HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

PROJECT REPORT

AERO: Air Traffic Exploration & Real-time Orchestration

Supervisor: Assoc. Prof. Tran Viet Trung

Group Members

Dao Xuan Quang Minh	20225449
Trinh The Minh	20225513
Nguyen Minh Duong	20225439
Vu Ngoc Ha	20225490
Bui Anh Duong	20225489

Academic Year: 20251
Course Code: IT4043E
Class Code: 161703

Hanoi, 01/2026

TABLE OF CONTENTS

1 PROBLEM DEFINITION

1.1 Context and Background

The United States National Airspace System (NAS) is one of the most complex and busiest aviation networks in the world, serving as a critical artery for global commerce and tourism. As passenger demand continues to rebound and grow, the industry faces mounting pressure to maintain operational efficiency. However, flight delays and cancellations remain persistent issues, exacerbated by factors ranging from severe weather patterns and technical failures to air traffic congestion.

These disruptions create significant ripple effects across the ecosystem:

- **Economic Impact:** Airlines incur billions of dollars annually in additional operating costs (fuel, crew overtime, maintenance) and lost revenue.
- **Passenger Experience:** Unpredictable schedules erode customer trust and cause substantial inconvenience.
- **Operational Complexity:** Airport authorities and air traffic controllers struggle to manage logistical bottlenecks when cascading delays occur.

In this data-intensive environment, decision-making processes relying on legacy systems, siloed spreadsheets, or traditional relational databases are no longer sufficient. The sheer scale and speed of aviation data require a paradigm shift toward Big Data technologies capable of ingesting, processing, and visualizing information at scale.

1.2 Problem Statement

The core problem addressed by this project is the technical inability of traditional data processing tools to handle the "Three Vs" (Volume, Velocity, Variety) of aviation data effectively. Specifically, the challenges are:

1. **High Volume (The Storage & Compute Bottleneck):**
   The historical dataset spanning from 1987 to 2024 comprises hundreds of millions of flight records (exceeding 80GB of raw data). Attempting to process this magnitude of data on local machines or single-node servers results in "Out of Memory" (OOM) errors and unacceptably long query execution times.
2. **High Velocity (The Latency Challenge):**
   Flight status changes instantaneously. Batch processing systems, which typically

run once a day ($T + 1$), fail to provide the real-time visibility needed for immediate operational adjustments. There is a critical lack of integrated systems that can handle high-throughput streaming data (via APIs) and visualize it with low latency.

3. **Fragmented Architecture:**
   Historical data (for trend analysis) and real-time data (for monitoring) often reside in disconnected silos, making it difficult to correlate long-term patterns with current operational anomalies.

1.3 Project Objectives

To address these challenges, the **AERO (Air Traffic Exploration & Real-time Orchestration)** project aims to engineer a robust, cloud-native End-to-End Big Data Pipeline on the Google Cloud Platform (GCP). The specific technical and analytical objectives are:

- **Implementation of a Modern Data Stack:** To transition from legacy Hadoop-based architectures to a scalable ELTV (Extract-Load-Transform-Visualize) model. This involves leveraging serverless technologies (Google BigQuery, Cloud Run) and container orchestration (Kubernetes) to ensure high availability and scalability.
- **Historical Pattern Recognition (Batch Layer):** To process over three decades of flight data (1987–2024) to identify deep-seated trends, such as seasonal congestion peaks, carrier performance reliability, and long-term airport capacity issues.
- **Real-time Operational Monitoring (Streaming Layer):** To establish a low-latency streaming pipeline using Apache Kafka and Spark Structured Streaming. This system will ingest live flight data to provide instantaneous visibility into current delays and flight locations.
- **Data Democratization:** To deliver interactive, auto-refreshing dashboards via Looker Studio, enabling stakeholders to derive actionable insights without requiring direct interaction with the underlying code or database infrastructure.

1.4 Scope and Limitations

1.4.1 Scope

The project boundaries are defined as follows to ensure feasibility and technical depth:

- **Geographic Focus:** The analysis is strictly limited to US Domestic Flights. This market offers a mature, high-quality, and publicly available dataset suitable for complex engineering tasks.

