous ingestion, processing, and visualization without relying on offline batch jobs.
- **Architectural Simplicity:** By eliminating the batch layer, the system architecture is simpler and easier to maintain. All flight records—both historical and incoming—are processed using the same Spark Structured Streaming pipeline, reducing operational overhead and code duplication.
- **Scalability:** Apache Kafka and Apache Spark are horizontally scalable, allowing the system to handle increasing data volume and velocity as more flight records or additional data sources are introduced.
- **Efficiency:** Given the moderate scale of the dataset, Kappa Architecture avoids the redundancy of maintaining separate batch and stream processing layers. A single streaming pipeline is sufficient to compute analytical results, leading to more efficient use of computational resources and reduced system complexity.

3.3 Components and their specific roles

3.4 Kafka

3.4.1 Description

Apache Kafka is a distributed event streaming platform designed for high volume and low latency data ingestion. In this architecture, Kafka serves as the central message backbone and handles the continuous influx of flight data.

3.4.2 Role in the system

- Centralized ingestion : Raw flight data is published to Kafka topics as JSON messages. This provides a single point of entry for all streaming records before any transformation is applied.
- Buffer : Kafka acts as a buffer between the data producers and the Spark processing engine. This ensures that bursts in data do not overwhelm following components in

the pipeline.

- Integration with Spark : Kafka provides the source for Apache Spark, which consumes the data using offset-based tracking. This integration supports "exactly-once" processing semantics by coordinating Spark checkpoints with Kafka offset management.

3.4.3 Producer and Consumer

In Kafka, the producer script reads raw data from a CSV file. It uses the `kafka-python` library to send each row of the CSV as a JSON object. As our dataset is very large, the producer is configured for performance and safety, we use compression and batch message to reduce bandwidth and achieve a higher throughput.

The consumer, connects to kafka bootstrap servers and subscribes to the flights-topic. It reads the stream of messages and provides them to Spark for real-time transformation.

3.5 Apache Spark

3.5.1 Description

Apache Spark is an open-source unified analytics engine designed for large-scale data processing. In this project, Spark serves as the core processing layer responsible for transforming high-volume flight event streams into structured analytical datasets.

The system is implemented using Spark Structured Streaming, which enables continuous processing of data ingested from Kafka using a micro-batch execution model. This approach supports scalable transformations, complex aggregations, and stateful computations while providing fault tolerance through checkpointing and event-time processing. Spark's rich DataFrame API allows the pipeline to combine real-time stream processing with advanced analytical operations, making it suitable for near real-time analytics and downstream visualization.

3.5.2 Role in the System

Within the system architecture, Apache Spark fulfills the following responsibilities:

- **Streaming Data Ingestion:** Consumes flight event streams from Kafka with controlled ingestion rates, offset management, and **backpressure** handling to ensure stable streaming execution.

- **Data Parsing and Type Normalization:** Parses JSON payloads using a robust schema strategy, followed by explicit type casting to normalize numeric and categorical fields while handling missing or malformed values.
- **Multi-stage Data Transformation:** Applies chained transformations including column derivation, **custom UDF**, timestamp generation, and filtering to prepare data for analytical processing.
- **Data Enrichment via Joins:** Enriches streaming flight records by joining with static airline and airport reference datasets, adding contextual and geographic attributes required for higher-level analytics. Static data is **cached** for better performance.
- **Complex Aggregations and Analytics:** Computes airline-level performance metrics, delay statistics by root cause, route-level delay analytics, and hourly time-series statistics using advanced aggregation functions.
- **Data Persistence and Output Management:** Writes aggregated results to Cassandra using exactly-once semantics and archives enriched streaming data to HDFS for offline analysis.
- **Runtime Monitoring and Data Quality Validation:** Executes per-batch data quality checks and logs key metrics to provide observability into streaming health and data correctness.

3.6 HDFS

3.6.1 Description

HDFS (Hadoop Distributed File System) is a distributed storage system designed to reliably store large volumes of data across clusters of commodity hardware. In this project, HDFS serves as the checkpoints storage layer, providing scalable and fault-tolerant data persistence ensuring the Spark streaming runs without errors. HDFS integrates smoothly with Spark Structured Streaming, by using the checkpoints option.

