HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

PROJECT REPORT

TAXI TRIP ANALYSIS SYSTEM
FOR NYC TLC YELLOW TAXI RECORDS

Course: Big Data Storage and Processing

Supervisor: Assoc. Prof. Tran Viet Trung

Authors:	Student ID:
Nguyen Duc An	20225432
Mai Viet Bao	20225474
Nguyen Minh Duc	20225437
Do Tuan Nam	20235535
Vu Nhat Nguyen Thu	20225462

Hanoi, January 2026

ABSTRACT

This project presents the design and implementation of a scalable big data analytics system based on the Lambda Architecture for New York City taxi trip data. The system integrates Apache Kafka as a distributed message broker, Apache Hadoop Distributed File System (HDFS) for fault-tolerant data lake storage, Apache Spark for both stream and batch processing, Apache Cassandra for the serving layer, and Grafana for visualization.

Taxi trip records, sourced from NYC TLC datasets, are ingested into Kafka topics as JSON messages. The processing pipeline bifurcates into two parallel layers: the Speed Layer and the Batch Layer. In the Speed Layer, Spark Structured Streaming consumes data in real-time, applying event-time processing with watermarking to compute low-latency aggregated metrics over sliding windows. Concurrently, the Batch Layer archives raw data to HDFS in Parquet format to ensure long-term data durability. Periodic batch jobs re-process this historical data to generate high-accuracy daily aggregations, correcting potential late-arrival or out-of-order data issues inherent in the streaming path.

The entire pipeline is containerized using Kubernetes in Docker to ensure reproducibility and ease of deployment. This project demonstrates how modern stream-processing technologies can be combined to build a scalable, fault-tolerant, and near real-time analytics system suitable for urban mobility data.

Contents

1 Problem Definition

1.1 Selected Problem

Large metropolitan areas generate a continuous stream of transportation data through taxis, ride-hailing services, and other mobility providers. Although these data sources contain valuable information about travel demand, congestion, operational efficiency, and economic activity, they are often analyzed in a delayed or fragmented manner. As a result, governments, transportation companies, and citizens lack timely and accessible insights into how urban mobility systems operate in real time.

The problem addressed in this project is the development of a real-time urban mobility analytics and visualization system that transforms streaming trip-level data into actionable insights. Using the Yellow NYC Taxi dataset as a representative case, the system visualizes key indicators such as trip demand by zone, revenue distribution, fare efficiency, average speed, payment behavior, and peak versus off-peak travel patterns. The objective is not to perform isolated analysis on a single dataset, but to demonstrate how real-time data processing can support continuous monitoring and informed decision-making in urban transportation systems.

1.2 Motivation for Big Data–Driven Analysis

This problem is inherently suitable for Big Data technologies due to the scale, speed, and complexity of transportation data in large cities. Urban taxi systems generate millions of trip records over time, each containing spatial, temporal, and financial attributes. In a real-world environment, these records are produced continuously, requiring systems capable of ingesting and processing data streams with low latency. Furthermore, the data spans multiple dimensions, including location, time, distance, fare, and payment information, which must be aggregated and analyzed across different temporal windows and geographic zones.

Traditional batch-oriented processing is insufficient for such requirements, as it limits visibility into current system conditions. Big Data streaming architectures enable continuous aggregation, real-time metric computation, and live dashboard updates, allowing stakeholders to observe trends as they emerge rather than after they have already passed.

1.3 Scope and Limitations

1.3.1 Scope

The scope of this project is to design and implement a scalable streaming analytics pipeline that simulates real-time taxi trip data and presents insights through interactive dashboards. The system focuses on descriptive and operational analytics by presenting real-time metrics, time-series trends, and simple comparisons between different zones. While the NYC Taxi dataset is used for implementation and evaluation, the system is intentionally designed to be dataset-agnostic, allowing it to be adapted to other cities or transportation providers with similar trip-based data structures.

From a stakeholder perspective, the system is envisioned as a shared analytical tool. Government agencies can use it to monitor congestion, evaluate policy impacts, and support urban planning. Transportation companies can analyze demand patterns, revenue distribution, and operational efficiency. At the same time, researchers and citizens can benefit from improved transparency and public access to aggregated mobility data.

1.3.2 Limitations

Several limitations are acknowledged in this work. First, the real-time data stream is simulated by replaying historical trip records rather than being collected from live vehicles, which may not fully capture real-world variability. Second, access to high-frequency, standardized transportation data remains a significant challenge in many cities. Unlike New York City, where trip data is centrally collected and publicly released, similar data flows may not yet exist or be openly accessible in other urban contexts. Issues related to data ownership, inter-agency coordination, financial resources, and technical infrastructure are therefore considered outside the scope of this project. Finally, the system is designed primarily for visualization and operational monitoring, and does not address predictive modeling, demand forecasting, or automated decision-making.

1.4 Data Source and Streaming Simulation

1.4.1 The NYC TLC Yellow Taxi Dataset

The Yellow NYC Taxi dataset contains trip-level records collected from yellow taxis operating in New York City. The dataset is published and maintained by the New York City Taxi and Limousine Commission (TLC), the government agency responsible for regulating taxi services in the city. Taxi operators are required to submit electronic trip records through approved service providers, which ensures consistent data collection.

Each trip record describes a completed taxi trip and includes information about pickup and drop-off time, trip distance, payment type, fare amount, and pickup and drop-off zones. Locations are represented using standardized TLC Taxi Zone identifiers, which allow trips to be grouped by zone and borough without using raw GPS coordinates.

The dataset is released publicly as monthly files and covers multiple years of historical data. Due to its large size, clear structure, and public availability, it is commonly used in academic projects and urban transportation studies. In this project, the dataset is used as an example input source to design and evaluate a real-time data streaming and visualization system, rather than to study New York City specifically.

Details of the dataset schema and the associated zone lookup resources are provided in Appendix A.