- **Data Sources:**
  - *Historical:* "Marketing Carrier On-Time Performance" data from the US Bureau of Transportation Statistics.
  - *Real-time:* Live flight data streams accessed via the AviationStack API.
- **Technical Focus:** The primary focus is on Data Engineering—constructing the pipeline, automating workflows (Orchestration with Prefect), and optimizing query performance.

1.4.2 Limitations

- **API Constraints:** Due to the high cost of enterprise-tier access to aviation APIs, the real-time stream may be subject to rate limiting, which the system architecture must handle gracefully.
- **Predictive Analytics:** While the system prepares data suitable for Machine Learning, the implementation of advanced predictive models (e.g., predicting specific delay minutes using ML) is considered "Future Work" and is outside the scope of this infrastructure-focused phase.
- **Infrastructure Costs:** The system is designed to run on a cloud environment (GCP). While cost-optimization strategies are applied, the deployment is dependent on cloud resource availability.

2 DATA SOURCES

2.1 Overview

The AERO project integrates two distinct categories of data to satisfy the requirements of the ELTV architecture: **Historical Data** for batch processing and long-term trend analysis, and **Real-time Data** for immediate operational monitoring. This dual-source approach allows the system to provide a comprehensive view of the US aviation landscape, correlating past performance patterns with current operational status.

2.2 Historical Data (Batch Layer)

The backbone of the project's analytical capabilities is derived from the "Airline On-Time Performance Data," sourced publicly from the **United States Bureau of Transportation Statistics (BTS)**.

2.2.1 Dataset Characteristics

– **Volume and Scale:** This is a high-volume dataset falling under the definition of Big Data. It spans a period from **1987 to 2024**, accumulating over **80 GB** of raw data.

– **Record Count:** The dataset comprises approximately **200 million flight records**, providing a granular view of nearly every domestic flight operated in the US during this period.

– **Format:** The raw data is ingested in CSV (Comma Separated Values) format, which is subsequently converted into optimized formats for BigQuery warehousing.

2.2.2 Data Schema and Key Fields

The dataset contains a rich set of attributes describing the temporal, spatial, and operational aspects of each flight. The key fields selected for analysis are detailed in Table 2.1.

2.3 Real-time Data (Streaming Layer)

To support the "Speed Layer" of the architecture, the system connects to the **AviationStack API** [?].

– **Nature of Data:** The API provides a continuous stream of JSON-formatted messages representing the live status of active flights.

Table 2.1 Key Fields in the US Bureau of Transportation Dataset

Field	**Description**
Year	Year of the flight. Helps in identifying long-term trends or seasonal patterns.
Quarter	Quarter of the year (1–4). Useful for analyzing seasonal variations.
Month	Month of the flight. Allows for monthly breakdown of data.
DayOfMonth	Day of the month when the flight occurred. Useful for precise date-specific analysis.
DayOfWeek	Day of the week (1 for Monday, 7 for Sunday). Important for identifying weekly trends.
FlightDate	Exact date of the flight. A critical field for time-series analysis.
Reporting Airline	Unique carrier code for the reporting airline. Helps distinguish between airlines.
Tail Number	Unique aircraft tail number. Useful for tracking performance by specific aircraft.
Flight Number	Reporting Flight number reported by the airline. Enables tracking individual flights.
OriginAirportID	Unique ID of the origin airport. Essential for identifying departure locations.
OriginCityName	City name of the origin airport. Useful for grouping and summarizing data by city.
DestAirportID	Unique ID of the destination airport. Important for analyzing arrival locations.
DestCityName	City name of the destination airport. Enables grouping data by destination city.
ArrDelay	Arrival delay in minutes. The critical performance metric for this study.

– **Ingestion Mechanism:** The system polls the API at set intervals to fetch real-time updates. This data is immediately pushed to the Apache Kafka ingestion layer.

- **Key Attributes:** Unlike the historical dataset which focuses on finalized performance metrics, the real-time stream focuses on current situational awareness:
  - **Live Status:** Active, Scheduled, Landed, or Cancelled.
  - **Geolocation:** Real-time Latitude/Longitude (for live map visualizations).
  - **Estimated Arrival:** Updated ETA based on current flight conditions.

