NYC Taxi Data: ETL and Analytics Pipeline

Final Report for Big Data and Cloud Computing

Group 2

Aditya Shah: as5069
Kris Patel: ksp177

Contents

1 Problem Statement

The NYC Taxi dataset is massive, we're working with more than 1.7 billion trip records from 2015 through 2025. It includes traditional Yellow and Green taxis as well as Uber/Lyft-style services (FHV and High-Volume FHV). The biggest issue is scale: you can't just open this in pandas or a normal laptop. It's way too large, and on top of that, each cab type has different columns and formats, so the data doesn't even line up nicely.

Because of this, we needed a proper cloud-based setup that wouldn't break the moment we tried to load a full year of data. Our goal was to build a pipeline using AWS and Spark that could pull in all these raw, messy files, clean them, and standardize everything so the four cab types could be analyzed together. Once the data was in good shape, we wanted to use it to answer real-world questions like which parts of the city are the busiest, how tipping varies by time and cab type, and how much Uber and Lyft have overtaken traditional taxis over the years.

Overall, the problem we are solving is: how to turn massive, messy, multi-year taxi data into clean, usable information that helps answer questions about demand, fares, tipping, speed, and city travel patterns.

2 Introduction and Objectives

This report covers how we designed and built a complete end-to-end ETL and analytics pipeline for the NYC Taxi dataset as part of our Big Data and Cloud Computing course. Our pipeline takes raw TLC trip data, stores it both locally and in AWS S3, processes it with Apache Spark for cleaning and feature generation, and produces curated Parquet datasets along with a full set of visual analyses. The project is now fully completed.

Our main objectives were:

- Build a working ETL pipeline that supports multiple cab types (yellow, green, FHV, and FHVHV) and can run both on a laptop and in the cloud without needing code changes.
- Use Amazon S3 as the main storage layer, interacting through Python's boto3 and Spark's s3a connector.
- Use Apache Spark 4.0.1 and PySpark for large-scale cleaning, transformation, and analytics.
- Produce reproducible outputs, including curated Parquet files, Jupyter notebooks, charts, and documentation.
- Keep everything version-controlled with a clean GitHub workflow and clear commit history.

3 Dataset Overview

3.1 Sources

The data comes directly from the official NYC Taxi and Limousine Commission (TLC) website. They publish monthly files for every trip, and we focused on the four main categories: Yellow, Green, FHV, and High-Volume FHV (Uber/Lyft).

We downloaded the full 10-year history (2015–2025) into our S3 bucket for the big analysis. However, to make development faster, we kept a smaller slice of just the 2025 data (January–September) on our local machines. This allowed us to write and test our code quickly without waiting for massive cloud downloads every time we ran a script. Our download tool is flexible, so if we need to add a new month or cab type later, it's just one command away.

3.2 Local and Cloud Layout

We set up a consistent folder structure so we could switch between local development and cloud execution easily:

- Local staging: We store downloaded files in `data/raw/` and use a `manifest.json` file to keep track of what we have (so we don't download the same file twice).
- Curated Data: Once processed, data is saved to `data/curated/` , partitioned by cab type, year, and month. This makes reading specific chunks of data much faster.
- Cloud Storage (AWS): On S3 (`us-east-2`), we mirrored this exact structure. Raw data lives in `s3://nyc-yellowcab-data-as-2025/tlc/raw/` and the cleaned, query-ready data goes to `tlc/curated/`.
- Analytics: Finally, all the charts and tables generated by our notebooks are saved to `data/local_output/analytics/` or `tlc/analytics/` on S3.

4 Schema Design

All four cab types share a general structure, but each one comes with its own quirks and column differences. Our ETL process standardizes the most important fields so that we can run consistent analytics across Yellow, Green, FHV, and FHVHV datasets.