1.4.2 Simulating Real-Time Flow

Since historical datasets are static by nature, this project simulates real-time data flow by first sorting trip records by drop-off timestamp and then splitting the data across multiple

producers operating at a controllable emission rate. Each producer represents an independent data source, such as a vehicle or a dispatch station, and continuously publishes trip events to the streaming system. This setup allows the system to emulate concurrent data generation, simulate varying traffic loads, and evaluate the behavior of the pipeline under multi-producer and high-throughput conditions. As a result, streaming analytics, window-based aggregations, and real-time dashboard updates can be tested without requiring access to proprietary live data feeds.

2 Architecture and Design

2.1 Overall Architecture

The system is designed following the *Lambda Architecture*, which separates data processing into a real-time processing path and a batch processing path. This design supports low-latency analytics while also ensuring accurate and complete historical results.

Figure 1 illustrates the overall system architecture.

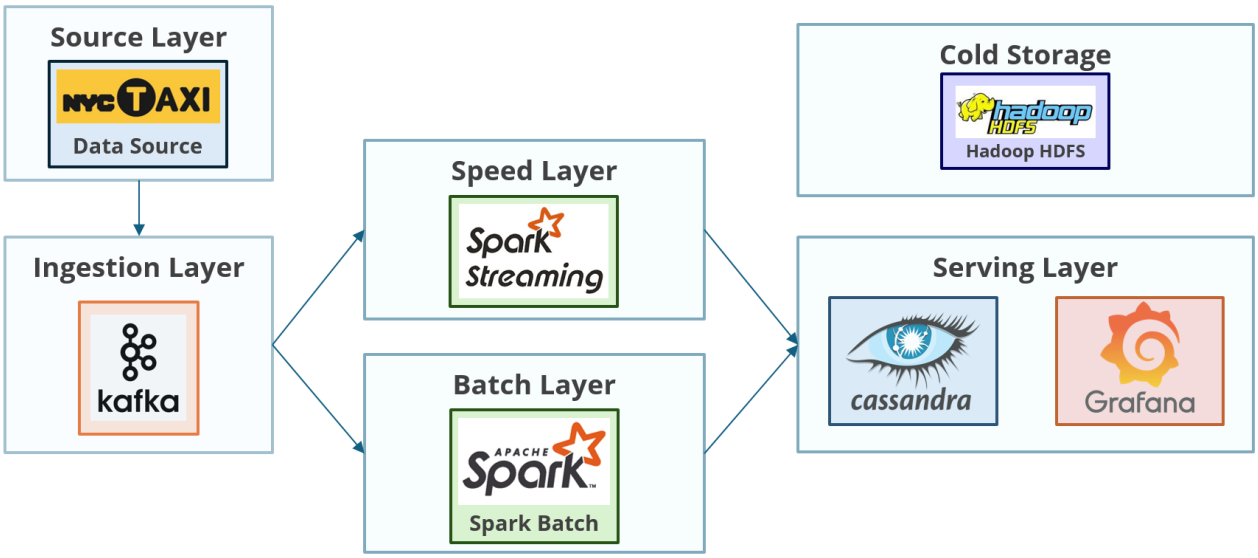

Figure 1: System architecture following the Lambda Architecture pattern

In the **Speed Layer**, trip events are published to Apache Kafka, which acts as the central ingestion layer. Apache Spark Structured Streaming consumes events from Kafka and performs real-time, window-based aggregations. The aggregated results required for dashboards are written to Apache Cassandra, which provides low-latency access for visualization. Grafana queries Cassandra to display real-time metrics and time-series trends.

In the **Batch Layer**, the same events are independently consumed from Kafka by a Spark batch application. This batch job processes the data in larger intervals and writes the results to the Hadoop Distributed File System (HDFS) for long-term storage and offline analysis. Data stored in HDFS is partitioned by event date to support efficient querying and historical exploration.

Apache Kafka serves as the shared data source for both the speed and batch layers. By reading from the same event stream, the two layers remain logically consistent while operating under different latency and storage requirements. The speed layer prioritizes fast updates for monitoring, while the batch layer focuses on durable storage and completeness of historical records.

The entire pipeline is deployed on a Kubernetes cluster, which manages container orchestration, scalability, and fault isolation for system components. Kafka brokers, Spark streaming and batch applications, HDFS services, Cassandra, and the data ingestion gateway are all deployed as Kubernetes workloads.

This architecture supports real-time dashboards, scalable stream processing, and reliable historical data storage. The Lambda Architecture is well suited to the project's focus on monitoring and visualization, where low-latency insights and long-term data retention are both required.

2.2 System Components and Their Roles

2.2.1 Apache Kafka

Apache Kafka is a distributed, fault-tolerant event streaming platform that buffers high-throughput data from our taxi trip generator.

Role: decoupling the data generation from processing. In our Lambda architecture, Kafka's can retain data logs and allow different consumer components to consume and "replay" historical data at custom rates whenever we need to recompute metrics or backfill data, which is fundamental to the architecture.

2.2.2 Apache Spark Structured Streaming

Apache Spark Structured Streaming is the computational core of the system. It provides a unified, high-level API for stream processing built on the Spark SQL engine.

Role: We utilize Spark to ingest data from Kafka, perform complex transformations (such as windowed aggregations for 30-minute and 1-hour intervals), and handle late-arriving data with watermarking. Its "exactly-once" processing semantics and ability to run in both continuous and "available-now" (batch) modes make it the ideal engine for our unified pipeline.

2.2.3 Apache Cassandra

Apache Cassandra is our serving layer, chosen for its high scalability and write-optimized nature.

Role: It stores the aggregated metrics (e.g., zone performance, global KPIs) produced by Spark. Cassandra's wide-column store structure allows for extremely fast writes of time-series data and efficient reads by partition keys (e.g., querying a specific zone's history), making it suitable for powering our real-time dashboards.

2.2.4 Apache Hadoop HDFS

Apache Hadoop Distributed File System (HDFS) serves as the storage foundation for our Batch Layer.

Role: Unlike Cassandra which stores aggregated metrics for the Speed Layer, HDFS is responsible for storing the raw, immutable "Master Dataset" in Parquet format. This allows us to perform comprehensive historical analysis, data recovery, and deep learning tasks on the full granularity of data, ensuring the "correctness" of our system over time as per the Lambda Architecture paradigm.