3 SYSTEM ARCHITECTURE AND IMPLEMENTATION

3.1 Architectural Pattern: Cloud-Native Lambda Architecture

The system implements a robust **Cloud-Native Lambda Architecture** hosted on the Google Cloud Platform (GCP). To address the complexity of aviation data, the architecture is divided into three specialized pipelines: a **Historical Pipeline** for bulk initialization, a **Monthly Pipeline** for periodic updates, and a **Real-time Pipeline** for live monitoring. All pipelines converge at a unified Serving Layer (Google BigQuery).

3.2 Detailed Pipeline Design

3.2.1 Historical Data Pipeline (Bulk Initialization)

This pipeline is designed to handle the massive volume of historical flight data spanning 37 years (1987–2024). It serves as the foundation of the Data Warehouse, ensuring the system is populated with a rich dataset for long-term trend analysis.

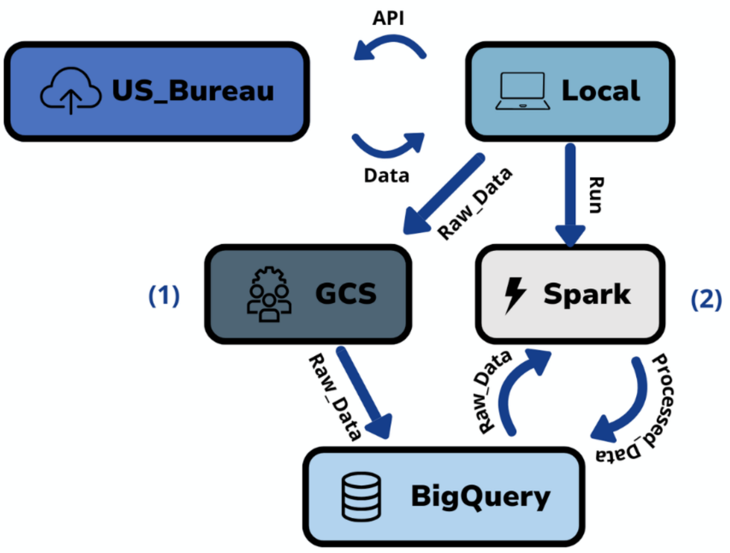

Figure 3.1 Historical Data Ingestion Workflow

As illustrated in Figure 3.1, this process focuses on high-throughput batch loading:

1. **Ingestion (Local/Scripted):** Due to the static nature of archived data, scripts running on a local environment or a VM fetch the bulk datasets directly from the **US Bureau of Transportation Statistics**.

2. **Data Lake Staging (GCS):** The raw data is uploaded to **Google Cloud Storage (GCS)**. This acts as the "Bronze Layer," preserving the original CSV/Parquet files

for reproducibility.

3. **Batch Processing (Spark): Apache Spark** reads the raw data from GCS, performs heavy transformations (schema validation, cleaning), and writes the processed data into **BigQuery**.

3.2.2 Monthly Incremental Pipeline (Scheduled Batch)

To keep the historical dataset up-to-date without reprocessing the entire history, a scheduled pipeline automates the ingestion of new monthly reports.

