DAT535 Project Report

Spark Medallion architecture pipeline

Mattis Sørensen
Preben Solsvik Paulsen
University of Stavanger, Norway
matt.sorensen@stud.uis.no
p.paulsen@stud.uis.no

ABSTRACT

This project utilizes Apache Spark to create a pipeline for the processing of TLC Trip Record data. It follows the Medallion architecture, producing a bronze, silver, and gold layer consisting of data ingestion, data cleaning, and data analyzing, respectively. The system is designed to handle large-scale datasets while ensuring data quality, reliability, and reproducibility throughout the workflow. Finally, the processed gold layer data is used as input to train machine learning models for predictive analysis. The resulting predictive models offer insight into travel patterns and outcomes, demonstrating the effectiveness of a structured end-to-end architecture for transportation analytics.

1 INTRODUCTION

Urban transportation systems have led to the generation of massive amounts of trip related data. This is especially true in large cities like New York City, where TLC trip records are collected at an enormous scale. In the increasingly data driven world of today, organizations such as these rely on high quality, large scale datasets to improve efficiency and support informed decision making. Such data driven insights benefit not only the organizations, but also the city and its residents, as they are used to reduce congestion, smarter traffic management, and lower emissions.

To better tailor the dataset to the project goals, only a subset of the original features of the original TLC trip record was used. This decision was motivated by several factors. First, the dataset had to be scrambled and intentionally unstructured to simulate realistic ingestion scenarios and to emphasize the value of the medallion architecture in restoring order and quality. Second, many columns in the raw dataset were irrelevant to analytical objectives, redundant, or offered little informational value, and were therefore excluded. By selecting only the essential features, the project focused on meaningful transformations while reducing unnecessary processing overhead.

The purpose of the medallion architecture is to logically organize the data with the goal of incrementally and progressively improving the structure and quality of the data as it flows through each layer. [1] The architecture consists of three main layers. The Bronze layer, which stores raw ingested data, does basic validity checks and meta tags, the Silver layer, where the Bronze data is cleaned and processed, and the Gold layer where data from the Silver layer is used to create refined and enriched data prepared for the specific needs of the project.

Apache Spark is a distributed data processing engine designed for large-scale analytics and cluster computing. It provides an efficient in-memory computation model, allowing it to outperform traditional MapReduce systems for iterative workloads, streaming applications, and machine-learning pipelines. Spark offers high level APIs in multiple languages, but in this project, PySpark was used. Spark uses a Directed Acyclic Graph DAG based execution engine which organizes tasks in a way that ensures execution in the correct order without any cycles. These characteristics make Spark a suitable choice for building scalable pipelines and analytics workflows in modern data intensive environments [6].

The code and scripts used in this project can be found here:

https://github.com/dlqnt/DAT535

1.1 Use Case

Although large scale transportation datasets such as the NYC TLC Trip Records contain valuable information, turning this raw data into operational insights remains a non-trivial challenge due to its size, irregularity, and structure. The core research problem addressed in this project is how to design and implement a scalable data processing pipeline using the medallion architecture and Apache Spark with the end goal of producing analytics-ready data capable of supporting both descriptive and predictive use cases of transportation. This involves not only organizing and cleaning the data across Bronze, Silver, and Gold layers, but also demonstrating how the resulting Gold layer dataset can be leveraged to answer concrete business questions, such as identifying hourly demand patterns, understanding payment-type behavior, and analyzing trip-distance distributions.

We also investigate how machine learning models trained on this data perform in demand forecasting, fare prediction, payment type classification, and trip pattern clustering. The goal of these models is to provide meaningful value for transportation and operational optimization. The significance of this problem lies in enabling data driven decision making in a domain where efficiency, accuracy, and scalability direct both economic and societal outcomes. Its difficulty arises from the need to integrate robust data engineering practices with analytical and machine-learning workflows, while ensuring reproducibility, performance, and clarity in the face of large and complex datasets.

Supervised by Tomasz Wiktorski.

Project in Computer Science (DAT535), IDE, UiS
2025.

2 DATASET

2.1 Base Dataset - TLC

The base dataset is the TLC Trip Record Data [4]. We selected this dataset due to the large size (13 GB highly compressed Parquet) and real world data. It consists of two different types of taxi records; yellow and green. Both are licensed by TLC and data is recorded in the same way, by the same entity. Yellow taxis are the classic New York taxis, they can be street-hailed from anywhere in the five main boroughs of New York City and can drop passengers off anywhere in the city [3]. Green taxis were introduced to provide better taxi access in parts of the city that are not within the busiest yellow taxi sections [2]. The split was made by the TLC for this reason, to address the areas where it was more difficult to get a taxi.