3.6.2 Role in the System

Within the overall system architecture, HDFS plays a supporting role:

- **Checkpoint Storage:** Provides a durable location for Spark Structured Streaming checkpoints, allowing recovery and consistent execution.
- **Docker Context:** Used primarily inside the Docker environment; outside of this scope, HDFS was not stable enough to be effectively used in the Kubernetes envi-

ronment.

3.7 Apache Cassandra

3.7.1 Description

Apache Cassandra is a distributed, wide-column NoSQL database designed for high availability, horizontal scalability, and high write throughput. In this project, Cassandra serves as the primary persistent storage layer for analytical results produced by the Spark streaming pipeline.

Cassandra is particularly well-suited for this workload due to its append-heavy write pattern, fault-tolerant architecture, and ability to scale linearly as data volume grows. The database schema is designed around query patterns required for downstream analytics and visualization rather than normalized relational modeling.

3.7.2 Role in the System

Within the system architecture, Cassandra fulfills the following roles:

- **Persistent Storage for Streaming Outputs:** Cassandra stores aggregated results generated by Spark Structured Streaming, ensuring durable and fault-tolerant persistence of real-time analytics.

  Schema-on-Write for Analytical Tables Cassandra schemas are explicitly defined to match the structure of aggregated outputs produced by the Spark streaming pipeline. Rather than storing raw events, the database follows a schema-on-write approach, where data is validated, transformed, and aggregated before insertion. This ensures consistency, query efficiency, and predictable performance for downstream analytics.

  The following analytical tables are maintained:

  – **airline_stats:** Stores airline-level performance metrics aggregated over streaming batches. This table captures counts of cancelled, on-time, and delayed flights, along with average departure and arrival delays. It is optimized for airline-centric analytics and dashboard visualizations.

  – **delay_by_reason:** Stores aggregated delay statistics grouped by delay category (e.g., weather, security, airline-related delays). For each delay reason, the table maintains the total number of occurrences and the average delay duration, enabling root-cause analysis of flight delays.

  - **route_stats:** Stores route-level delay analytics between origin and destination airports. This table captures geographic and city-level metadata alongside average arrival delays, supporting spatial analysis and route performance comparisons.
  - **hourly_stats:** Stores time-series statistics aggregated by scheduled departure hour. Metrics include flight volume, average delay, delay variability, median and tail latency (95th percentile), and cancellation counts, supporting temporal trend analysis.
- **High-Throughput Write Handling:** Cassandra efficiently handles continuous batch writes from Spark using append-only semantics, making it suitable for streaming workloads with frequent updates.
- **Query-Optimized Data Modeling:** The data model is denormalized to match access patterns used by monitoring dashboards and analytical queries, reducing the need for complex joins at query time.
- **Integration with Analytics and Visualization Tools:** Cassandra acts as the backend data source for Grafana dashboards, enabling near real-time visualization of airline performance, delay patterns, and route statistics.
- **Fault Tolerance and Availability:** Data is written using configurable consistency levels to balance availability and correctness, ensuring resilience against node or container failures in a distributed deployment.

3.8 Grafana

3.8.1 Description

Grafana is an open-source analytics and visualization platform designed for real-time monitoring and exploratory data analysis. In this project, Grafana serves as the primary visualization layer, providing interactive dashboards for monitoring flight performance metrics and delay patterns derived from the streaming data pipeline.

Grafana connects directly to Apache Cassandra as a data source, enabling near real-time visualization of pre-aggregated analytics produced by Spark Structured Streaming. By separating visualization concerns from data processing, the system maintains a clean architectural boundary between computation and presentation.

3.8.2 Role in the System

Within the overall system architecture, Grafana fulfills the following roles:

Table 2: Cassandra Analytical Tables

Table Name	**Primary Key**	**Description**
airline_stats	airline	Airline-level performance metrics, including flight counts by status and average departure and arrival delays.
delay_by_reason	delay_reason	Aggregated delay statistics grouped by delay category, capturing frequency and average delay duration.
route_stats	(original_airport, destination_airport)	Route-level delay analytics enriched with geographic and city metadata, supporting spatial and route comparison analysis.
hourly_stats	scheduled_hour	Time-series statistics aggregated by scheduled departure hour, including delay distributions, cancellation rates, and traffic volume.

