

Scalable Data Infrastructure for Real-Time and Batch Analytics

CS744: Big Data Systems

Team DataBloom

Gokulnath Sourirajan, Thilak Raj Murugan, Harsha Varthan Chellaiah Ramamoorthy

1 Abstract

This report presents the design and implementation of a scalable data infrastructure to handle both batch and streaming workloads using the NYC Taxi trip dataset. Leveraging the Lambda architecture, we integrate Apache Kafka, Spark, and Google BigQuery to support real-time and historical data processing. Our system design supports low-latency querying, large-scale batch analytics, and orchestrated data workflows using Apache Airflow.

We chose the NYC Taxi dataset due to its realistic structure, high volume, and relevance to urban mobility analytics. The dataset spans over a decade, covering millions of trips across different taxi types. Our implementation focuses on practical challenges like schema drift, stream simulation from static sources, resource-efficient Spark jobs. Through this project, we explore the trade-offs and design principles that influence scalable data infrastructure, from storage formats to compute engines and message brokers. We also evaluate the system's performance on key metrics such as query latency, throughput, and resource utilization. Our findings demonstrate the feasibility of deploying hybrid data architectures using open-source and cloud-native components.

2 Introduction

In the era of big data, organizations generate vast volumes of structured and unstructured

information that must be processed, analyzed, and acted upon in near real-time. From finance and healthcare to logistics and entertainment, the ability to extract insights from data in a scalable and timely manner has become a critical business differentiator. Traditional data systems designed for batch workloads are often insufficient to meet the latency and responsiveness demands of modern applications involving both batch and real time processing.

To bridge this gap, the Lambda architecture has emerged as a design pattern that combines the accuracy of batch processing with the low-latency responsiveness of stream processing. This dual-pipeline model enables applications to access both historical trends and up-to-the-minute insights. In this project, we apply this model to build a scalable analytics platform using the NYC Taxi dataset—a publicly available dataset that reflects the complexities of real-world urban data.

We use technologies such as Apache Kafka for stream ingestion, Apache Spark for both batch and streaming transformations, Google BigQuery for analytical storage and querying. Speed Layer handles real-time data ingestion using Kafka and processes them using stream processing technologies. Low latency is key here and data streams are typically stored temporarily in fast access systems like Kafka. Batch Layer processes huge volumes of data in batches, storing results in a data lake. The Serving Layer retrieves data from this data lake,

allowing it to serve historical data for queries.

Our goals include designing and deploying a pipeline that can scale from gigabytes to terabytes of input data, simulate real-time data flow from static archives, handle schema inconsistencies, and support analytical queries with minimal latency. This report documents the design rationale, implementation details, challenges encountered, and experimental results of our system.

3 System Design and Implementation

3.1 Apache Kafka

Apache Kafka is a distributed, fault-tolerant messaging system designed for high-throughput, low-latency data streaming. At its core, Kafka organizes data into *topics*, each divided into ordered *partitions* that are replicated across multiple *brokers* to ensure durability and availability. Producers publish messages to a chosen topic partition, while consumers subscribe to one or more topics and maintain their own offsets to track progress.

Kafka’s design emphasizes three key properties:

1. **Scalability:** By partitioning topics, Kafka can distribute load across brokers and parallelize both reads and writes.
2. **Durability:** Written data is stored on disk and replicated, providing persistence even in the event of broker failures.
3. **Performance:** Kafka achieves high throughput by employing sequential disk I/O and batching, and by decoupling producers and consumers with a pull-based model.

Kafka’s fault-tolerant and high-throughput nature makes it a perfect fit for this our case, as it enables us to reliably transmit large volumes of data while ensuring low-latency updates to real-time metrics. It plays a critical role in the streaming layer, where it is used to handle the real-time flow of preprocessed taxi trip records.

Additionally, by decoupling the data producers (GCS) from the consumers (Spark Streaming), Kafka allows us to scale the system independently in the future, whether by adding more sources or expanding the processing capabilities.

After the raw data is cleaned and preprocessed in GCS, the records are converted into JSON format and published to a Kafka topic. This enables our system to handle incoming trip data incrementally, where Kafka serves as the buffer that allows us to stream data to downstream consumers, such as the Spark Streaming application, for real-time processing.