2.2.5 Grafana

Grafana is the visualization layer that sits on top of Cassandra.

Role: It provides monitoring and operational visibility. By connecting directly to Cassandra, Grafana allows us to build dynamic dashboards that visualize key metrics (like revenue per zone, average speed, and tip ratios) in near real-time, enabling immediate insights into the taxi network's performance.

2.3 Data Flow and Component Interaction

2.3.1 End-to-End Streaming Pipeline

The data lifecycle in our system follows a linear path designed for low-latency processing and scalability:

1. **Ingestion:** The `Producer` application generates synthetic taxi trip data and publishes it to the `taxi-trips` topic in **Kafka**.

2. **Processing: Spark Structured Streaming** subscribes to the Kafka topic. It absorbs the raw stream, applies the unified business logic (cleaning, enriching, and aggregating), and manages state.

3. **Storage:** Processed results are written to **Cassandra** tables. Spark acts as the Cassandra client, efficiently batching writes to the distributed store.

4. **Visualization: Grafana** periodically queries Cassandra to update the live dashboards, providing the end-user interface.

2.3.2 Kubernetes Deployment

To ensure portability and consistence with production standards, we deploy the entire stack on Kubernetes using **Kind (Kubernetes in Docker)**. Kind allows us to simulate a realistic multi-node Kubernetes cluster locally using Docker containers as nodes. This approach was chosen over simpler Docker Compose setups to give us access to native Kubernetes primitives like `StatefulSets` (crucial for stateful services like Kafka and Cassandra) and `Headless Services` for stable network identities. It provides a "production-ready" local environment that

validates our configuration manifests (separation of Core, Data Layer, and Apps) before any cloud deployment.

2.4 Implementation Details

2.4.1 Source Code Organization

Our codebase is organized to separate concerns effectively as per the pipeline requirements, which includes the following major components:

- **Ingestion Gateway:** A FastAPI-based HTTP gateway (`gateway.py`) acts as a buffer between the raw data generator and Kafka. This decouples the simulation logic from the message broker and allows for batch producing optimization.
- **Spark Structured Streaming Job:** Located in `spark/streaming.py`, this is the single source of truth for all business logic. It handles both stream and batch modes via command-line arguments.
- **Cassandra Schema:** Defined in `cassandra/init.cql`, using specific partition keys (e.g., `pickup_zone`, `shard_id`) to optimize for read-heavy dashboard queries.
- **Grafana Dashboards:** Provisioned as code in `grafana/provisioning`, ensuring that dashboard configurations are version-controlled and reproducible.

2.4.2 Data Processing with Spark

- **Windowed Aggregations:** We use event-time based windowing (e.g., `window(col("event_time"), "30 minutes")`) to group taxi trips. This ensures that metrics like "total revenue per 30 mins" reflect the actual time trips occurred, not when they were processed.
- **Batch Layer Ingestion (HDFS):** To support the Batch Layer, we implemented a dedicated stream that writes processed data to HDFS in Parquet format. We use a `processingTime="30 seconds"` trigger to batch writes. This balances the trade-off between data freshness and the "small files problem" in HDFS, ensuring efficient storage usage while keeping the Data Lake updated.
- **State Management and Late Data Handling:** A specific problem with our NYC Taxi data is that data only arrive at drop-off time (or even later) while the pickup timestamp is needed for indexing and aggregation. Upon examining historical data, 99.9% trip duration are within 1 hour 48 minutes. We therefore enable lenient Watermarking (`.withWatermark("event_time", "2 hours")`). This instructs Spark to maintain the state of older windows for up to 2 hours, allowing 99.9% of trips (including late-arriving data) to arrive at the correct pool before the state is discarded.

- **Exactly-Once Processing Semantics:** We achieve end-to-end exactly-once guarantees by using Spark's checkpointing mechanism (`/tmp/spark_checkpoints`) and Kafka's offsets. If a failure occurs, the application restarts from the last committed offset, ensuring no data is lost or counted twice.

2.4.3 Advanced Transformations and Aggregations

- **Multi-Stage Transformations:** The raw JSON stream goes through a multi-step pipeline: Parsing → Filtering (removing invalid trips) → Enrichment (adding time-based features) → Aggregation.
- **Broadcast Join Strategy:** For enriching trips with Zone information (mapping LocationIDs to text names), we utilize a **Broadcast Hash Join**. Since the `taxi_zone_lookup` table is small and static, broadcasting it to all worker nodes avoids expensive shuffles and significantly speeds up the enrichment phase.

2.4.4 Environment-Specific Configuration

- **Kafka Configuration:** We use a single-node definition in local development but define `StatefulSets` in Kubernetes to tolerate pod restarts. We use `ADVERTISED_LISTENERS` to differentiate between internal pod communication and external access.
- **Spark Configuration:** The Spark job is tuned with `spark.cassandra.connection.keepAliveMS` to maintain persistent connections to the database, reducing connection overhead during high-throughput verification.
- **Cassandra Configuration:** Tables are designed with `CLUSTERING ORDER BY (window_start DESC)` to ensure that the most recent data appears first in our time-series queries, optimizing dashboard load times.

2.4.5 Deployment Strategy

- **Containerization and Orchestration:** All components are containerized using Docker. In our Kubernetes deployment, we use `Deployment` objects for stateless apps (Gateway, Spark) and `StatefulSets` for stateful storage (Kafka, Cassandra) to ensure stable network identities and persistent storage volumes.
- **Service Networking:** We utilize Kubernetes `Services` (ClusterIP) for internal service discovery (e.g., `broker:9092`, `cassandra:9042`). This abstract naming ensures that our application code remains unchanged regardless of where the pods are actually scheduled in the cluster.

2.4.6 Monitoring Setup

- **Metrics Collection:** We rely on Grafana's direct connection to Cassandra to visualize business metrics.