- **Real-Time Analytics Visualization:** Displays airline performance metrics, delay statistics, route-level analytics, and hourly delay trends based on continuously updated data in Cassandra.
- **Geospatial Visualization:** Utilizes Grafana Geomap panels to visualize route-level delays and airport-to-airport connections using latitude and longitude attributes stored in Cassandra. We can only visualize routes to a specific chosen airport due to **Grafana limitation** on route visualization.
- **Query-Driven Dashboarding:** Executes parameterized CQL queries against Cassandra tables, leveraging schema-on-write designs to ensure low-latency dashboard performance.
- **User Interaction and Filtering:** Supports dynamic filtering by airline, route, city, or time range, enabling exploratory analysis without requiring direct access to the underlying data infrastructure.

3.9 Data Flow

3.9.1 Overview

The data flow in the flight delay analytics system consists of four main stages:

1. **Ingestion:** Flight dataset → Kafka topic (flight events).
2. **Processing:** Kafka → Apache Spark Structured Streaming (real-time transformation, enrichment, and aggregation).
3. **Storage:** Spark → HDFS (raw and intermediate data) and Cassandra (aggregated analytical tables).
4. **Visualization:** Cassandra → Grafana (interactive dashboards and real-time insights).

3.9.2 Data Ingestion

Source: The data originates from historical flight records stored in flights.csv, containing information such as airlines, routes, departure and arrival delays, cancellation status, and delay reasons.

Process:

- A Kafka producer reads the CSV files, converts each record into JSON format, and streams the data into a Kafka topic.
- Apache Kafka acts as a durable and fault-tolerant message broker, ensuring ordered delivery and data replay capabilities.

- Kafka retention enables historical reprocessing without maintaining a separate batch pipeline.

3.9.3 Data Processing

Component: Apache Spark Structured Streaming consumes flight events from Kafka and processes them in real time.

- **Input:** Kafka topic containing raw flight events in JSON format.
- **Transformations:**
  - Parse JSON messages and normalize schemas using robust **type casting**.
  - Read and **cached** static data from airlines.csv and airports.csv including additional IATA airline codes and geographic information of airports.
  - **Handle missing** or malformed delay values to ensure data quality. Drop unnecessary columns.
  - **Add Timestamp**: Append the current processing timestamp (in GMT+7) to each record for temporal tracking.
  - Enrich records with derived attributes such as scheduled hour and separate on time flights and delayed flights using **Custom UDF** according to US rules on flights.
- **Aggregations:**
  - Airline-level statistics (on-time, delayed, cancelled flights and calculate average delay).
  - Route-level average delays enriched with geographic information.
  - Hourly delay trends (avg, std, median, ...) and statistical summaries.
  - Delay statistics grouped by delay reasons.

3.9.4 Role of Timestamp

- **Temporal Tracking:** Each aggregated record includes an update timestamp to track when metrics were computed.
- **Real-Time Analysis:** Enables time-based filtering and trend analysis in Grafana dashboards.
- **Monitoring and Debugging:** Supports auditing, latency detection, and troubleshooting in the streaming pipeline.

3.9.5 Data Storage

Components:

- **HDFS:** Stores raw and intermediate datasets for archival, replay, and offline analysis.
- **Apache Cassandra:** Stores aggregated results in schema-on-write analytical tables optimized for dashboard queries.
- Airline performance metrics are stored in `airline_stats`.
- Route-level analytics are stored in `route_stats`.
- Delay distributions by reason are stored in `delay_by_reason`.
- Temporal delay patterns are stored in `hourly_stats`.

3.9.6 Data Visualization

Component: Grafana serves as the visualization layer of the system.

- Grafana connects directly to Cassandra using a dedicated data source plugin.
- Dashboards provide insights into airline performance, delay trends, route congestion, and hourly delay distributions.
- Geospatial visualizations enable exploration of route-level delays using geographic coordinates.
- Interactive filters and time ranges allow users to dynamically explore the data.