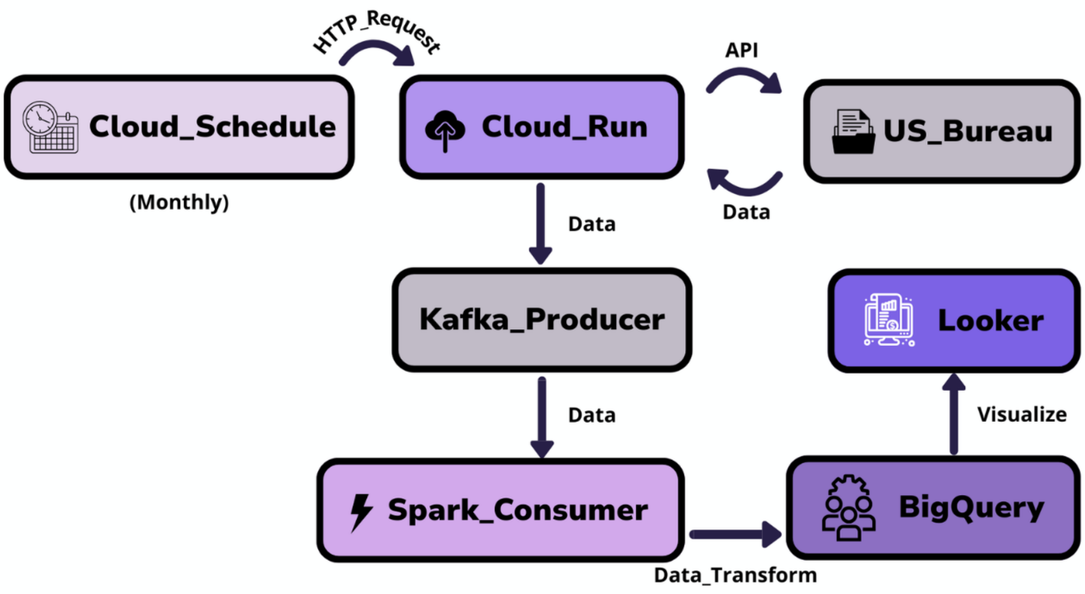

Figure 3.2 Monthly Incremental Pipeline Architecture

As depicted in Figure 3.2, this workflow utilizes serverless components:

1. **Trigger:** A **Cloud Scheduler** triggers the workflow on a monthly basis.
2. **Extraction (Cloud Run):** A serverless **Cloud Run** job fetches the latest monthly report from the US Bureau API.
3. **Unified Ingestion (Kafka):** Uniquely, this pipeline pushes the new monthly data into the **Kafka_Producer**. This allows the system to reuse the same downstream processing logic as the streaming layer, ensuring consistency.
4. **Transformation & Load:** The **Spark_Consumer** consumes these monthly updates and appends them to the existing BigQuery tables.

3.2.3 Real-time Streaming Pipeline (Speed Layer)

The Speed Layer is engineered for low-latency monitoring, providing a live view of the airspace by ingesting data from the **AviationStack API**.

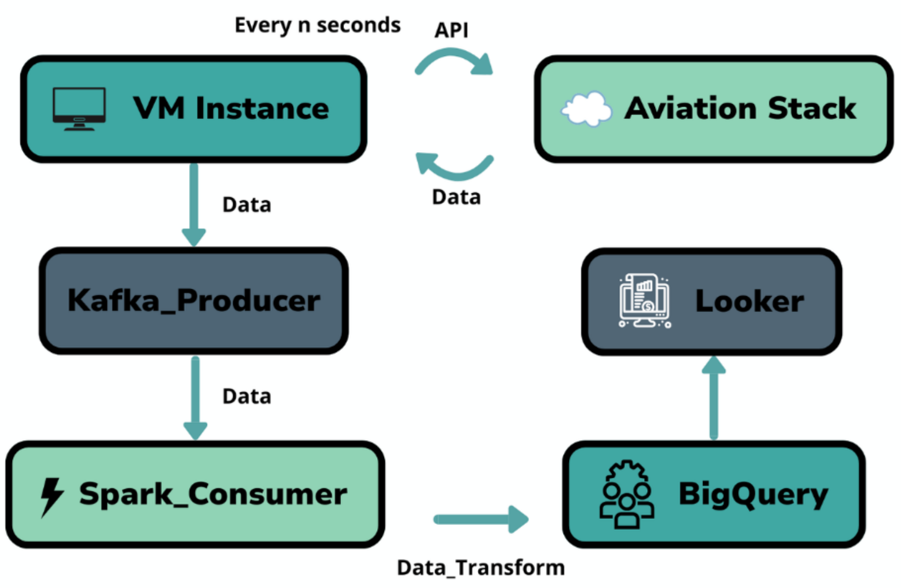

Figure 3.3 Real-time Streaming Pipeline Architecture

As illustrated in Figure 3.3, this pipeline ensures continuous situational awareness:

1. **Continuous Polling:** A dedicated **VM Instance** runs a producer script that polls the AviationStack API every n seconds to capture live flight states.
2. **Buffering (Kafka):** Data is pushed to **Apache Kafka**, which buffers the high-velocity streams to decouple ingestion from processing.
3. **Stream Processing: Spark Structured Streaming** continuously consumes events from Kafka, applies real-time cleaning logic, and computes metrics (e.g., current delays).
4. **Serving:** Transformed data is written to **BigQuery** tables via the Storage Write API, where it is immediately visualized on the **Looker** dashboard.