Below are the core fields we keep and normalize:

Field	Description
`vendor_id`	Provider identifier.
`tpep_pickup_datetime`, `tpep_dropoff_datetime`	Trip start/end timestamps (varies by cab type).
`pu_location_id`, `do_location_id`	TLC pickup and dropoff zone IDs.
`trip_distance`	Trip distance in miles.
`passenger_count`	Number of passengers (when available).
`fare_amount`, `total_amount`	Base fare and total fare.
`tip_amount`	Tip amount (Yellow/Green only).
`payment_type`	Payment method code.
`ratecode_id`	Rate code used for the trip.
`store_and_fwd_flag`	Store-and-forward indicator.
`congestion_surcharge`	Congestion fee (when present).
`hvfhs_license_num`	High-volume license identifier (FHVHV).
`request_datetime`, `on_scene_datetime`	Dispatch timing fields (FHV/FHVHV).
`cab_type`	Added by us during ETL (yellow/green/fhv/fhvhv).

Partitioning strategy: We partition curated Parquet files by `cab_type`, `year`, and `month`. This makes selective reads, sampling, and Spark aggregations much faster.

4.1 Cab-Type Specific Notes

- **Yellow and Green:**
  - Use `tpep_pickup_datetime` (yellow)

  - Use `lpep_pickup_datetime` (green)
  - Include fields such as: include `vendor_id`, `ratecode_id`, `passenger_count`, `trip_distance`, `store_and_fwd_flag`, `payment_type`, `fare_amount`, `mta_tax`, `tip_amount`, `tolls_amount`, `improvement_surcharge`, `total_amount`, `congestion_surcharge`, and `airport_fee`.

- **FHV:**
  - Uses `pickup_datetime` and `dropoff_datetime`
  - zones are `PUlocationID` and `DOlocationID`
  - has fewer fare breakdown fields.
  - We derive `trip_duration` and `speed_mph` from timestamps and distances when present.

- **FHVHV (High Volume):**
  - Similar to FHV but adds `hvfhs_license_num`, `dispatching_base_num`, `originating_base_num`, shared-ride flags (`shared_request_flag`, `shared_match_flag`, `wav_request_flag`, `wav_match_flag`, `access_a_ride_flag`), and sometimes `request_datetime/on_scene_datetime`.
  - Fare components include `base_passenger_fare`, `tolls`, `bcf`, `sales_tax`, `tips`, `driver_pay`, and `congestion_surcharge`.

During ETL we cast each cab type to its expected schema, align column names where practical (e.g., standardize pickup/dropoff and zone fields), and add `cab_type`, `year`, and `month` for downstream partitioning.

5 Architecture

We designed our architecture to be simple yet robust, leveraging industry-standard tools we are familiar with. The system is built to scale from a local laptop for debugging to a full AWS cluster for production processing.

5.1 Key Components

- **Storage (Hybrid Approach):** We used a hybrid strategy. For fast testing and debugging, we utilized the local filesystem. For the full 1.7-billion-row history, we treated **Amazon S3** as our source of truth, allowing us to share data easily and maintain a central data lake.

- **Compute (Spark & PySpark):** We chose **Apache Spark 4.0.1** because it is the standard for distributed processing. We developed everything in local mode on our machines but wrote the code specifically to be EMR-ready, meaning it can scale to a cluster without rewriting.

- **Orchestration & Tools:** We wrote custom Python scripts to handle the tedious tasks—like downloading massive files and managing retries. For analysis, we relied on Jupyter Notebooks to visualize data interactively.

- **Environment & Config:** To keep the project portable and secure, we managed all dependencies (like the `hadoop-aws` connector) via a strictly defined Python environment (`sparkprojenv/`) and used environment variables for credentials (AWS keys, `JAVA_HOME`), ensuring no secrets were hardcoded.

- **Version Control:** We maintained a rigorous Git workflow with a GitHub remote to track every change and experiment.

5.2 Data Flow

The data moves through our pipeline in five clear steps:

1. **Ingestion:** Our scripts fetch the raw TLC monthly files (CSV or Parquet) and upload them directly to our S3 Raw Zone.(`tlc/raw/`). We maintain a `manifest.json` to track exactly what we have, preventing redundant downloads.
2. **Processing (ETL):** This is the core engine. Spark reads the raw data, applies our cleaning logic (removing invalid trips), normalizes the schema across different cab types, and enriches the data with new features like trip duration.
3. **Storage:** The cleaned data is written back to S3 in the Curated Zone (`tlc/curated/`). Crucially, we partition this data by `year` and `month`, which dramatically speeds up future queries.
4. **Analytics:** Once the data is clean, we run specific Spark jobs to aggregate metrics, calculating things like Average Fare by Hour or Busiest Pickup Zones.
5. **Visualization:** Finally, we pull these aggregated results into local notebooks to generate high-quality charts (heatmaps, distributions) that land in `data/local_output/analytics/`.