4 Implementation details

4.1 Source Code with Complete Documentation

The project source code is organized into modular directories, each corresponding to a major system component:

- **grafana/**: Contains dashboard definitions and data source provisioning files used to visualize flight delay metrics and system statistics.
- **spark/**: Includes Apache Spark Structured Streaming scripts responsible for real-time data transformation, aggregation, and analytics.
- **kafka/**: Holds the Kafka producer implementation that streams flight data into Kafka topics.
- **cassandra/**: Contains initialization scripts for creating keyspaces, tables, and indexes in Cassandra.

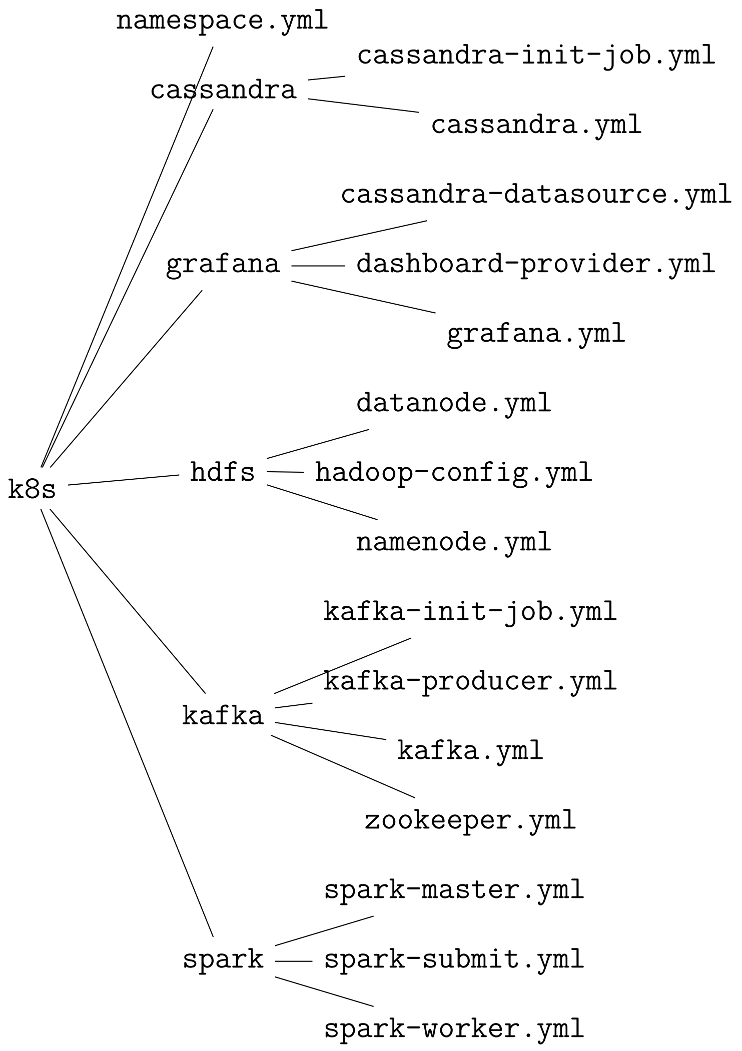
4.2 Environment-specific configuration files

- **docker-compose.yaml**: Defines and orchestrates all system services, enabling a reproducible multi-container deployment.
- **requirements.txt**: Specifies Python dependencies required by the Kafka producer.

4.3 Deployment strategy

The deployment strategy goal for this pipeline is to ensure scalability, resilience, and reproducibility. Each core service (Kafka, Zookeeper, HDFS NameNode/DataNode, Spark jobs, and Grafana) is encapsulated in its own manifest and deployed as a StatefulSet, Deployment or Job depending on its role. This approach guarantees a stable deployment.

The Kubernetes deployment is organized into a modular directory tree, where each component (Cassandra, Grafana, HDFS, Kafka, Spark) has its own configuration files. The overall structure is shown below:



Each directory encapsulates the manifests required to deploy and configure its respective service:

- **Cassandra:** initialization job and core deployment.
- **Grafana:** datasource configuration, dashboard provisioning, and deployment.
- **HDFS:** NameNode, DataNode, and Hadoop configuration.
- **Kafka:** broker, producer job, initialization, and Zookeeper.
- **Spark:** master, worker, and submission job definitions.
- **Namespace:** global namespace definition for resource isolation.