- **Logging and Observability:** Standard output logs from Spark and the Gateway are captured by the container runtime (Docker/Kubernetes) for debugging. Spark's UI (`4040`) is exposed for detailed job inspection.

3 Lessons and Experiments

3.1 Lesson 1: Data Ingestion

Problem Description

The NYC Yellow Taxi dataset represents a classic *Edge-to-Cloud* data engineering challenge. In a real-world deployment, the data originates from telemetry units embedded in vehicles—environments defined by limited hardware resources, intermittent network connectivity, and constrained power budgets. The primary challenge was bridging the gap between low-level legacy hardware (taximeters, GPS modules, simple timers) and modern, high-throughput Big Data infrastructure. Our goal was to design an ingestion scheme that minimized the computational footprint on the taxi's onboard computer (often running stripped-down Linux distributions) while ensuring high-fidelity, low-latency data transmission to our central pipeline.

Approaches Tried

During the design phase, we evaluated two distinct architectural patterns for moving data from the "Edge" (the taxi) to the "Core" (Kafka):

- **Approach 1: Direct Kafka Production.** In this model, the onboard unit acts as a full Kafka Producer. We simulated this by having a script read the CSV and push messages directly to the Kafka Broker using Python's `confluent_kafka`. While fast and easy to simulate, it tightly coupled the simulation logic with the message broker, making it harder to scale the generator independently. In a real deployment with thousands of vehicles, managing persistent TCP connections to the Kafka cluster would lead to connection exhaustion on the brokers.
- **Approach 2: RESTful Gateway.** We introduced an intermediate abstraction layer: a FastAPI-based web server acting as a gateway. The taxi simply sends a standard HTTP POST request, and the server handles the complexity of writing to Kafka. This introduced a new component to maintain (the API server). However, it dramatically simplified the client-side requirements.

Final Solution

We selected **Approach 2** and implemented a high-performance Ingestion Gateway using **FastAPI** and `confluent_kafka` abstracted away from virtual taxi. In this architecture, the taxi's onboard hardware is decoupled from the message broker. The transmission logic on the taxi is reduced to a simple `cURL` command or a lightweight HTTP request, which is native to almost all embedded systems. The FastAPI gateway accepts the JSON payload, performs schema

validation (using Pydantic models to reject malformed data at the door), and asynchronously produces the record to the Kafka topic `taxi-trips`.

During our simulation, we encountered HTTP overhead with sending 1 row per request and a non-zero chance of malformed trip data (e.g. timer reporting the date in 1970). To facilitate higher speedup during simulation and protect the further pipeline, we added a batch reception endpoint (`/trips`) and aggregated trips before polling POST request every few seconds. This trick allowed simulation speedup from a maximum of 70x (about 1 minute in simulation per IRL second) to 3600x (about 1 hour in simulation—approx. 1500 trips per IRL second).

Key Takeaways

In most cases, it's best to "disconnect" data generation fleet and broker (in our case, by adding a middleware API endopint). Message queue like Kafka is a convenient technology but still not as easy to configure as simple HTTP requests, especially on edge hardware. In addition, adding basic "common sense" validation at the gate can alleviate a lot of workload downstream, contributing to a faster and more "real-time" pipeline.

3.2 Lesson 2: Data Processing with Spark

Problem Description

Data processing in Python-based environments often tempts developers to use familiar libraries like Pandas or arbitrary Python functions. However, when we initially attempted to process the stream using naive Python `lambda` functions and converting micro-batches to local Pandas DataFrames (`toPandas()`), the performance was unacceptable. The warmup time to serialize these function closures to executors took 2-3 minutes, and processing a single 30-minute window took nearly a minute, causing the streaming consumer to lag behind the Kafka producer in our simulation.

Approaches Tried

- **Approach 1: Python-Native Loops & UDFs.** We treated Spark merely as a distributed task runner for Python code, using `rdd.map()` and typical Python control flow. This incurred massive serialization overhead (pickling data between JVM and Python workers) and disabled Spark's Catalyst optimizer. While being easy to program, it's missing out a lot of the performance.
- **Approach 2: Catalyst-Optimized Transformations.** We rewrote the entire pipeline using only native Spark SQL expressions (`pyspark.sql.functions`). We replaced Python `if-else` logic with `when().otherwise()`, and replaced explicit iterations with optimized `window()` aggregations. It required tight matching with data source via `StructType`, `StructField`, type annotations and more complex syntax.

Final Solution

Our final implementation operates entirely within the JVM context managed by PySpark's JVM gateway, never materializing data back to Python objects on workers.

- **Vectorized Operations:** By using native Spark expressions (e.g. `col("trip_distance") / col("duration")`), we leverage code generation that runs at C-like speeds.
- **Broadcast Hash Join:** Instead of a standard shuffle join for the zone lookup, we force a `broadcast()`.

This change reduced the "warmup" time to under a minute and cut micro-batch processing latency for hourly tables from about a minute to a mere couple of seconds, allowing real-time synchronization with Cassandra even in stressful simulation.

Key Takeaways

In PySpark, Python is just the API layer. Despite having bridging functions and methods for legacy purposes, it's best to avoid using them and take the time to migrate for maximum performance.

3.3 Lesson 3: Stream Processing

Problem Description

Designing a robust streaming pipeline required balancing latency, accuracy, and resource usage. We faced three core challenges: determining the right aggregation windows for different business needs, managing memory pressure from unbounded state (late data), and ensuring data consistency in the event of worker failures. Naively running a "Daily Revenue" aggregation as a continuous stream, for instance, would cause the state store to grow linearly throughout the day, eventually leading to Out-Of-Memory (OOM) crashes.

Approaches Tried

Regarding the varying aggregation window serving different business needs, we experimented with 2 approaches:

- **Approach 1: Watermark Abuse.** We extended the watermark duration corresponding to the aggregation window (e.g., 1-day watermark for daily data). This consequently stored thousands of trips in RAM (about a few GBs), which was not a problem in a few seconds of our simulation, but was likely at Out-Of-Memory (OOM) and RAM failure risks if kept throughout the day.
- **Approach 2: Scheduled Batch Job.** We scheduled jobs that run once every period of time to aggregate needed data. This likely added more code to maintain, but was more reliable and computationally efficient.