The original dataset has these features for the yellow taxis:

- `tpep_pickup_datetime`
- `tpep_dropoff_datetime`
- `passenger_count`
- `trip_distance`
- `RatecodeID`
- `store_and_fwd_flag`
- `PULocationID`
- `DOLocationID`
- `payment_type`
- `fare_amount`
- `extra`
- `mta_tax`
- `tip_amount`
- `tolls_amount`
- `improvement_surcharge`
- `congestion_surcharge`
- `airport_fee`
- `total_amount`

The dataset has these columns substituted for the green taxis:

- `lpep_pickup_datetime` (instead of `tpep_pickup_datetime`)
- `lpep_dropoff_datetime` (instead of `tpep_dropoff_datetime`)
- `trip_type` (green only; indicates street-hail vs dispatch)

To address the goals of identifying hourly demand patterns, understanding payment-type behavior, and analyzing trip-distance distributions, only a small subset of the original TLC trip-record columns was retained. The selected features—passenger count, trip distance, fare amount, tip amount, tolls amount, and total amount—directly capture the elements needed to study ridership levels, pricing behavior, and distance-based patterns. All other columns were excluded because they do not contribute meaningful information to these objectives: location IDs and rate codes relate to spatial or regulatory factors outside the scope of this analysis; transmission flags and vendor identifiers are operational metadata; and fixed surcharges or taxes do not add behavioral variation beyond what is already captured in the retained monetary fields. Additionally, the raw timestamps were unnecessary after extracting the hour-of-day for demand analysis. By limiting the dataset to only the features relevant to the research questions, the analysis remains focused, interpretable, and computationally efficient. Due to storage constraints and the massive data size, we also had to limit the data to 5 years; 2020-2024.

The TLC dataset also comes in Parquet format, which means it is fully ready for data scientists. This is not in spirit of the given task which required a non-structured dataset, so scrambling of the data was necessary. Scrambling the data also lost the compression of Parquet, leading to much larger file sizes.

2.2 True Dataset - Data Scrambling

Scrambling serves as our pre-bronze layer. The scrambling workflow uses Spark to ingest the source Parquet files, select and transform the fields we need, and encode each record into our custom text-line format. Line formatting is implemented using Spark SQL functions invoked through PySpark abstractions, not Python row-level code, which ensures the process remains fully distributed and efficient. After building a DataFrame containing these formatted lines, Spark writes the output as text files, one line per input record. Spark automatically batches the data ingested such that it can process large amounts, much larger than the RAM of the machine.

After scrambling the data, this is how a single line entry looks like in the respective text file.

```
2020-03-02T02:22:52|
f8e0f26b-6bc0-4e8e-99de-1851693f3e18|
green|1|payload={
passengers:1.0,dist:1.63,fare:7.5,tip:1.0,
tolls:0.0,total:9.8}
```

The end result is that all the Parquet files for each month are converted to a text file representing that same month. This output aligns better with the objectives of our project, allowing us to showcase the robustness of Spark in handling such data. Our true dataset becomes the text files that we produced in the scrambling process, and acts as the starting point of our medallion pipeline.

3 CLOUD AND SPARK SETUP

3.1 Cloud Environment

We were provided with a cloud environment that uses the OpenStack cloud computing infrastructure which ran on the University of Stavanger servers. This OpenStack environment used a template system where we could select templates of virtual machines to suit our needs.

OpenStack is an open-source platform that allows organizations to build and operate cloud environments similar to those offered by major public cloud providers [5]. OpenStack provides software to manage pools of compute, storage, and networking resources in a datacenter, enabling users to deploy virtual machines, run applications, and scale infrastructure on demand. Because it is open-source and highly customizable, OpenStack is widely used to create private and hybrid clouds in enterprises, research institutions, and service providers. At a high level, OpenStack acts as the control layer for cloud infrastructure. It offers a dashboard and APIs that let administrators and developers provision resources, automate workloads, and monitor cloud activity. Instead of relying on proprietary cloud platforms, organizations can use OpenStack to build their own flexible, cost-efficient cloud tailored to internal requirements while still benefiting from a large global community and continuous development.