Key aspects of the deployment strategy include:

- **Namespace isolation:** All components are deployed under the `bigdata` namespace to simplify resource management and avoid conflicts.
- **StatefulSets for persistence:** Kafka, Zookeeper, Cassandra, and HDFS are deployed as StatefulSets with persistent volume claims to maintain data durability across pod restarts.
- **Jobs:** The Kafka producer and the Spark streaming are defined as Kubernetes Jobs, ensuring they run to completion and can be retried on failure. The same goes with the Cassandra and Kafka initialization process.
- **Resource limits:** CPU and memory requests/limits are defined for each pod to prevent resource starvation and to allow efficient scheduling.
- **Init containers:** Dependencies are checked before startup (e.g., waiting for Zookeeper before Kafka, waiting for NameNode before DataNode) to guarantee correct service ordering.
- **Scalability:** The architecture supports horizontal scaling by allowing to increase the number of replicas.

5 Lessons learned

5.1 Lesson 1: Containerization and Service Orchestration with Docker

5.1.1 Problem Description

Context and Background Initially, the system relied on the Bitnami Spark image for distributed processing and MongoDB as the NoSQL data store.

Challenges Encountered Several challenges emerged during the containerization phase:

- **Licensing constraints:** As of September 2025, the Bitnami Spark distribution was no longer freely available, making it unsuitable for academic and long-term use.
- **Monitoring limitations:** MongoDB lacked a stable and free Grafana data source plugin, limiting observability and dashboard integration.
- **Service orchestration complexity:** Coordinating startup order, health checks, and inter-service dependencies across Kafka, Spark, Cassandra, and HDFS required careful configuration.

These issues directly impacted deployment reliability, monitoring capabilities, and long-term maintainability of the system.

5.1.2 Approaches Tried

Approach 1: Local PySpark and Native Java Dependencies Spark jobs were executed locally using PySpark within notebook environments, with Kafka and Cassandra accessed externally.

Trade-offs:

- Simplified debugging and rapid iteration during early development.
- Suitable for validating data ingestion and transformation logic.
- Not deployable as a production pipeline due to environment inconsistency and lack of orchestration.

Approach 2: Docker-based Spark Distribution and Database Migration The system was re-architected using the official Apache Spark Docker images with a dedicated Spark master, worker, and submit container. MongoDB was replaced with Cassandra to enable seamless Grafana integration and improved write scalability.

Trade-offs:

- Increased initial setup complexity.
- Required explicit management of Spark dependencies, checkpoints, and resource limits.
- Provided a fully containerized, reproducible, and production-aligned environment.

5.1.3 Final Solution

The final architecture adopted a **hybrid development and deployment model**:

- **Local Spark** was used for exploratory analysis and rapid prototyping within Jupyter notebooks.

- **Dockerized Spark** (master, worker, and submit containers) was used for executing the streaming pipeline.
- **Cassandra** replaced MongoDB as the primary NoSQL store, enabling direct integration with Grafana for monitoring and analytics.
- Docker health checks, explicit service dependencies, and persistent volumes were employed to improve system stability and fault tolerance.

This approach balanced developer productivity with deployment robustness.

5.1.4 Key Takeaways

Technical Insights

- Container orchestration significantly reduces environment drift in distributed systems.
- Health checks and dependency ordering are critical for reliable startup in multi-service pipelines.
- Monitoring requirements should influence database and technology selection early in the design phase.

Best Practices

- Evaluate licensing and ecosystem support before committing to core infrastructure components.
- Separate exploratory workflows (notebooks) from production pipelines (script-based execution).
- Use persistent volumes for stateful services such as Kafka, Cassandra, and HDFS.

Recommendations

- Use notebooks for rapid experimentation and validation.
- Use Python scripts executed via `spark-submit` for reproducible, deployable pipelines.
- Prefer widely adopted, open-source distributions to minimize long-term operational risk.