Regarding the method of running periodic batch jobs, we found 2 possible approaches:

- **Approach 1: Spark Streaming with Checkpoints.** We utilized the existing streaming script but changed the processing method with `writeStream.trigger(availableNow=True)`. This took advantage of Spark checkpoints to achieve Exactly-Once processing efficiency while also minimize the codebase to maintain.
- **Approach 2: Scheduled Batch Job with Checkpoints.** We scheduled a different processing script that run once every period of time to aggregate needed data. The checkpoint of last seen data was stored manually as a JSON file on a persistent storage. This likely added more code to maintain but was a little more flexible in terms of specific transformations.
- **Approach 3: Scheduled Batch Job reprocessing everything.** That different processing script reprocessed past data depending on the broker's data replay capability. This ensured the highest data accuracy but was demanding in every other way.

Final Solution

We implemented a multi-layered strategy tailored to the specific nature of our data:

- **Business-Aligned Windowing:** We split aggregations into "Tactical" (15-30 mins) for immediate traffic control logic and "Strategic" (1 hour) for revenue monitoring.
- **Optimal Watermarking:** We analyzed the arrival distribution and set a watermark of **2 hours** (`.withWatermark("event_time", "2 hours")`). This specific duration covers 99.9% of late-arriving trips while safely discarding tail events to free up memory.
- **Hybrid State Management:** For long-term (e.g., 24-hour) aggregations, we purposefully *do not* use the streaming engine. Instead, we triggered a "Batch Mode" run of the same script (`-mode batch`) nightly. This allows us to calculate daily stats statelessly, avoiding the massive RAM cost of holding a 24-hour state window open.

Key Takeaways

Stream processing is an exercise in compromise. **Recovery and Exactly-Once** semantics are handled via checkpointing (`spark-checkpoints-pvc`), but memory stability requires strict watermarking. Don't use streaming for everything & offload long-window aggregations to batch jobs to keep your streaming micro-services lightweight and resilient.

3.4 Lesson 4: Data Storage

Problem Description

Designing the storage layer for a streaming pipeline required balancing raw data preservation for reprocessing with fast query responses for real-time dashboards. Storing raw trip records and aggregating them on-demand would not scale for dashboard queries expecting sub-second latency.

Approaches Tried

- **Approach 1: Single Storage System.** Initially, we considered using one database for both raw storage and serving. However, no single system excels at both high-throughput writes of raw events and low-latency analytical queries.
- **Approach 2: Query-time Aggregation.** Computing aggregations (e.g., revenue per zone, trips per hour) at query time would require scanning massive amounts of raw data, leading to unacceptable latency for dashboard refresh rates.

Final Solution

We adopted a **Lambda Architecture** with separated storage concerns:

- **HDFS** stores raw trip events partitioned by `event_date`, preserving data fidelity for batch reprocessing.
- **Cassandra** serves pre-aggregated results using **Query-Driven Modeling**. We created specific tables for each dashboard view (e.g., `zone_performance_30m`, `global_kpis_30m`, `peak_analysis_30m`) with `CLUSTERING ORDER BY (window_start DESC)`.

Spark Structured Streaming performs the aggregations upstream (windowed groupBy operations) before writing to Cassandra. This means Cassandra only serves the final numbers, delivering millisecond-level query responses.

Key Takeaways

In Big Data serving layers, store the *answers*, not the raw data. Use HDFS or similar distributed file systems for raw data preservation, and design your NoSQL schema (especially in Cassandra) based strictly on your query patterns—one table per query.

3.5 Lesson 5: System Integration

Problem Description

Integrating five distinct distributed systems (Kafka, Spark, Cassandra, Grafana, and the Python Gateway) revealed deeper complexity than component-level development. In the early stages, our system was fragile: if the Kafka broker was slow to start, the Spark job would crash immediately. Same case when Cassandra took too long to spin up. If Cassandra was under heavy load, the Grafana dashboard would hang indefinitely rather than failing gracefully. We needed robust patterns to handle the inherent unreliability of distributed networks.

Approaches Tried

- **Approach 1: "Hope and Pray."** We initially assumed all services would be up 100% of the time. We hardcoded startup orders in Docker Compose (`depends_on`), which failed in Kubernetes where pod scheduling is non-deterministic.

- **Approach 2: Infinite Retries.** We added simple ‘while True‘ retry loops. This solved startup crashes but created "Retry Storms" where failing services were hammered by thousands of reconnection attempts, preventing them from recovering.

Final Solution

We adopted a defensive integration strategy:

1. **Service Discovery:** We completely decoupled physical IP addresses from application logic. All components communicate via Kubernetes **ClusterIP Services** (e.g., ‘kafka.big-data.svc:9092‘). This allows K8s to transparently load-balance requests across pods (if we were to scale replicas) and handle pod IP changes automatically.

2. **Error Handling Timeouts:** We replaced indefinite waits with strict timeouts across the stack. In the Gateway, ‘producer.flush(timeout=5.0)‘ ensures we don’t block HTTP clients if Kafka is down. In Spark, ‘spark.network.timeout‘ acts as a global safety valve to kill stuck tasks.

3. **Circuit Breaker ("Lite" Implementation):** While we didn’t use a library like Hystrix, we implemented the pattern’s core concept at the Gateway. If the producer buffer fills up (due to downstream Kafka failure), the Gateway immediately returns 503 errors to the client rather than accepting more data into an unbounded queue. This "failing fast" protects the gateway from OOM crashes during outages.

Key Takeaways

Distributed systems will fail and when such time comes, the goal is not to prevent failure but to contain it. Timeouts and Backpressure (failing fast) are not any less important than Service Discovery and Uptime in such systems.