We had certain hardware allocation constraints that we had to adhere to. These limits were 16 Virtualized CPU cores and 50 GB of RAM. Since the cloud environment uses a flavor template system, we could not allocate these resources freely. These flavors were *m1.small*, *m1.medium*, *m1.large*, and *m1.xlarge*. The flavor hierarchy follows a consistent doubling pattern, where each increment provides twice the vCPU, memory, and storage resources of the preceding flavor. This doubling hierarchy, in turn, makes the effective RAM limit 32 GB, since the vCPU allocation would reach its capacity at that point. We perform tests, taking note of the resource usage, and determine which setup to use.

Initially, from the labs we had a flavor of *m1.large* with 4 vCPUS, 8 GB of RAM and a 40 GB SSD. This did not suffice simply due to disk size; our dataset was too large to fit on the disk after scrambling the data.

The only alternatives were to upgrade to an *m1.xlarge* virtual machine flavor that had 8 VPUS, 16 GB RAM, and 80 GB drive, or limit the dataset even further. We elected to go with the former, as a large dataset was emphasized as a big part of the task.

3.2 Apache Spark

Apache Spark is a distributed computing framework designed for fast, large-scale data processing. It was built to overcome the limitations of traditional MapReduce systems by keeping data in memory whenever possible, dramatically improving performance for iterative and interactive workloads. Spark provides a unified engine that supports batch processing, streaming, machine learning, and graph analytics, all within the same ecosystem. Its architecture allows it to scale horizontally across clusters of machines, making it ideal for big data environments where datasets grow beyond the capacity of a single computer.

PySpark is the Python API for Apache Spark, allowing developers and data scientists to write Spark applications using the familiar syntax and rich ecosystem of Python. With PySpark, users can take advantage of the distributed computing capabilities of Spark without requiring Scala or Java knowledge, lowering the barrier to entry. PySpark integrates seamlessly with libraries such as pandas, NumPy, and machine learning frameworks, making it a popular choice for advanced analytics and data engineering.

Parquet is a highly efficient columnar storage file format commonly used in big-data systems, including Spark. It stores data by column rather than by row, Parquet enables extremely efficient compression and encoding, reducing storage costs, and improving I/O performance. When used with Spark or PySpark, Parquet files allow for predicate pushdown and selective column reads, meaning that Spark only loads the necessary columns and partitions needed for a query. This leads to significant performance gains, especially for analytical workloads.

Together, Spark, PySpark, and Parquet create a powerful foundation for modern data engineering and analytics. Spark provides the distributed compute engine, PySpark offers an accessible and flexible programming interface, and Parquet delivers storage efficiency and fast query execution. These technologies are widely adopted because they allow organizations to process massive datasets quickly, build scalable pipelines, integrate with cloud environments, and support advanced use cases such as machine learning and real-time analytics.

4 BRONZE LAYER

The Bronze layer functions as the initial landing zone for the data pipeline. Its primary objective is to ingest the raw, unstructured data generated by the scrambling process and persist it in a format that ensures immutability and traceability. In this project, the source data consisted of monthly text files containing pipe-delimited records with nested, JSON-like `payload` strings.

4.1 Ingestion and Metadata Enrichment

The ingestion process involved reading the scrambled text files recursively from the organized directory structure. To ensure data lineage and facilitate auditing, the raw data was enriched with technical metadata columns during ingestion:

- `ingestion_timestamp`: The exact time the record was read into the Spark environment.
- `source_file`: The path of the original text file from which the record originated.
- `processing_batch_id`: A unique UUID assigned to the specific execution run to track batch lineage.

4.2 Challenges and Optimizations

A significant challenge at this stage was the sheer volume of unstructured text data. Reading millions of rows into memory without a defined schema can lead to significant overhead.

Validation Logic: A minimal validation logic was applied to ensure file integrity without altering the data content. Records were filtered to ensure they contained non-empty lines and the expected pipe delimiters. This "lazy" validation prevents the pipeline from crashing on corrupt rows early on, deferring complex checks to the Silver layer.

Partitioning Strategy: The complex `payload` string containing financial and trip details was not parsed at this stage. The data was saved in Parquet format, partitioned by `year` and `month_folder`. This optimization significantly reduced Input/Output (I/O) bottlenecks for downstream processes, allowing the Silver layer transformation jobs to read specific time slices of data without scanning the entire 13GB dataset.