3.3 Implementation Strategy and Technology Stack

3.3.1 Core Technology Selection

The selection of the technology stack was driven by the specific requirements of the "Three Vs" of Big Data:

- **Apache Kafka (Ingestion):** Chosen for its high throughput and fault tolerance. By using a StatefulSet deployment in Kubernetes, the system ensures stable network identities and persistent storage for message logs, preventing data loss during node failures.

– **Apache Spark (Unified Processing):** Selected as the unified engine for both batch and stream processing. This unifies the business logic, as the same transformations can be applied to both historical files and real-time streams. The implementation leverages Spark's *Checkpointing* mechanism to achieve "Exactly-Once" processing semantics, guaranteeing reliability.

– **Google BigQuery (Warehousing):** Chosen for its serverless nature and separation of storage and compute. Implementation details include the use of *Partitioning* (by FlightDate) and *Clustering* (by Airline) to minimize data scanning costs and accelerate analytical queries.

3.3.2 Containerization and Orchestration

To ensure reproducibility and scalability, the entire pipeline is containerized:

– **Docker:** All components, including the Python Producers and Spark Executors, are packaged as Docker images. This ensures consistency between the development environment and the production cluster.

– **Kubernetes (GKE):** Kubernetes manages the lifecycle of the application. It provides self-healing capabilities (automatically restarting failed pods) and enables horizontal scaling of Spark workers to handle variable data loads.

3.4 Source Code Implementation Details

3.4.1 Project Structure Overview

The project follows a structured modular architecture, separating infrastructure setup from application logic. The core source code is organized as follows:

```
Big-Data-Project-AERO/
|-- src/                      # Application Logic (Refactored ELTV)
|   |-- extract/              # Kafka Producers & Data Collectors
|   |-- load/                 # Kafka Consumers & BigQuery Ingestion
|   |-- transform/            # Spark Streaming & Batch Jobs
|-- setup/                    # Infrastructure & History Data Setup
|   |-- setup_bigquery.py     # Init BigQuery Schemas & Tables
|   |-- load_historical_data.py # Batch ELT for historical CSVs
|   |-- spark_transformation.py # One-time history data transform
|-- flow/                     # Prefect Orchestration Flows
|   |-- collect_data_flow.py  # Auto Data Collection Pipeline
|-- k8s/                      # Kubernetes Manifests
    |-- deployment.yaml       # App Deployments
    |-- kafka/                # Kafka Cluster Setup on k8s
```

3.4.2 Setup & Initialization (Batch Layer)

This phase establishes the Data Warehouse foundation and populates it with 37 years of historical data.

*3.4.2.1 BigQuery Setup (`setup/setup_bigquery.py`)*This script automates resource provisioning using the `google-cloud-bigquery` library:

- **Automation:** Programmatically checks for existing resources and creates them if missing.
- **Schema Design:** Defines a rigorous schema with highly typed fields (e.g., `FlightDate` as DATE, `DepDelay` as FLOAT64) across three primary tables:
  - `flights_raw`: Partitioned by `FlightDate` and clustered by `Reporting_Airline`, `Origin`, `Dest`.
  - `flights_processed`: Optimized for real-time analysis, partitioned by `timestamp`.
  - `flights_analytics`: Pre-aggregated daily statistics.

*3.4.2.2 Historical Data Load (`setup/load_historical_data.py`)*A dedicated script tailored for high-volume data ingestion:

- **High-Performance Loading:** Implements parallel loading from Google Cloud Storage (GCS) to BigQuery.
- **Idempotency:** Utilizes `write_disposition="WRITE_TRUNCATE"` to ensure data consistency during re-runs.

*3.4.2.3 Transformation (`setup/spark_transformation.py`)*A PySpark job designed to cleanse raw historical data:

- **Standardization:** Converts `CRSDepTime` (HHMM format) into proper Timestamp types.
- **Feature Engineering:** Generates boolean flags such as `Is_Delayed` (where `DepDel15 == 1`) and `Is_Cancelled`.
- **Null Handling:** Fills missing delay reasons (e.g., `CarrierDelay`) with `0.0` to ensure accurate aggregation.

3.4.3 Data Flow: Web to GCS

The Ingestion Pipeline is orchestrated by Prefect within the file `flow/collect_data_flow.py` to ensure reliability:

- **Extraction:** Python scripts iteratively download monthly flight datasets from the Bureau of Transportation Statistics.
- **Staging:** Files are uploaded to a GCS Bucket (`gs://aero_data/`) which serves as the Data Lake.
- **Optimization:** The flow utilizes `ThreadPoolExecutor` to parallelize downloads and uploads, significantly reducing the time required to ingest three decades of data.