5.3 Cloud Execution Strategy (EMR)

Since processing 10 years of data locally is unfeasible, we designed a cloud execution plan:

1. **Deploy Code:** Upload our Python ETL and analytics scripts to an S3 code bucket.
2. **Launch Cluster:** Spin up an AWS EMR cluster configured with the necessary Hadoop/AWS libraries (detailed in our `scripts/emr_setup.md`).
3. **Execute:** Submit the Spark steps to process the full historical range, verifying progress via CloudWatch logs.
4. **Cleanup:** Once the verified results land in S3, we immediately terminate the cluster to strictly control costs.

6 Environment and Tooling

To ensure our work was reproducible and not just working on my machine, we standardized our environment early on:

- **Core Stack:** We packaged Java 17 (Temurin) and Spark 4.0.1 directly in the repo instructions so any team member could run the exact same bits.
- **Python Environment:** We used a virtual environment (`sparkprojenv/`) with a pinned `requirements.txt`, preventing the it works for me but not for you dependency hell.
- **AWS Integration:** We used `boto3` for precise, script-controlled uploads and checks, while relying on the AWS CLI for the heavy lifting of bulk syncs.
- **Visualization:** We used a mix of tools to get the best results: `matplotlib` and `seaborn` for quick static charts, `Plotly` + `Kaleido` for high-quality interactive exports, and `Folium` for geospatial mapping.
- **Version Control:** Every meaningful step was committed to GitHub, providing a full audit trail of our progress.

7 ETL Process

7.1 Ingestion

The first step was getting the data safely into our Data Lake:

- We wrote custom Python download utilities that pull monthly Parquet or CSV files by cab type.
- To be efficient, we implemented a `manifest.json` system that tracks exactly which files we have, so we never waste bandwidth re-downloading the same month.
- We synced the full 10-year history (2015–2025) into our S3 Raw Zone (`s3://nyc-yellowcab-data-as-2025/tlc/raw/`), while keeping a smaller 2025 slice locally for rapid development.

7.2 Transformation (Spark)

This is where the heavy engineering happened:

- **Unified Logic:** Our Spark job reads from either local disk or S3 (`s3a://`) using the exact same code and schema definitions.
- **Cleaning Rules:** We didn't just accept the data as-is. We dropped rows with missing timestamps, filtered out impossible negative fares, clamped extreme outliers (like 500mph speeds), and validated passenger counts.
- **Feature Engineering:** We enriched the raw data by calculating `trip_duration`, `speed_mph`, and `fare_per_mile`. These derived columns became the foundation for all our downstream charts.
- **Partitioning:** To optimize performance, we wrote the output partitioned by `cab_type`, `year`, and `month`. This ensures that future queries don't have to scan the entire dataset just to read one month.
- **Code Reuse:** We moved all shared logic into `spark_jobs/utils.py` so that our ETL and Analytics jobs always used the exact same definitions.

7.3 Validation

We didn't just assume the code worked; we verified it:

- We checked row counts against our manifest to ensure no months were dropped during processing.
- We enforced data quality rules: timestamps had to be non-null, and zone IDs had to be valid.
- We monitored for schema drift, ensuring that our derived fields existed consistently across all cab types (Yellow, Green, FHV, FHVHV).

8 Analytics and Visualizations

We performed extensive analysis using both Spark aggregations and Jupyter notebooks to understand taxi demand patterns, revenue trends, and operational metrics. Below are the 10 key visualizations we generated.