3.2 Lambda Architecture

Our system follows the Lambda architecture, a data-processing pattern that integrates both batch and real-time processing to enable scalable, fault-tolerant analytics on large datasets. This model divides the data pipeline into three layers—batch, speed, and serving—each tailored for specific latency and accuracy requirements. We adopted this architecture to address the practical need for combining long-term historical analysis with low-latency insights, which are increasingly expected in modern data platforms.

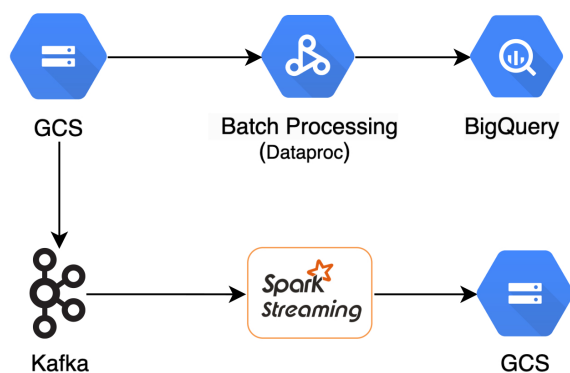


Figure 1: Lambda architecture implementation for scalable data infrastructure

For our implementation, we used the publicly available NYC Taxi and Limousine Commission dataset, which includes trip-level records such

as pickup/dropoff times, locations, fares, and passenger counts. The raw data, spanning ten years, was stored in Parquet format on Google Cloud Storage (GCS), serving as the foundational input to both the batch and streaming components of our system. The dataset includes three main categories: Yellow Taxi trips, Green Taxi trips, and High Volume For-Hire Vehicle (FHV) records. The high-volume category includes ride-hailing services such as Uber, Lyft, Via, and Juno offering a comprehensive view of both traditional and app-based transportation across New York City. The architecture of this system, as depicted in Figure 1, illustrates the integration of batch and streaming layers, showcasing how we process and analyze this data to generate both historical and real-time insights. This design enables independent scaling of storage and compute, allowing the system to efficiently adapt to varying data volumes and processing workloads.

3.3 Data Preprocessing

Before data could be processed through the batch or streaming layers, we performed a dedicated preprocessing step to ensure consistency and quality. Raw Parquet files stored in Google Cloud Storage were first ingested and parsed to standardize schema and enforce data integrity. We explicitly cast each column to its appropriate datatype, such as timestamps, numerical values, and categorical fields. Additionally, we filtered out records containing invalid or missing vendor IDs to eliminate corrupted entries. Since each file corresponded to a specific month of a given year, we also discarded any records with timestamps that did not align with the file’s expected temporal range. The cleaned and preprocessed data was then written back to GCS, forming a consistent and reliable base for both batch and stream processing.

3.4 Batch Layer

The Batch layer operates on fixed, bounded datasets and is designed to process data in periodic intervals rather than continuously. While

it provides a robust and scalable foundation for large-scale analytics, it inherently introduces latency, as the results reflect only the data available up to the most recently completed batch. Consequently, it cannot capture the most recent updates until the next batch cycle is executed. In our implementation, this layer consumes preprocessed Parquet files stored in Google Cloud Storage (GCS), with each batch corresponding to one year of data from multiple NYC Taxi services—yellow, green, and high-volume for-hire vehicles. Spark jobs on Google Cloud Dataproc (a managed Spark cluster provided by Google Cloud Platform) are used to merge these records and generate two outputs. The first is a unified table of all trip records appended to BigQuery, enabling comprehensive historical analysis. The second is a set of monthly and yearly aggregates, such as ride counts, average fares, and vendor-wise breakdowns, written to a reporting table in BigQuery. Previous reports are archived before being replaced, maintaining a consistent historical snapshot. This setup provides a reliable but eventually consistent view of the dataset suitable for long-term analytical workloads.