5.2 Lesson 2: Kafka Performance Optimization

5.2.1 Problem Description

Context and Background : During the initial deployment of the streaming pipeline, the system struggled to maintain the "real-time" requirements of the air travel industry. While Apache Kafka was successfully ingesting flight data, the default configuration led to significant latency between data generation and availability for processing.

Challenges Encountered Several performance bottlenecks were identified during our test and verifications:

- High producer latency : Sending records individually from the CSV source resulted in excessive network overhead and low throughput.
- Ressource contention : Without defined limits, the Kafka brokers and Spark executors competed for CPU and memory within the Docker environment, causing instability.

5.2.2 Approaches Tried

Approach 1 : Default Producer Settings. The initial implementation used synchronous writes for every record to ensure durability. It guarantees message delivery but has an extremely slow ingestion rates that failed to simulate a high-velocity flight data stream.

Approach 2 : Unconstrained Micro-batching The system was configured with `startingOffsets: earliest` without a maximum offset limit. However, it attempted to process the entire 260,000+ record backlog in a single burst and resulted in memory overflows and prolonged execution hangs, delaying downstream writes.

5.2.3 Final Solution

The performance was optimized through a multi-layered approach to balancing throughput and latency :

- Batching and Compression : Enabled producer-side batching and GZIP compression to reduce network overhead and increase the volume of data handled per request.
- Flow Control : Introduced a limit of 1,000 offsets per trigger in the Kafka source configuration. This enforced incremental processing and eliminated the cold start hangs previously observed.
- Parallelism Tuning : Adjusted the number of Kafka partitions to match the number of Spark executor cores, ensuring that data could be processed in parallel across the cluster.

5.2.4 Key Takeaways

Controlling micro-batch size is essential for predictable startup behavior and stable streaming execution. Producer-side batching is critical for transforming high-volume raw data into a high-velocity stream without saturating the network.

5.2.5 Best Practices

We should always align kafka partition counts with Spark parallelism to maximize resource utilization and Monitor consumer lag continuously to identify if the processing layer needs to be scaled horizontally.

5.3 Lesson 3: Data Quality Management in Kafka–Spark Streaming Integration

5.3.1 Problem Description

Context and Background During integration of Kafka-based ingestion with Spark Structured Streaming, the pipeline exhibited incorrect aggregations and prolonged startup delays. Despite successful message consumption, computed delay metrics were consistently null or zero, and no data was written to Cassandra during the initial execution phase.

Root Cause Analysis Two primary data quality issues were identified:

1. **JSON Parsing Mismatch:** The Kafka producer converted CSV inputs to JSON, representing missing values as empty strings (`""`). The Spark schema defined several fields (e.g., `DEPARTURE_DELAY`) as `IntegerType`. Spark's `from_json` parser interprets empty strings for numeric fields as null, resulting in widespread null propagation and invalid aggregations.
2. **Streaming Cold Start Hang:** The stream was configured with `startingOffsets=earliest` without limiting batch size. On startup, Spark attempted to process the entire Kafka backlog (over 260,000 records) in a single micro-batch, causing a prolonged hang and delaying downstream writes.

5.3.2 Solution Implemented

Robust Schema Parsing

- Modified the initial schema to ingest all fields as `StringType`.
- Added explicit casting logic to convert strings to numeric types post-parsing.
- This approach correctly handles valid integers, negative values, and empty strings (converted to null), preserving data integrity.

Streaming Configuration Optimization

- Introduced `maxOffsetsPerTrigger=1000` in the Kafka source configuration.

- This enforced incremental processing, eliminated cold start hangs, and enabled immediate data persistence.

Data Quality Monitoring

- Implemented a per-batch data quality check that logs total record counts and null ratios for critical fields (e.g., `AIRLINE`, `DEPARTURE_DELAY`).
- Added explicit output flushing to ensure log visibility in containerized execution.

5.3.3 Verification Results

- **Streaming Logs:** Batch-level logs confirmed stable ingestion with expected null rates, indicating correct handling of missing values.
- **Cassandra Output:** Aggregation tables were populated with valid, non-zero metrics, including accurate average delays and route-level statistics.
- **Testing:** A dedicated unit test suite validated transformation logic across edge cases, including empty strings, valid integers, and negative values.