3.6 Lesson 6: Performance Optimization

Problem Description

During the stress testing phase (simulating 3600x speedup), we identified two critical bottlenecks. First, the Ingestion Gateway became CPU-bound due to the overhead of handling thousands of individual HTTP requests per second. Second, our initial Docker images were bloated (>1GB), leading to slow pod startup times in the Kubernetes cluster and excessive bandwidth usage during local development updates.

Approaches Tried

- **Approach 1: Naive Implementation.** The gateway ran on a blocking WSGI server, and the Dockerfiles were written sequentially without multi-stage builds. This resulted in significant I/O wait times and large image layers that effectively cached "dead weight" (build tools).

- **Approach 2: Vertical Scaling.** We attempted to simply increase the CPU limits for the gateway container. While this delayed the saturation point, it did not solve the fundamental inefficiency of the request-response cycle.

Final Solution

We applied targeted optimizations across the stack:

1. **Gateway & Producer Tuning:** We switched to an Asynchronous Server Gateway Interface (ASGI) using **FastAPI** and **Uvicorn**. Crucially, we tuned the underlying Kafka Producer to use `linger.ms=10` and `queue.buffering.max.messages=100000`. This allows the producer to buffer records in memory for 10ms to create larger batches, drastically reducing network syscalls for a negligible latency trade-off.
2. **Docker Multi-Stage Builds:** We refactored our Python Dockerfiles to use the **Builder Pattern**. By using a temporary `builder` stage to compile dependencies with `uv` and then copying only the virtual environment (‘.venv‘) to a `distroless` or `slim` runtime image, we reduced the Gateway image size by over 60%.
3. **Layer Squashing:** In the Spark image, we combined file operations (`mkdir`, `chmod`, `curl`) into single `RUN` instructions. This prevents the creation of intermediate layers for temporary files, keeping the image lightweight.

Key Takeaways

Performance is often about I/O efficiency, not just raw code speed. Optimizing the "borders" between systems (HTTP Batching, Kafka Producer Batching) yields higher returns than optimizing internal logic. Similarly, optimizing artifacts (Docker images) pays off in deployment velocity and cluster stability.

3.7 Lesson 7: Monitoring & Debugging

Problem Description

Initially, we had "blind spots" where the pipeline would silently fail (e.g., zero records written to Cassandra) but the Spark job would appear "Running." We had no visibility into whether data was actually flowing. Another instance would be a setting in Spark that concludes the script once a table is done aggregated instead of waiting for all necessary aggregations. While the script ran successfully, the outcome is unwanted and unpredictable.

Approaches Tried

- **Approach 1: Log Grepping.** ssh-ing into containers to read `stdout` logs. This was tedious and unscalable.
- **Approach 2: Spark UI.** Using the Spark UI (port 4040) gave job details but didn’t show end-to-end business metrics.

- **Approach 3: Automated Test.** We wrote concise CI/CD tests targetting fragile components using GitHub Action. While tedious, it saved lots of time checking which commit break the pipeline.

Final Solution

Since our development pace is rather fast, we adopted automated test early but had to dropped midway due to the lack of time to write such scripts. Therefore we integrated **Grafana** directly with our serving layer (Cassandra) in tandem. By visualizing the output data (e.g., "Trips Ingested per Minute"), we created a feedback loop. If the graph drops to zero, we instantly know upstream processing has failed.

Key Takeaways

It's all about the balance. Writing automated tests saves time if you work with large enough scale of a project in large enough time. If the time writing those tests diminishes the saved time and mind spaces, sometimes it's much better to just log the end result out and enable `-verbose` flag.

3.8 Lesson 8: Data Quality & Testing

Problem Description

The taxi dataset contains noise—trips with negative distances, zero passengers, or impossible timestamps. Ingesting this polluted our KPIs (e.g., infinite average speed calculations), which is even more important in streaming pipeline where the amount of data for each aggregation is not large and outliers definitely stands out.

Approaches Tried

- **Approach 1: Spark Cleanup.** We used Spark for what it's supposed to do: Cleaning up the data. Although Spark is fast, multiple filters and transformations on multiple features would drag down the performance, which is noticeable on perpetually running scripts processing real-time data.
- **Approach 2: Upstream Cleanup.** We handled some filtering right after ingesting data from the API, right before they are delegated to data broker. This added more complexity to the components but alleviate some weight for downstream processes.

Final Solution

We implemented a hybrid of the 2 approaches **for our simulation**. We explicitly filter out invalid records (`trip_distance > 0`, `total_amount > 0`) immediately after parsing from API, and filter further with Spark. This ensures that only high-quality, valid data enters our aggregations and storage. But we fully awared that this introduced more breaking points and

apparent incoherence between data transporation (e.g. upstream filtering removing something downstream processes expected).

Key Takeaways

Depending on the data characteristic and your business needs, you can integrate data filtering in multiple stages to enhance robustness and optimize performance.

3.9 Lesson 9: Fault Tolerance

Problem Description

Streaming jobs are long-running processes that will inevitably encounter failure (network blip, pod restart). When our Spark driver crashed, it would restart and re-process the entire Kafka topic from the beginning, creating massive duplicates in Cassandra.

Approaches Tried

- **Approach 1: Manual Offset Management.** Trying to track offsets manually was error-prone.
- **Approach 2: Transient State.** Running without checkpoints meant losing state on restart.

Final Solution

We relied on **Checkpointing**. By configuring `checkpointLocation`, Spark saves the streaming state and Kafka offsets to a persistent volume (`spark-checkpoints-pvc`). On restart, it picks up exactly where it left off, ensuring exactly-once semantics.

Key Takeaways

State must be persistent. In a stateful streaming application, the checkpoint directory is as important as the database itself. It is the only bridge across the chasm of failure.

3.10 Lesson 10: HDFS Network Identity in Containers

Problem Description

During the integration of the Batch Layer, Spark successfully connected to the NameNode but failed to write data blocks to DataNodes, throwing `java.net.ConnectException`. The DataNodes were advertising their internal container IPs (e.g., `172.18.0.x`) to the NameNode. Spark, running in a separate container, could not route traffic to these dynamic internal IPs properly due to Docker network isolation nuances.