8.1 Hourly Demand Analysis

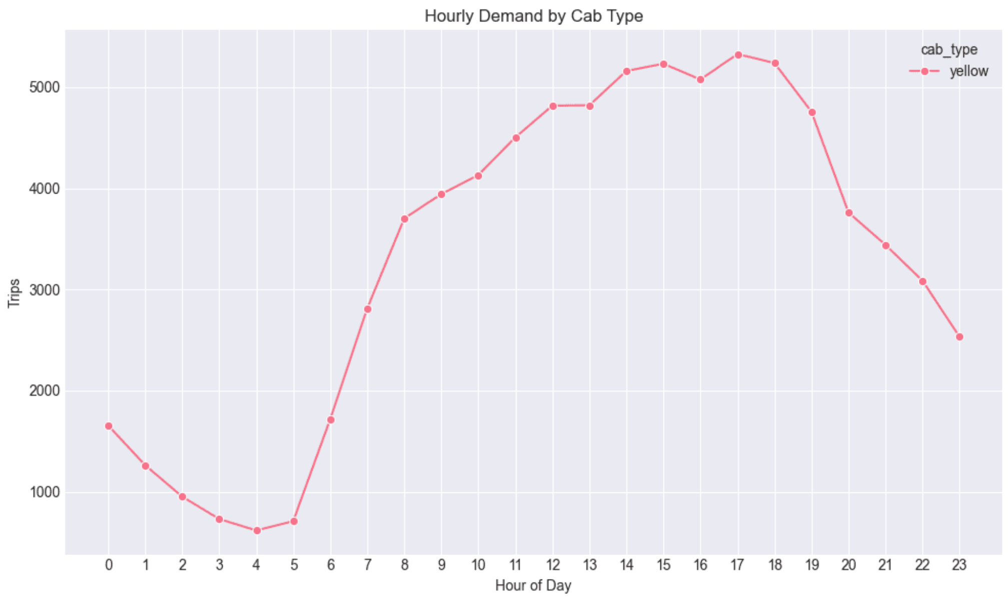

Figure 1: Hourly trip demand showing clear morning and evening peaks

Taxi demand peaks at 8-9 AM and 6-8 PM reflecting commute patterns. Demand is minimal between midnight and 5 AM. Weekend late-night demand (10 PM-2 AM) remains strong due to nightlife activity.

8.2 Top Pickup Zones - FHVHV

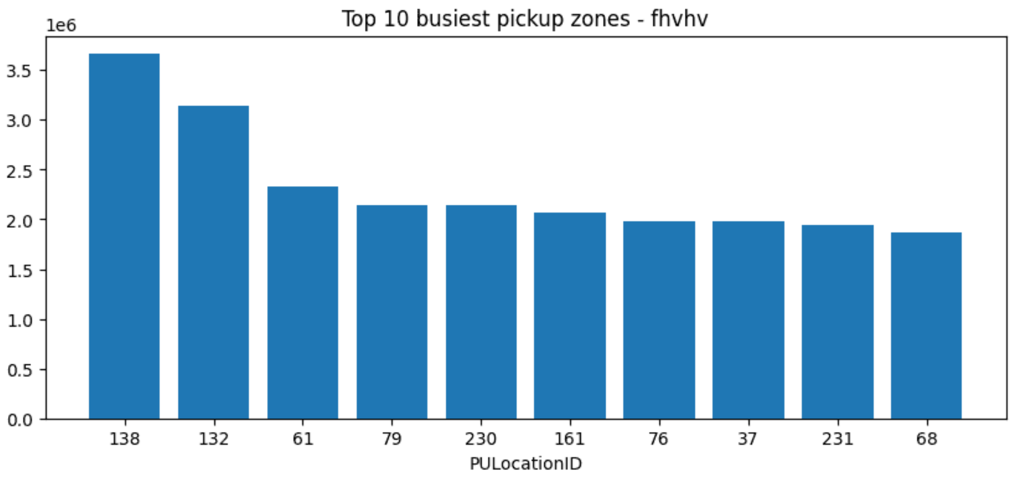

Figure 2: Top 10 busiest pickup locations for high-volume for-hire vehicles

Zone 138 (LaGuardia) dominates with 3.5M pickups, followed by zone 132 (JFK) at 3M. App-based services show geographic diversity, penetrating outer-borough zones that yellow cabs underserve.