3.5 Speed Layer

The Speed layer is designed to handle real-time or near-real-time data processing, complementing the batch layer by providing timely updates based on the most recent records. It processes data in small batches or continuous streams to compute metrics with minimal latency, offering a more current—though potentially less complete—view of the system. In our implementation, preprocessed NYC Taxi records stored in Google Cloud Storage are serialized into JSON format and emitted as streaming messages into an Apache Kafka topic hosted on CloudLab. A Spark Streaming consumer subscribes to this topic and incrementally aggregates key metrics such as trip counts, average fare values, and service-specific activity over tumbling and sliding windows. The resulting summaries are written back to GCS, where they can be immediately accessed by downstream visualization tools and monitoring systems. This enabled us

to simulate a real-time processing environment while maintaining control over input volume and timing, allowing for performance evaluation under constrained yet realistic conditions.

3.6 Serving Layer

The Serving layer merges the outputs of the Batch and Speed layers to offer a unified query interface. It typically maintains and exposes precomputed views, allowing clients to choose between comprehensive historical analytics and the freshest real-time data. While our work centered on the Batch and Speed implementations, this layer conceptually can be thought of as an extension of BigQuery’s querying capabilities—providing the same efficient access to historical data while also incorporating the freshest streaming results. In essence, BigQuery’s performance in handling large-scale, ad hoc queries illustrates how a true Serving layer would unify both data horizons into a single, low-latency interface.

4 Evaluation

To evaluate the scalability and performance of our data platform, we conducted experiments using the NYC Taxi dataset with two distinct processing modes: batch processing using Apache Spark, and stream processing using Kafka and Spark Streaming.

4.1 Experimental Setup

For batch processing, the Dataproc cluster is configured to run as a single-node instance with 4 vCPUs and 16GB of memory. It uses YARN as the resource manager to handle resource allocation and job scheduling within the cluster.

In case of streaming, Spark and Kafka are deployed as Docker containers on a Cloudlab host equipped with 48 logical CPU cores and 512 GiB of RAM. The Spark setup consists of one Spark master and two Spark workers, enabling parallel and distributed stream processing. The Kafka stack uses Confluent’s official docker images[1] and includes ZooKeeper for

coordination, the Kafka broker for messaging, Schema Registry for Avro schema management, Kafka REST Proxy for HTTP-based access, and Control Center for cluster monitoring and management. Together, these components form a complete and scalable streaming pipeline.

4.2 Batch Processing

In the Figure 2, as the input data size for batch processing increases from 6 GiB to 41.34 GiB, the processing time correspondingly rises from 6 minutes to 28 minutes. BigQuery demonstrates excellent scalability and low latency, scanning 80 GiB of data in just 6 seconds. The final output, stored in BigQuery, amounts to 114 GiB and contains over one billion records. This shows that BigQuery is more efficient at running analytic queries over large dataset (75GB+) compared to Spark. This is because BigQuery runs queries in a parallel manner, leveraging thousands of nodes in Google’s infrastructure to process data chunks concurrently. In addition to parallelism, it uses columnar storage, allowing it to scan only the necessary columns instead of full rows. At the core of its execution is the Dremel engine, a scalable, tree-based architecture that enables fast aggregation and efficient hierarchical querying.

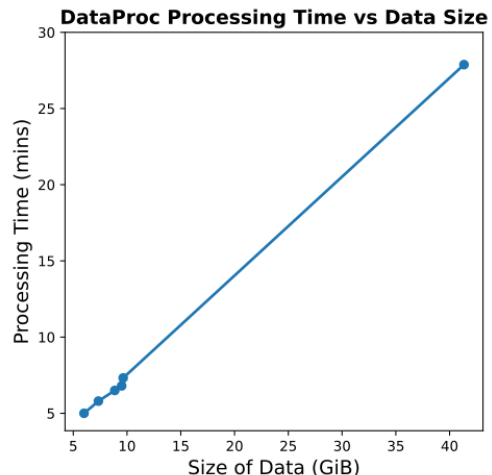


Figure 2: Processing Time vs Data Size

4.2.1 Disk I/O & CPU Utilization

During the job execution, disk I/O plot 3 shows an initial read throughput spike to 384 KiB/s due to the ingestion of the partitioned 41.34 GiB dataset. As the job progresses, the write throughput surpasses the read throughput, reflecting the ongoing output writes to BigQuery.