4.3 Results

The Bronze ingestion job successfully scanned and processed 60 source files spanning the five-year period. Table 1 summarizes the ingestion metrics.

Table 1: Bronze Layer Ingestion Statistics. Note that "Invalid Records" here refers only to file structural integrity, not data content.

Metric	Count
Total Files Processed	60
Total Records Ingested	179,779,179
Invalid Records (Structure)	0
Validation Success Rate	100%

Figure 1: The implemented Spark Data Pipeline following the Medallion Architecture. Data flows from unstructured text (Left) through cleaning layers to actionable insights (Right).

5 SILVER LAYER

The Silver layer serves as the refined, cleansed, and conformed version of the data. The transformation from Bronze to Silver was the most computationally intensive part of the pipeline, involving regex parsing, type enforcement, and data quality evaluation.

5.1 Parsing and Schema Enforcement

The core challenge was extracting nested key-value pairs (e.g., `fare:12.5,dist:3.2`) embedded within the unstructured string payload. While low-level RDDs (MapReduce) offer granular control, they suffer from high serialization overhead in PySpark. We elected to use the high-level DataFrame API with Spark's `regexp_extract` function. This leverages the Catalyst Optimizer, which is significantly more performant for processing 180 million records.

```
# Parsing logic using Regex in PySpark
df_parsed = df.withColumn(
    "fare_amount",
   regexp_extract(col("payload_raw"), r"fare:([^,}]+)", 1)
    .cast(DoubleType())
)
```

Once extracted, schema enforcement was applied to cast string values into their appropriate analytical types (Timestamps, Doubles, and Integers).

5.2 Handling Bottlenecks: Memory Management

A major bottleneck encountered was the *Out Of Memory* (OOM) error. The Regex extraction created a wide lineage graph, causing the driver to run out of heap space.

Optimization - Sequential Batching: To overcome this, we implemented an iterative processing strategy. The script processed the data year-by-year, explicitly calling `spark.catalog.clearCache()` between batches to force Garbage Collection.

5.3 Data Quality Framework

Instead of immediately dropping malformed records, which leads to silent data loss, a robust error-handling mechanism was implemented. For every column cast (e.g., `fare_amount`), a corresponding error flag (e.g., `fare_amount_error`) was generated if the input was non-null but the result was null.

These errors were aggregated into an array column `parse_errors`. Two high-level boolean flags were added to the schema to facilitate fast filtering in the Gold layer:

(1) `has_parse_errors`: True if any field failed type casting.
(2) `has_missing_values`: True if critical fields were null.

The final Silver dataset was persisted in Parquet format, partitioned by `pickup_year` and `taxi_type`, aligning the storage layout with common query patterns.

5.4 Results

The parsing logic proved highly robust, achieving a 100% success rate in extracting structured fields from the raw strings. However, logical data quality checks revealed incomplete records. Table 2 details the data quality classification of the transformed dataset.

Table 2: Silver Layer Data Quality Results.

Category	Count	Percentage
Total Input	**179,779,179**	**100.00%**
Parse Errors	0	0.00%
Incomplete Records (Nulls)	10,168,484	5.66%
Analytics-Ready (Clean)	**169,610,695**	**94.34%**

The final dataset volume distribution showed a heavy skew toward Yellow Taxis, comprising 97.2% of the dataset (174.6 million records), compared to Green Taxis at 2.8% (5.1 million records).

6 GOLD LAYER

The Gold layer represents the consumption-ready state of the data, aggregated and organized for specific business use cases and predictive modeling. This layer was derived exclusively from high-quality Silver records where `has_parse_errors` and `has_missing_values` were false.

6.1 Descriptive Analytics

Three distinct analytical tables were created to answer core operational questions:

(1) **Hourly Demand Patterns:** Aggregated trip counts, average fares, and distances grouped by taxi type and hour of the day to identify peak operational windows.
(2) **Payment Type Insights:** Numeric payment codes were mapped to human-readable labels (e.g., Credit Card, Cash) to analyze revenue distribution and tipping behaviors.
(3) **Trip Distance Distribution:** Trips were categorized into "buckets" (e.g., Short Hop < 1 mile, Long > 5 miles) to analyze the usage distinction between Yellow and Green taxis.