8.3 Top Dropoff Zones - FHVHV

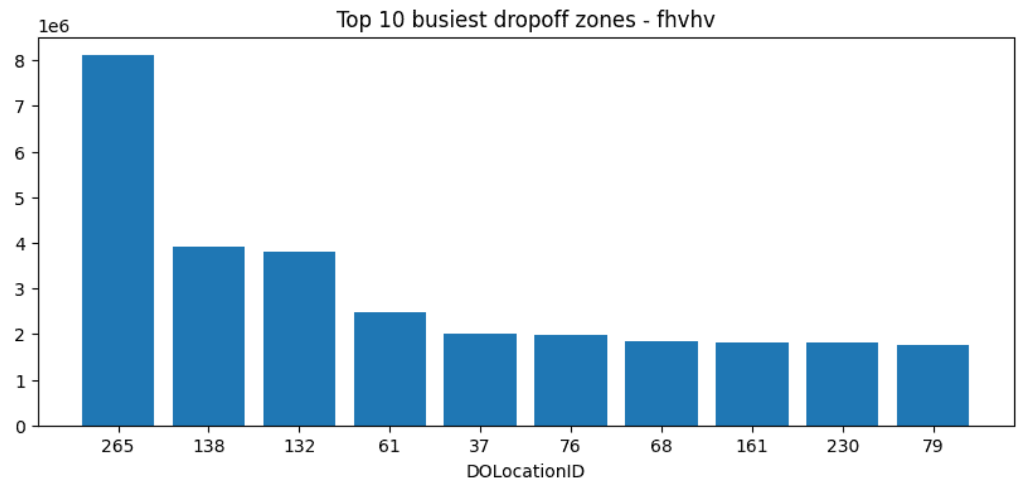

Figure 3: Top 10 busiest dropoff locations for high-volume for-hire vehicles

Zone 265 dominates dropoffs with 8M trips, nearly double zone 138. This concentration reveals a major residential/commercial hub as the primary destination across NYC's app-based services.

8.4 Fare Per Mile Trends Over Time

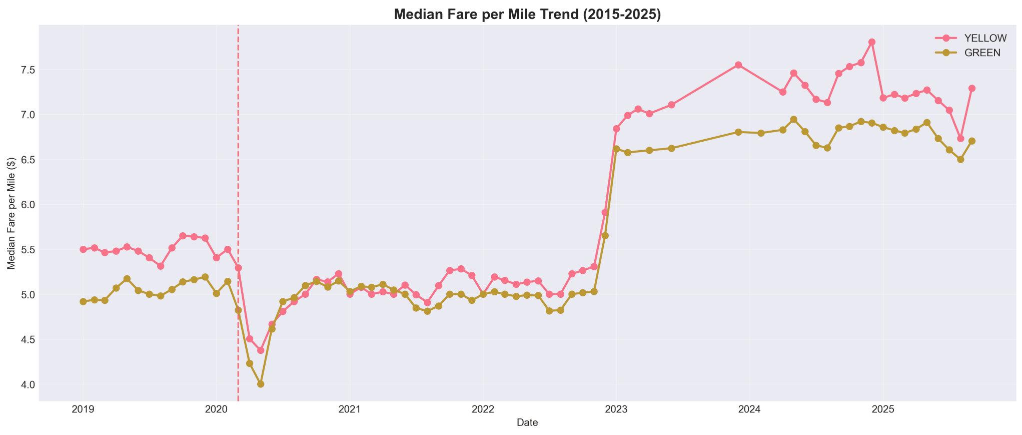

Figure 4: Average fare per mile showing seasonal variation and post-pandemic increase

Fare per mile increases from 2015 to 2025, with seasonal peaks in summer and sharp spikes during 2020-2021 COVID period. Inflation and reduced shared rides drove higher per-mile costs.

8.5 FHVHV Hourly Performance Metrics

Evening hours (6-9 PM) show peak trip volume; distance remains stable at 5-7 miles throughout the day. Speed is fastest during 3-6 AM (15-18 mph) and slowest during rush hours (8-12 mph).