Approaches Tried

- **Approach 1: Hosts File Manipulation.** We attempted to manually map IP addresses in the `/etc/hosts` file of the Spark container, but this was unsustainable as IPs changed on every restart.
- **Approach 2: Port Forwarding.** We tried exposing DataNode ports to the host machine, but this broke the internal communication within the Docker network.

Final Solution

We enforced **Hostname-based Advertisement**. We configured the DataNode environment variables (`HDFS_CONF_dfs_datanode_use_datanode_hostname=true`) to force it to register with the NameNode using its Docker service name (`datanode`) instead of an IP address. Since all containers share the same Docker network user-defined bridge, they can resolve these hostnames reliably.

Key Takeaways

In containerized distributed systems, internal IP addresses are ephemeral and unreliable. Always configure services to advertise themselves using stable DNS hostnames (Service Discovery) rather than raw IP addresses.

3.11 Lesson 11: Distributed Storage Metadata Synchronization

Problem Description

After restarting the cluster using `docker compose down` followed by `up`, the DataNode container would immediately exit. Logs revealed a `IncompatibleClusterIDException`. The NameNode had been reformatted (generating a new Cluster ID), but the DataNode's persistent volume still retained metadata from the previous cluster, causing a "Split-Brain" scenario where the worker rejected the master.

Approaches Tried

- **Approach 1: Manual Deletion.** We manually exec-ed into the container to delete the `/hadoop/dfs/data` directory, but this was tedious and manual.
- **Approach 2: Soft Restarts.** We tried restarting only the services without bringing down the volumes, but this led to inconsistent states.

Final Solution

We implemented a **Hard Reset Strategy** for infrastructure changes. We utilized `docker compose down -v` to destroy all volumes, ensuring that both NameNode and DataNode start with a clean slate. For production scenarios, we learned that the `VERSION` file in DataNode volumes must be synchronized with the NameNode's namespace ID.

Key Takeaways

Distributed storage systems maintain strict metadata consistency between Master and Worker nodes. When using ephemeral environments (like Docker Compose), partial persistence is dangerous; either persist everything permanently or reset everything completely.

3.12 Lesson 12: Service Orchestration and Race Conditions

Problem Description

Even when all containers were "Healthy", Spark jobs failed immediately upon startup with `AnalysisException` (Cassandra table not found) or `SafeModeException` (HDFS read-only). The `depends_on` directive in Docker Compose only waits for the container to start, not for the application (Java/DB) to be ready to accept connections or write operations.

Approaches Tried

- **Approach 1: Restart Policy.** We relied on `restart: on-failure` for Spark, hoping it would eventually work after several crashes. This created noisy logs and unpredictable startup times.
- **Approach 2: High Sleep Times.** We added `sleep 60` commands, which slowed down development and didn't guarantee readiness on slower machines.

Final Solution

We developed a **Sequential Initialization Script** (`batch-processing.sh`). This script automates the operational lifecycle: it creates infrastructure -> actively polls for health (using `cqlsh` and `hdfs dfsadmin`) -> executes setup commands (Schema Load, `safemode leave`, `chmod 777`) -> and only then launches the Spark application.

Key Takeaways

Infrastructure readiness is not binary. Distributed systems have a "warm-up" phase (Safe Mode, Gossip Protocol). Orchestration scripts must actively verify application-level readiness (Health Checks) before launching dependent consumers.

3.13 Lesson 13: Configuration Drift in Pre-built Images

Problem Description

Spark repeatedly failed with `Connection Refused` when trying to connect to `hdfs://namenode:9000`. We assumed the standard Hadoop RPC port was 9000 based on general documentation. However, deeper investigation into the specific Docker image logs revealed the service was actually binding to port 8020, ignoring our environment variable overrides for the RPC address.

Approaches Tried

- **Approach 1: Force Bind.** We tried forcing the NameNode to bind to 9000 via environment variables, but the image's entrypoint script prioritized internal defaults.
- **Approach 2: Port Mapping.** We mapped host port 9000 to container port 8020, but this didn't affect internal container-to-container communication.

Final Solution

We adopted an **Adaptation Strategy**. Instead of fighting the infrastructure's defaults, we updated the Spark configuration to match the infrastructure (`HDFS_NAMENODE=hdfs://namenode:8020`). We also synchronized the `fs.defaultFS` core configuration across all nodes to ensure the entire cluster agreed on the port.

Key Takeaways

"Defaults" vary by distribution. Always verify the actual listening ports inside the container (using `netstat` or logs) rather than relying on external documentation. Consistency across client (Spark) and server (Hadoop) configuration is paramount.

3.14 Lesson 14: Connector-Specific Query Semantics and Macro Reliability

Problem Description

When provisioning Grafana dashboards against Cassandra using the HadesArchitect plugin, queries that previously worked intermittently began to fail after restarting. Two distinct symptoms appeared: (1) panels failed with parser errors or returned no rows when the dashboard used Grafana time macros such as `$__timeFrom()` / `$__timeTo()`, and (2) time-series panels displayed dozens of tiny series (one per timestamp) instead of grouped series by zone because the plugin interpreted the first selected column as the series identifier. In short, the datasource plugin had strict, non-obvious expectations about query form (macro usage and column order) that we had not followed.

Approaches Tried

- **Macro variants.** We experimented with different macro forms (`$__timeFrom()`, `$__timeFrom`, and explicit numeric timestamps) inside raw CQL to force the plugin to accept time ranges.
- **Structured vs raw queries.** We tried both the plugin's structured query mode (setting `keyspace`, `table`, `columnTime`, `columnValue`) and raw CQL mode to see which path avoided parsing failures.

- **Column re-ordering and aliasing.** We rewrote SELECT statements, aliasing the timestamp as `time` and moving different columns to the front to test how the plugin chooses the series key.
- **Pre-aggregation / multiple targets.** To get multi-series displays without cross-partition scans, we tested separate targets per zone and also created small pre-aggregated snapshot tables (top-N) written by Spark.