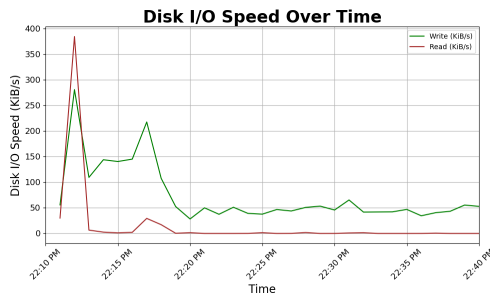


Figure 3: Disk I/O Speed over Time

As shown in Figure 4, CPU utilization also peaks at the start, reaching 77.5% as the Spark environment initializes, and maintains an average utilization of around 70% throughout the job.

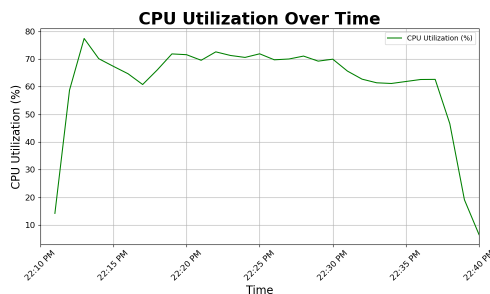


Figure 4: CPU utilization over Time

4.3 Streaming

In case of streaming, the consumer subscribes to a specific Kafka topic and is ready to receive the messages. The producer publishes each trip record as a message to the Kafka topic. As the messages start populating the topic, the consumer starts receiving the trip records. The consumer runs analytical queries to calculate metrics like total trips and revenue aggregated over a fixed 5 minute window based on the

“pickup” timestamp value. We also use a 5 minute watermark window based on the pickup timestamp to account for late trip records.

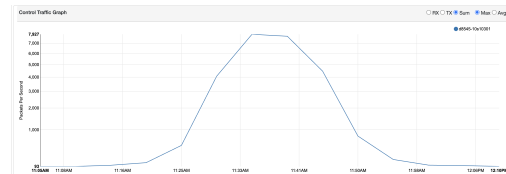


Figure 5: Network Traffic Over Time

During the execution of the streaming pipeline, the network experienced a significant increase in traffic, peaking at 7,927 packets per second (Figure 5). This is expected as there is a lot of communication involved between the Kafka container and the Spark containers running the producer and consumer client code.

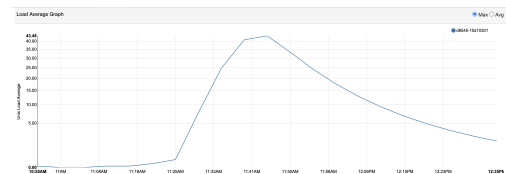


Figure 6: CPU utilization over Time

We also observe that the CPU utilization increases as the streaming pipeline is run and has a peak utilization of 90.58% (Figure 6). In this context, the load average indicates the number of CPU cores actively used out of the 48 logical cores available on the machine.

5 Conclusion

In this project, we set out to validate a scalable data infrastructure capable of both high-throughput batch processing and low-latency stream analytics—an essential requirement for modern data-driven organizations. While the NYC Taxi dataset provided the workload, our primary aim was to explore system trade-offs, resource utilization, and performance characteristics across different processing paradigms.

Our implementation followed a simplified Lambda-style pipeline that supports both batch

and stream processing. For batch operations, raw Parquet files stored in Google Cloud Storage were preprocessed and recomputed on Google Cloud Dataproc, with final aggregates persisted in BigQuery. In parallel, for streaming, preprocessed records were emitted as JSON into a Kafka topic on CloudLab, where Spark Streaming consumed and aggregated them to produce near-real-time metrics. This dual-path design demonstrated how a unified infrastructure can deliver both comprehensive historical analytics and low-latency real-time insights while accommodating independent and self-hosted components.

We observed that BigQuery’s Dremel engine leveraged columnar storage and massive parallelism to scan over 80 GiB of data in under 10 seconds, demonstrating exceptional scalability for analytical queries. In contrast, Spark’s batch jobs scaled linearly—processing times increased from 6 minutes for 6 GiB to 28 minutes for 41 GiB—underscoring its predictability for large-batch workloads. Additionally, running Kafka outside of a managed environment illustrated the complexity and overhead involved, suggesting that production systems may benefit from managed or serverless streaming services.