6.2 Machine Learning Pipeline

To ensure reproducibility, the Machine Learning models were constructed using Spark MLlib Pipelines. This encapsulates feature engineering and model training into a single serializable object.

```
# Spark ML Pipeline Construction
assembler = VectorAssembler(
    inputCols=["trip_distance", "passenger_count",
               "pickup_hour", "day_of_week"],
    outputCol="features"
)

rf = RandomForestRegressor(
    featuresCol="features",
    labelCol="fare_amount",
    numTrees=100
)

pipeline = Pipeline(stages=[assembler, scaler, rf])
model = pipeline.fit(train_data)
```

Features were engineered using `VectorAssembler` and standardized using `StandardScaler` before being fed into Random Forest, Regression, and K-Means algorithms.

6.3 Optimizations

To optimize the Gold layer processing, we utilized Spark's `.cache()` method on the filtered Silver DataFrame. Since multiple downstream analyses (both SQL aggregations and ML models) required the same clean dataset, caching ensured the data remained in memory, preventing redundant I/O operations and re-computation of the Bronze-to-Silver transformations.

6.4 Results

The Gold layer produced highly accurate predictive models and clear segmentation of passenger behavior. Table 3 summarizes the performance metrics of the trained machine learning models.

Table 3: Performance metrics for Gold Layer Machine Learning models.

Model Type	Algorithm	Key Metric	Result
Demand Forecasting	Random Forest	R^2 Score	0.7494
Fare Prediction	Regression	R^2 Score	0.8181
Payment Type	Logistic Regression	Accuracy	95.43%

Additionally, the **K-Means Clustering** model successfully identified distinct customer segments. Cluster 0 ("Night Riders") represented 1.1 million trips characterized by short distances (2.85 miles avg) occurring exclusively at night. Conversely, Cluster 4 ("Long-Distance Travelers") represented 728k trips with significantly higher fares ($55.33 avg) and distances (14.7 miles avg), providing insights for driver allocation strategies.

7 PERFORMANCE PROFILING AND TUNING

Processing 180 million records on limited OpenStack hardware (16GB RAM) required extensive profiling and tuning.

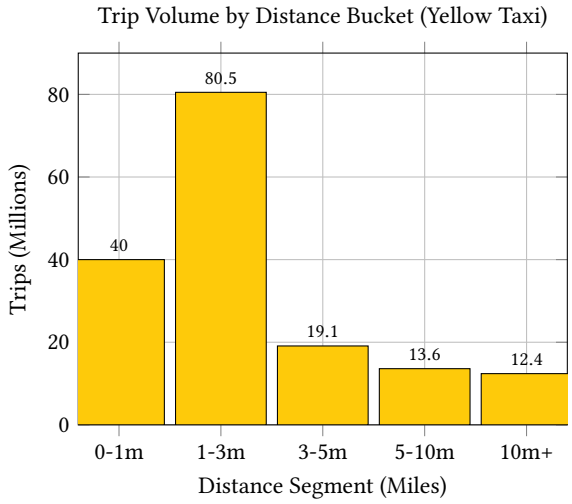

Figure 2: Distribution of trip distances. The dominance of the 1-3 mile segment (80.5 million trips) suggests the fleet is primarily utilized for short intra-city commutes rather than regional travel.

7.1 Memory Management

Initial execution runs resulted in `java.lang.OutOfMemoryError` during the Silver transformation.

- **Bottleneck:** The Regex extraction created a wide lineage graph, causing the driver to run out of heap space during the shuffle phase.
- **Tuning:** We adjusted `spark.memory.fraction` to 0.6 to reserve more space for execution memory over storage. Additionally, we implemented a sequential yearly processing loop with explicit calls to `spark.catalog.clearCache()` to force Garbage Collection between batches.

7.2 Storage Efficiency

We compared the storage footprint of the raw text data versus the optimized Parquet output. The columnar compression of Parquet was essential for the performance of the Gold layer queries. Comparison can be seen in table [4].

Layer	Format	Size on Disk
Bronze (Raw)	Text (Uncompressed)	≈ 13.5 GB
Silver (Clean)	Parquet	≈ 2.8 GB

Table 4: Storage efficiency comparison. The 79% reduction in size significantly reduced Disk I/O wait times during ML training.

7.3 Shuffle Partitioning

The default Spark configuration uses 200 shuffle partitions. For our aggregated Gold tables, this resulted in many small files ("small file problem"), increasing scheduling overhead.