5.3.4 Key Takeaways

- Data type assumptions at ingestion boundaries can silently corrupt downstream analytics.
- Explicit parsing and casting provide stronger guarantees than implicit schema enforcement in streaming systems.
- Controlling micro-batch size is essential for predictable startup behavior in Kafka–Spark pipelines.
- Continuous data quality checks are critical for observability and early anomaly detection.

5.4 Lesson 4: HDFS Storage Technique

5.4.1 Problem Description

Context and background While setting up the pipeline, there were concerns on about how to make it more reliable and more precisely on how to handle the checkpoints used by the Spark processing.

Challenges encountered Setting up a Hadoop File System was challenging on many aspects. The first challenge was deciding which data should be stored in HDFS. Then, there have been several issues regarding the stability of its deployment and its ability to be a reliable part of the pipeline.

System impact This lack of stability had a direct impact on the pipeline, if the spark streaming could not write in HDFS, it would end up in failing every time.

5.4.2 Approaches tried

At first the idea of storing the enriched flights into HDFS was studied. Doing so would allow to have access to an archive of all the flights which is realistic. However, this idea had to be separated in two approaches : one the Docker deployment, and the other one on Kubernetes.

Docker approach: On this approach, a solution to have a stable HDFS deployment had been found, which made possible not only to write the Spark streaming checkpoints on it but also to be able to store enriched flights data to the HDFS node.

Kubernetes approach: While the Docker deployment of HDFS was stable, it was not the case of the one on Kubernetes. This lack of stability was compromising the whole pipeline, making it impossible to write effectively on HDFS without risking frequent failures.

5.4.3 Final Solution

While the solution on Docker was working and sufficient, the fact that it was not working on the final deployment, Kubernetes, led to abandoning this idea and forced the use of local Spark checkpoints, which are, supposedly, less stable than it would have been with HDFS.

5.4.4 Key Takeaways

- **Technical insights:** HDFS can provide reliable checkpointing and data archiving in controlled environments (e.g., Docker), but its stability in Kubernetes requires careful configuration and monitoring.
- **Best practices:** Always validate HDFS deployment in the target environment before relying on it for critical pipeline components. Use dedicated user directories and proper replication factors to ensure resilience.
- **Recommendations:** For production-grade deployments, consider managed HDFS services or alternative distributed storage systems to avoid instability.

5.5 Lesson 5: Kubernetes deployment

5.5.1 Problem Description

Context and background After setting up a working Docker pipeline, the next step was to switch to a more production-like deployment: Kubernetes.

Challenges encountered The conversion from Docker to Kubernetes was not as straightforward as imagined. In the process, many checks were introduced, but it became clear they were excessive and counterproductive.

System impact These issues led to instability in the pipeline and slowed down the deployment process, reducing overall reliability. Sometimes, the checks would kill the process before it was ready even though everything was perfectly fine. At the same time, some environment variables were too redundant, causing issues.

5.5.2 Approaches tried

Approach 1 At first the idea was to replicate the Docker setup directly in Kubernetes, keeping the same checks and environment variables. This approach quickly showed its limits: the checks were too strict and killed healthy processes, while redundant variables created confusion within the clusters.

Approach 2 The next attempt was to simplify the deployment by progressively removing unnecessary checks and cleaning up the environment configuration. This reduced complexity and improved stability, but required rethinking the deployment with Kubernetes-native practices rather than copying Docker logic.

5.5.3 Final Solution

The final solution aimed to have a deployment as simple as possible at first, which proved to be the right option. Kubernetes has a lot of strong options, like the environment variables or the initialization containers. The simpler proved to be the most effective in most cases.

5.5.4 Key Takeaways

Technical insights Direct conversion from Docker to Kubernetes is rarely feasible; each environment requires its own configuration that may not be compatible with Kubernetes.

Best practices Avoid excessive verification steps that add complexity without improving reliability. Focus on essential scheduling checks.

Recommendations When migrating from Docker to Kubernetes, redesign the deployment with Kubernetes-native practices instead of attempting a one-to-one conversion. Keep checks minimal and targeted to actual failure points.