Looking forward, we aim to extend this pipeline with orchestration tools for better automation and scheduling. Additionally, integrating a Lakehouse architecture on top of cloud storage could unify batch and streaming data, providing ACID guarantees, schema evolution, and time-travel capabilities. Finally, expanding our benchmarks to include end-to-end latency, fault-tolerance under load, and cost-per-query analyses will bring us closer to production-grade standards.

6 Related Work

There has been a lot of work on building scalable OLAP and OLTP systems at Uber. Uber’s real-time data infrastructure has been extensively discussed in their work on distributed streaming architectures, emphasizing scalability, fault tolerance, and low-latency process-

ing for mission-critical applications [2]. Their system leverages Kafka for message queueing, Spark for batch processing, and Flink for real-time streaming. Apache Pinot [3], a real-time OLAP datastore, has been a key component in the provision of analytical queries at scale, enabling low-latency query execution over high-ingestion-rate datasets [2, 3]. The integration of Pinot into Uber’s architecture has facilitated the execution of millions of queries per second, demonstrating its effectiveness in supporting real-time decision-making and analytical workloads [4]. To efficiently store and manage large volumes of streaming data, Uber has also used Kafka Tiered Storage, allowing cost-efficient data retention and retrieval while ensuring system scalability [5, 6]. For transactional workloads, Uber has explored the use of Google Spanner to enhance OLTP performance, particularly on fulfillment and real-time decisioning platforms. By transitioning to a globally consistent and highly available transactional system, Uber has improved service reliability and query performance for mission-critical operations [7, 8, 9]. There has also been a recent trend in data lake house architectures to process PB scale data at Apple [10].

7 Contribution

- Thilak Raj Murugan : I proposed the project idea by drawing inspiration from the book *Designing Data-Intensive Applications* and several technical blogs detailing data infrastructure at companies like Uber, Netflix and Apple, which are also summarized in the related work section. Wrote the related work and evaluation sections for both batch and streaming components, and was responsible for generating graphs and designing the architecture diagram. My technical contributions included preprocessing the data, fixing bugs in the codebase, and setting up Dataproc clusters for batch processing and running Spark jobs. For the streaming pipeline, I configured and deployed Kafka and Spark using Docker containers on Cloudlab.

- Gokulnath Sourirajan: Contributed to the literature review and participated in the overall system design. Developed a shell script to automate batch processing by simulating yearly data ingestion from Parquet files. Played a key role in setting up and experimenting with the stream processing pipeline, including deployment and validation of the Kafka and Spark Streaming components.
- Harsha Varthan Chellaiah Ramamoorthy: Read literature and studied systems for batch and real-time processing, Lambda Architecture, and experimental design. Implemented data preprocessing and analytical queries for yellow and green taxi datasets. Also contributed to high volume data preparation, designed the project poster, and helped write the report.
- Common: In addition to the above tasks, we actively collaborated as a team on the poster presentation, final report, and throughout the development of the project.

- [7] Uber's Fulfillment Platform. <https://www.uber.com/blog/building-ubers-fulfillment-platform/>.
- [8] Uber Fulfillment Platform re-architecture. <https://www.uber.com/blog/fulfillment-platform-rearchitecture/>.
- [9] Google Spanner usecase at Uber. <https://tinyurl.com/4mh4v5hn>.
- [10] Petabyte Scale Row-Level Operations in Data Lakehouses. <https://doi.org/10.14778/3685800.3685834>.

References

- [1] Official Kafka Docker images provided by Confluent. <https://hub.docker.com/u/confluentinc>.
- [2] Uber Real-Time Data Infrastructure. <https://arxiv.org/pdf/2104.00087>.
- [3] Operating Apache Pinot at Uber. <https://www.uber.com/blog/operating-apache-pinot/>.
- [4] Serving Millions of Queries Using Pinot. <https://www.uber.com/blog/serving-millions-of-apache-pinot-queries-with-neutrino/>.
- [5] Kafka Tiered Storage. <https://www.uber.com/blog/kafka-tiered-storage/>.
- [6] Kafka Tiered Storage Implementation. <https://tinyurl.com/vjh44red>.