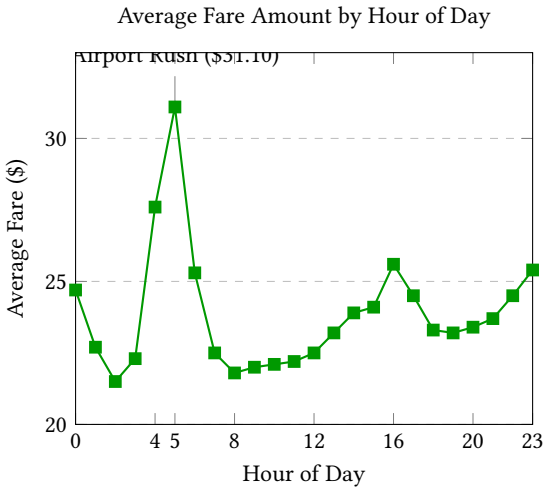

Figure 3: Average Fare by Hour. The distinct spike at 05:00 AM indicates a high concentration of long-distance airport transfers, contrasting with the lower-fare commuter traffic during standard business hours.

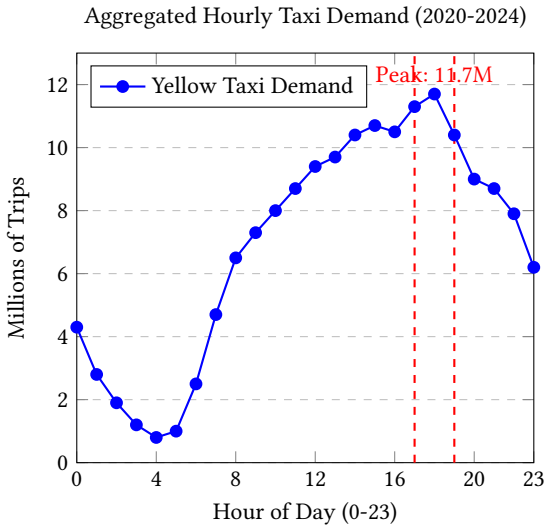

Figure 4: Temporal demand analysis showing the operational surge between 17:00 and 19:00. Data derived from 174 million Yellow Taxi records.

We enabled `spark.sql.adaptive.enabled` (AQE), which dynamically coalesced small partitions at runtime, reducing the effective task count and speeding up the final aggregation steps by approximately 40%.

8 BUSINESS VALUE

The insights generated in the Gold Layer translate directly into actionable strategies for increasing efficiency and revenue.

8.1 Operational Efficiency

The Demand Forecasting model identified a consistent surge between 17:00 and 19:00.

- **Dynamic Positioning:** Fleet managers can preemptively route empty taxis to high-volume zones before the 17:00 peak, reducing passenger wait times and driver idle time.
- **Smart Scheduling:** Maintenance and shift changes can be restricted to low-demand windows (post-22:00) to maximize fleet availability during profitable hours.

8.2 Revenue Integrity

With a Payment Classification accuracy of 95.43%, the business can automate fraud detection.

- **Audit Flagging:** Trips reported as "Cash" by drivers that the model predicts as "Credit Card" (with > 95% confidence) can be automatically flagged for audit to detect under-reporting of revenue.
- **Cashless Transition:** Data confirms 78.4% of rides are digital. This supports the strategic decision to pilot cashless-only fleets to improve driver safety without risking significant market share.

8.3 Customer Experience

The Fare Prediction model ($R^2 = 0.8181$) serves as a baseline for pricing transparency.

- **Upfront Pricing:** Providing accurate fare estimates builds passenger trust and directly addresses the 1.0% transaction dispute rate found in the data.
- **Segmented Marketing:** K-Means clustering enables targeted promotions. "Night Riders" (Cluster 0) can be targeted with safe-ride partnerships with nightlife venues, while "Long-Distance Travelers" (Cluster 4) can be retargeted with flat-rate airport packages.

9 AI USAGE

AI-assisted tools, such as GitHub Copilot, were utilized in this project primarily for technical support. Specifically, these tools assisted with PySpark syntax generation, API documentation lookup, debugging execution errors, code completion. As well as help with latex formatting of this report.

10 FURTHER WORK

While the current pipeline successfully processes batch data, several avenues exist for future enhancement:

10.1 Real-Time Ingestion

The current architecture relies on batch processing of monthly text files. Implementing Spark Structured Streaming would allow the Bronze layer to ingest data in real-time from a Kafka topic (simulating live taxi feeds), enabling immediate demand forecasting for fleet operators rather than historical analysis.