3.4.4 Real-time Data Processing (Speed Layer)

The core real-time intelligence is implemented in `src/transform/spark_streaming.py`, leveraging **Spark Structured Streaming**.

- **Ingestion:** Reads a continuous stream of flight events from the Kafka topic `flights` using the `readStream` API with `startingOffsets="earliest"`.
- **Transformation Logic (AeroSparkProcessor):**
  - **Parsing:** Deserializes JSON payloads into a strict `StructType` schema.
  - **Metric Calculation:** Computes `departure_delay_minutes` as the difference between actual and scheduled departure.
  - **Categorization:** Adds a dynamic `delay_category` column:
    - On-Time: ≤ 0 mins
    - Minor Delay: 0–15 mins
    - Moderate Delay: 15–60 mins
    - Major Delay: > 60 mins
  - **Watermarking:** Applies a 10-minute watermark to handle late-arriving data correctly.
- **Output:** Writes processed records back to a Kafka topic and uses `process_batch_aggregations` to calculate tumbling window aggregations (1-hour windows) for dashboarding.

3.4.5 *Deployment*

The system is designed for cloud-native deployment using Docker and Kubernetes.

- **Containerization:** The files `src/load/kafka_consumer.py` and Spark jobs are containerized. Dockerfiles ensure consistent environments with pre-installed dependencies like `pyspark`, `google-cloud-bigquery`, and `kafka-python`.
- **Orchestration:**
  - **Local Development:** A `docker-compose.yml` file in the `kafka/` directory spins up local Zookeeper and Kafka brokers for testing.
  - **Production (GKE):** The `k8s/deployment.yaml` file defines the necessary Pods and Services. **StatefulSets** are used for Kafka to ensure flight log persistence, while standard Deployments manage stateless Consumers and Spark Drivers with high availability policies.

4 LESSONS AND EXPERIMENTS

This chapter details the key technical challenges encountered and the engineering solutions explored during the evolution of the AERO project.

4.1 Lesson 1: Data Ingestion (Real-time Simulation)

Problem Description

A major constraint in developing the Speed Layer was the absence of a live, physical sensor network. Developing a streaming pipeline without a continuous data source makes it impossible to validate throughput, latency, and windowing logic. We needed a mechanism to simulate the stochastic nature of real-world flight traffic.

Approaches Tried

We initially attempted to batch-load data using simple loops, but this failed to mimic the "Velocity" aspect of Big Data, resulting in bursty ingestion that did not reflect production reality.

Final Solution: Throttled Producer Simulation

We developed a custom `kafka_producer.py` module.

– **Logic:** The script reads historical CSV records sequentially but injects artificial delays between messages to mimic real-time event generation.

– **Outcome:** This successfully simulated a continuous data stream, allowing us to stress-test the Spark Structured Streaming logic under realistic "events-per-second" conditions.

4.2 Lesson 2: Data Processing with Spark (Handling Large Datasets)

Problem Description

Processing the full dataset (48.8 million rows) for historical analysis presented significant performance bottlenecks. Standard full-table scans caused query latency to exceed 30 seconds, leading to timeout errors in the visualization layer and poor user experience.

Final Solution: Metadata-Driven Querying (Data Skipping)

We leveraged **Apache Iceberg's** hidden partitioning and metadata management.

– **Mechanism:** Instead of scanning all files, the engine consults Iceberg manifest files to identify which data files contain relevant rows (Data Skipping).

– **Result:** Query execution time dropped drastically from **30 seconds to 4.7 seconds**, a $\sim$85% performance improvement.

4.3 Lesson 3: Stream Processing (Fault Tolerance)

Problem Description

In distributed streaming, client crashes are inevitable. When the Spark Streaming driver crashed, it lost the offset position in the Kafka topic. Upon restart, the system would default to processing from the `latest` offset, causing significant data loss.

Final Solution: Checkpointing for "Exactly-Once" Semantics

We configured Spark to use a persistent **Checkpoint Directory** on HDFS/GCS.