Final Solution

We adopted a **compatibility-first** approach that obeys the plugin's documented contract:

1. Stop using the problematic macro form inside raw CQL. Instead either rely on the plugin's time fields (structured mode) or use raw CQL without `$__timeFrom()/$__timeTo()` macros and let Grafana expand time range in the request body. In our prototype we removed `$__timeFrom()` from WHERE clauses entirely and used explicit partition-key filters plus ordered queries.
2. Ensure the SELECT column order matches the plugin expectation: **identifier first**, `window_start AS time` second, and numeric value(s) last. Example pattern used successfully: `SELECT pickup_zone, window_start AS time, total_revenue AS value FROM ...`
3. For multi-series (top zones) avoid cross-partition aggregation at query time. We implemented either (a) multiple targets (one per zone) or (b) a Spark-written `top_zones` table that materializes the top-N per window and is cheap to query from Grafana.
4. Always include required partition-key predicates (e.g., `shard_id='global'` for global KPIs) so the plugin will execute the query rather than skipping it for missing fields.

Key Takeaways

- **Read the plugin contract.** Third-party datasource plugins may enforce query shapes that differ from generic SQL/CQL expectations (column ordering, mandatory keys). Follow the plugin's Query Editor guidance exactly.
- **Design for Cassandra constraints.** Avoid ad-hoc cross-partition aggregations in Grafana; prefer pre-aggregated tables for top-N and other global views, or use multiple per-partition targets when necessary.
- **Document and pin query patterns.** Keep a short reference in the project repo with canonical SELECT patterns (ID first, `time` alias, value last) and required partition-key filters so future dashboard edits remain compatible with the plugin.

∎

A NYC Taxi Dataset Schema and Spatial Resources

This appendix provides a structured overview of the Yellow NYC Taxi dataset schema and the auxiliary spatial resources used in this project. The dataset is released by the New York City Taxi and Limousine Commission (TLC) and follows a standardized format across monthly partitions.

A.1 Trip Record Schema

Each record in the Yellow Taxi dataset represents a completed taxi trip and includes temporal, spatial, operational, and financial attributes. The fields most relevant to this project are summarized in Table 1.

Table 1: Taxi Data Dictionary

Field Name	Description
VendorID	A code indicating the TPEP provider that provided the record. 1 = Creative Mobile Technologies, LLC 2 = Curb Mobility, LLC 6 = Myle Technologies Inc 7 = Helix
tpep_pickup_datetime	The date and time when the meter was engaged.
tpep_dropoff_datetime	The date and time when the meter was disengaged.
passenger_count	The number of passengers in the vehicle.
trip_distance	The elapsed trip distance in miles reported by the taximeter.
RatecodeID	The final rate code in effect at the end of the trip. 1 = Standard rate 2 = JFK 3 = Newark 4 = Nassau or Westchester 5 = Negotiated fare 6 = Group ride 99 = Null/unknown

Field Name	Description
store_and_fwd_flag	This flag indicates whether the trip record was held in vehicle memory before sending to the vendor, aka "store and forward," because the vehicle did not have a connection to the server. Y = store and forward trip N = not a store and forward trip
PULocationID	TLC Taxi Zone in which the taximeter was engaged.
DOLocationID	TLC Taxi Zone in which the taximeter was disengaged.
payment_type	A numeric code signifying how the passenger paid for the trip. 0 = Flex Fare trip 1 = Credit card 2 = Cash 3 = No charge 4 = Dispute 5 = Unknown 6 = Voided trip
fare_amount	The time-and-distance fare calculated by the meter. For additional information on the following columns, see `https://www.nyc.gov/site/tlc/passengers/taxi-fare.page`
extra	Miscellaneous extras and surcharges.
mta_tax	Tax that is automatically triggered based on the metered rate in use.
tip_amount	Tip amount – This field is automatically populated for credit card tips. Cash tips are not included.
tolls_amount	Total amount of all tolls paid in trip.
improvement_surcharge	Improvement surcharge assessed trips at the flag drop. The improvement surcharge began being levied in 2015.
total_amount	The total amount charged to passengers. Does not include cash tips.
congestion_surcharge	Total amount collected in trip for NYS congestion surcharge.

Field Name	Description
airport_fee	For pick up only at LaGuardia and John F. Kennedy Airports.
cbd_congestion_fee	Per-trip charge for MTA's Congestion Relief Zone starting Jan. 5, 2025.

Additional fields such as surcharges, tolls, congestion fees, and rate codes are available in the dataset but are not central to the analytics presented in this project. A full and authoritative data dictionary is maintained by the NYC TLC and accompanies each dataset release.

A.2 Taxi Zone Lookup and Spatial Data

Spatial information in the dataset is encoded using TLC Taxi Zone identifiers rather than raw geographic coordinates. To support geographic aggregation and visualization, the TLC provides a Taxi Zone Lookup Table that maps each zone identifier to a corresponding zone name and borough.

In addition, official geographic boundary data for taxi zones is publicly released in the form of a Taxi Zone Shapefile, distributed in Parquet format. This file defines the polygon boundaries for all taxi zones across the five boroughs of New York City: the Bronx, Brooklyn, Manhattan, Queens, and Staten Island. Borough-level taxi zone maps are also provided in image format, serving as visual references for spatial analysis and dashboard design.

These auxiliary spatial resources enable zone-level aggregation, borough comparisons, and geographic visualization without requiring access to precise GPS coordinates, thereby simplifying spatial processing while preserving analytical value.

A.3 Data Availability and Governance

The Yellow Taxi dataset is released as monthly partitions and covers multiple years of historical data. Files are publicly accessible through official TLC and NYC Open Data distribution channels. Dataset sizes vary by month, with individual files typically ranging from hundreds of megabytes to several gigabytes, resulting in multi-terabyte scale when aggregated over long time periods.

Data collection and publication are governed by regulatory requirements imposed by the TLC. Licensed taxi operators are responsible for generating trip records, approved technology vendors handle data transmission, and the TLC oversees validation, maintenance, and public release. This centralized governance model ensures consistency and data quality, while also enabling open access for research and system development.