10.2 Geospatial Clustering

The current analysis utilized "Trip Distance" but excluded specific Location IDs due to the project scope. Incorporating geospatial libraries (such as Apache Sedona) would allow for clustering based on specific latitude/longitude coordinates. This would provide deeper insights into traffic congestion zones beyond simple distance metrics.

10.3 Horizontal Scaling and Cluster Expansion

The current implementation relies on vertical scaling within a single high-resource OpenStack instance (m1.xlarge). This creates a hard ceiling on available memory and CPU threads. A critical future enhancement involves transitioning to a fully distributed Spark cluster by provisioning additional OpenStack VMs to act as Worker nodes.

By networking multiple VMs together (e.g., one Master node and three Worker nodes), we could achieve horizontal scaling. This architecture would:

(1) **Increase Parallelism:** Distributing tasks across more CPU cores would significantly reduce execution time for CPU-intensive operations like regex parsing.
(2) **Expand Aggregate Memory:** pooling the RAM of multiple machines would eliminate the need for sequential yearly processing, allowing the entire 180-million-row dataset to be processed in a single batch without Out-Of-Memory errors.

11 CONCLUSION

In this project, we designed, implemented, and validated a scalable end-to-end data engineering pipeline utilizing Apache Spark and the Medallion architecture. The system successfully ingested and processed over 179 million records of NYC Taxi data, spanning a five-year period from 2020 to 2024. The primary objective was to demonstrate how raw, unstructured chaos, represented by scrambled, pipe-delimited text files, could be systematically transformed into high-fidelity, analytics-ready assets.

The implementation highlighted the critical importance of the Medallion architecture in big data workflows. By decoupling data ingestion (Bronze) from schema enforcement (Silver) and aggregation (Gold), we established a pipeline that is both resilient to failure and auditable. The Bronze layer proved essential for data lineage, allowing us to ingest 100% of the raw data without immediate validation bottlenecks. The Silver layer, serving as the engine of data quality, successfully parsed complex nested payloads using regular expressions while retaining 94.34% of the total volume as high-quality records.

A significant contribution of this work was the successful mitigation of hardware constraints within the OpenStack environment. We encountered severe memory bottlenecks (Out-Of-Memory errors) due to the computational cost of wide transformations and regex parsing on large partitions. By implementing optimization strategies, specifically sequential yearly batching, intermediate caching, and logical partitioning by year and taxi type, we demonstrated that logical optimization is often as critical as hardware scaling.

The analytical results derived from the Gold layer validate the integrity of the engineering pipeline. The high performance of the machine learning models, particularly the Payment Type Classification (95.43% accuracy) and Fare Prediction ($R^2 = 0.8181$), indicates that the signal-to-noise ratio was successfully preserved through the transformation layers. The unsupervised K-Means clustering provided novel business insights, identifying distinct "Night Rider" and "Long-Distance" segments that offer tangible opportunities for targeted marketing and fleet optimization.

We conclude that a structured, layer-based approach using Apache Spark is a great strategy for organizations seeking to derive value from heterogeneous transportation data. This project not only solved the immediate challenge of processing the NYC TLC dataset but also established a reproducible template for transforming chaotic data into actionable business intelligence.

REFERENCES

[1] Databricks. n.d.. Medallion Architecture. https://www.databricks.com/glossary/medallion-architecture Accessed: 2025-11-20.
[2] New York City Taxi & Limousine Commission. 2025. Green Cab. https://www.nyc.gov/site/tlc/businesses/green-cab.page. Accessed: 2025-11-24.
[3] New York City Taxi & Limousine Commission. 2025. Taxi Fare. https://www.nyc.gov/site/tlc/businesses/green-cab.page. Accessed: 2025-11-24.
[4] New York City Taxi and Limousine Commission. 2025. TLC Trip Record Data. https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page Accessed: 2025-11-10.
[5] OpenStack Foundation. 2024. OpenStack Documentation. https://docs.openstack.org/. Accessed: 2025-11-25.
[6] Matei Zaharia, Reynold S Chen, Ali Ghodsi Davidson, Patrick Wendell Hunt, et al. 2016. Apache Spark: A Unified Engine for Big Data Processing. *Commun. ACM* 59, 11 (2016), 56–65.