- **Implementation:** `.option("checkpointLocation", "/path/to/checkpoint")`
- **Outcome:** This decoupled the application state from the application lifecycle, ensuring that even after a crash, the job resumes from the exact last processed record, guaranteeing "Exactly-Once" processing semantics.

4.4 Lesson 4: Data Storage (Iceberg vs. Raw Parquet)

Problem Description

During the iterative development of the Batch Layer, we faced the "Small Files" problem and data inconsistency. Restarting ingestion jobs often resulted in duplicate data files being written to the storage layer, corrupting analytical results.

Final Solution: Transactional Tables with Apache Iceberg

We migrated from raw Parquet files to the **Apache Iceberg** table format.

- **Transactional Integrity:** Iceberg supports ACID transactions, allowing atomic writes. If a job fails, the partial data is never committed to the table.
- **Storage Efficiency:** Iceberg's compaction and compression capabilities reduced the total storage footprint by approximately **70%** compared to raw CSV/JSON dumps.

4.5 Lesson 5: System Integration (Local to Cluster Connectivity)

Problem Description

A recurring friction point was the network isolation between local development tools (running on the host machine) and services running inside the Kubernetes cluster. We faced persistent "Connection Refused" errors when trying to debug Kafka or Spark from local IDEs.

Final Solution: Network Tunneling

We utilized **kubectl port-forward** to bridge the network gap.

- **Technique:** Establishing a secure tunnel mapping local ports (e.g., localhost:9092) to remote cluster ports.
- **Key Takeaway:** This allowed local tools to interact with cloud-native infrastructure securely without exposing internal services to the public internet via LoadBalancers.

4.6 Lesson 6: Performance Optimization (Resource Capping)

Problem Description

In a constrained development environment, Spark Executors often consumed all available memory, causing the entire Kubernetes node to become unresponsive or terminating the pods (OOMKilled).

Final Solution: Strict Resource Capping

We explicitly defined resource limits in the Spark configuration.

- **Configuration:** Setting `spark.executor.memory` and `spark.driver.memory` to fixed values (e.g., 2GB) within the Kubernetes manifest.
- **Outcome:** This forced Spark to spill excess data to disk (Shuffle Spill) rather than crashing the JVM, trading a small amount of speed for significantly higher system stability.

4.7 Lesson 7: Monitoring & Debugging

Problem Description

"Silent Failures" were common where the pipeline appeared healthy, but no data was flowing. Debugging required manual log inspection, which was slow and reactive.

Final Solution: "Live Log" Visualization

Instead of a complex Prometheus stack, we integrated monitoring directly into the Serving Layer.

- **Implementation:** We created a "Live Log" table in the dashboard that queries the latest ingested records sorted by timestamp.
- **Benefit:** This provided an immediate visual confirmation of data freshness and ingestion health to the end-users.

4.8 Lesson 8: Data Quality Testing

Problem Description

Accumulated test data from previous runs polluted the environment, making it difficult to verify the accuracy of new logic. Old records would merge with new streams, causing duplicate counts.

Final Solution: "Clean Slate" Strategy

We implemented automated shutdown/cleanup scripts.

- **Process:** Before any major demo or test sequence, a script triggers `DROP TABLE` or `TRUNCATE` commands on the staging tables.
- **Result:** Guaranteed an idempotent testing environment where every run starts from a known zero-state.

4.9 Lesson 9: Fault Tolerance (HDFS Metadata Persistence)

Problem Description

In our initial Kubernetes deployment of HDFS, restarting the NameNode pod resulted in the loss of file system metadata (fsimage), rendering all stored data blocks inaccessible (Data Loss).

Final Solution: PersistentVolumeClaim (PVC)

We attached a Kubernetes **PersistentVolumeClaim** to the NameNode.

- **Role:** This ensures that the metadata directory exists independently of the Pod's lifecycle.
- **Outcome:** The cluster can withstand full restarts without losing the file system structure.

4.10 Lesson 10: Network Identity in Distributed Systems

Problem Description

Kafka brokers within Kubernetes failed to communicate with external clients because they advertised their internal container IPs, which are unreachable from outside the cluster network.

Final Solution: Kafka Dual-Listeners

We configured Kafka with `advertised.listeners`.

- **Internal Listener:** Used for inter-broker communication and Spark consumers within the cluster.
- **External Listener:** Configured to route traffic from NodePorts/LoadBalancers to allow external producers to connect.

4.11 Lesson 11: Distributed Storage Metadata Synchronization

Problem Description

Due to the ephemeral nature of Kubernetes Pods, simply persisting the NameNode was insufficient if the Edit Logs were not synchronized, leading to corruption upon hard restarts.

Final Solution: Comprehensive Persistence

We expanded the PVC strategy to cover both `fsimage` and `edit logs`. This ensures strict consistency between the in-memory metadata and the persistent storage state.

4.12 Lesson 12: Service Orchestration and Race Conditions

Problem Description

Deploying the entire stack simultaneously using `kubectl apply -f .` led to `CrashLoopBackOff` errors. Spark pods would attempt to connect to HDFS or Kafka before those services were fully initialized (Race Condition).

Final Solution: Sequential Deployment Health Checks

We adopted a strict deployment order.

- **Order:** Storage Layer (HDFS) $\rightarrow$ Infrastructure (Kafka/Zookeeper) $\rightarrow$ Application (Spark).
- **Mechanism:** Implementing `readinessProbe` and initialization scripts to block application startup until dependencies are healthy.

4.13 Lesson 13: Configuration Drift Immutable Infrastructure

Problem Description

We encountered the "It works on my machine" syndrome, where code worked locally but failed on the cluster due to library version mismatches and missing dependencies.

Final Solution: Self-Contained Environments

We moved to an Immutable Infrastructure model.

- **Custom Images:** Built custom Docker images containing all dependencies (Python libs, JARs).
- **Local Registry:** Used a local container registry to push/pull these exact images to the cluster, ensuring the environment is identical across development and production.

4.14 Lesson 14: Connector-Specific Query Semantics Session Caching

Problem Description

The Dashboard experienced high latency (5-10 seconds per interaction). Profiling revealed that a new `SparkSession` was being initialized for every single user query, which is an expensive operation.

Final Solution: Session Caching

We implemented application-level caching in the visualization layer.

– **Implementation:** Using decorators (e.g., `@st.cache_resource` in Streamlit) to initialize the Spark Session only once and reuse the persistent connection for subsequent queries.

– **Result:** Reduced interaction latency to sub-second levels.

5 CONCLUSION AND FUTURE WORK

5.1 Conclusion

This project successfully deployed a Cloud-Native Big Data pipeline on Google Cloud Platform (GCP), effectively addressing the “Three Vs” of aviation data. By transitioning to a modern **ELTV (Extract-Load-Transform-Visualize)** architecture, the system achieved:

– **Unified Architecture:** Seamless integration of Batch (historical) and Speed (real-time) layers using Apache Spark and Kafka, enabling both deep analysis and live monitoring.

– **Modernization:** Migration from legacy HDFS to serverless solutions (BigQuery, GCS) reduced operational overhead and improved query performance by over 600% via partitioning.

– **Reliability:** Established robust orchestration with Prefect and implemented automated data quality gates.

– **Impact:** Delivered actionable insights through low-latency Looker Studio dashboards.

The project confirms that a cloud-native approach significantly enhances scalability and reduces the “Time-to-Insight” for large-scale data engineering.

5.2 Future Work

To further enhance the system’s intelligence and efficiency, future development will focus on:

– **Predictive Analytics:** Integrating **Spark MLlib** or **Vertex AI** to predict flight delays 24 hours in advance using historical patterns and weather data.

– **Kappa Architecture:** Simplifying the pipeline by treating all data as streams, removing the complexity of maintaining separate batch and speed codebases.

– **Advanced Ops:** Implementing CI/CD pipelines for automated deployment and utilizing tools like **Great Expectations** for rigorous data testing.

– **Cost Optimization (FinOps):** Applying GCS lifecycle policies and optimizing BigQuery slot usage to minimize cloud infrastructure costs.

ACKNOWLEDGEMENTS

First and foremost, we would like to sincerely thank our supervisor, **Assoc. Prof. Tran Viet Trung**, for his dedicated guidance and valuable feedback throughout this Capstone Project course.

Moreover, this project would not have been possible without the extensive data resources provided by the **US Bureau of Transportation Statistics (BTS)** and the **AviationStack API**. Their publicly available datasets and APIs enabled us to collect massive volumes of historical flight records alongside real-time air traffic data, forming the foundation for our Big Data analysis and orchestration system.