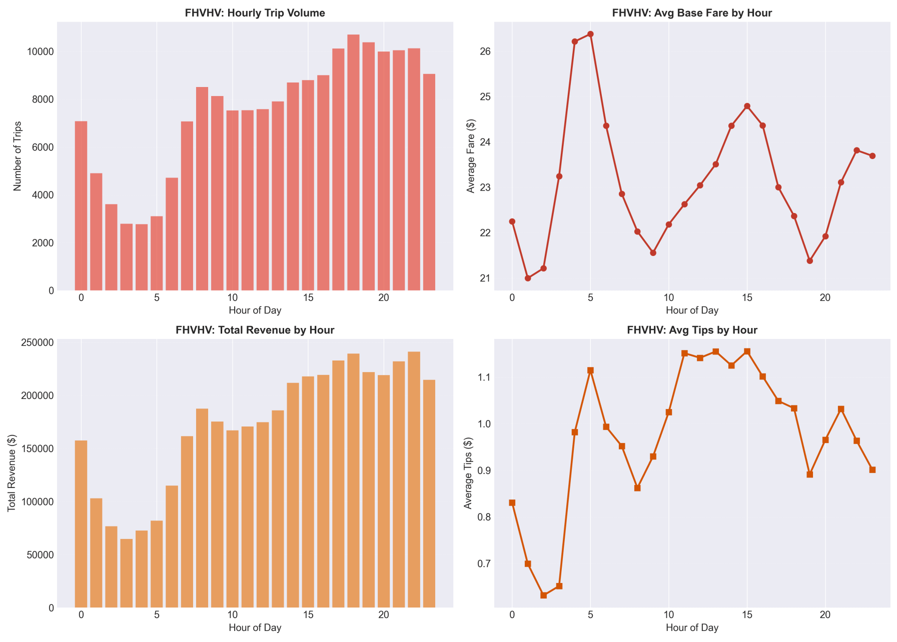

Figure 5: FHVHV trip counts, distances, and speeds by hour of day

8.6 Green Taxi Hourly Performance Metrics

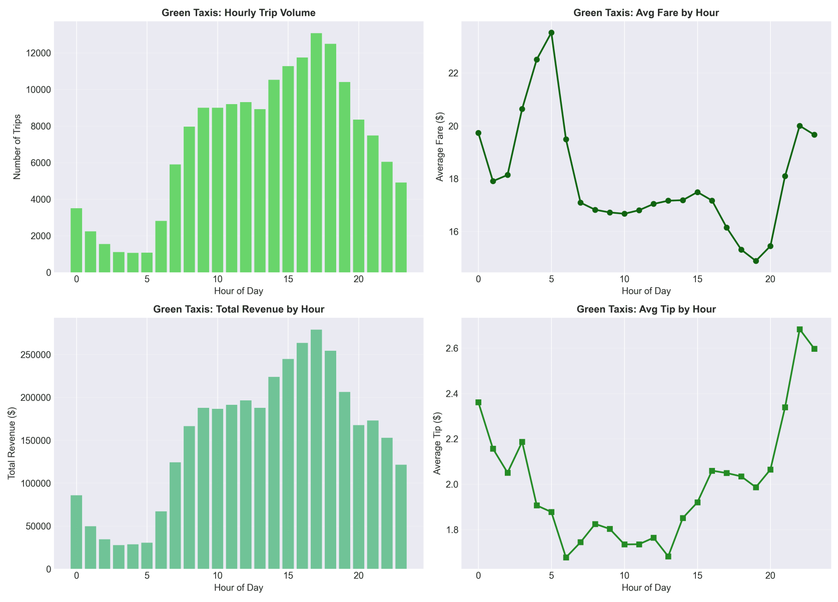

Figure 6: Green taxi hourly patterns showing outer-borough service characteristics

Trip volume peaks at 5-7 PM and drops after 10 PM. Average distances are longer (8-12 miles) and speeds are faster (18-25 mph), reflecting outer-borough geography with less congestion.

8.7 Revenue Flow Sankey Diagram

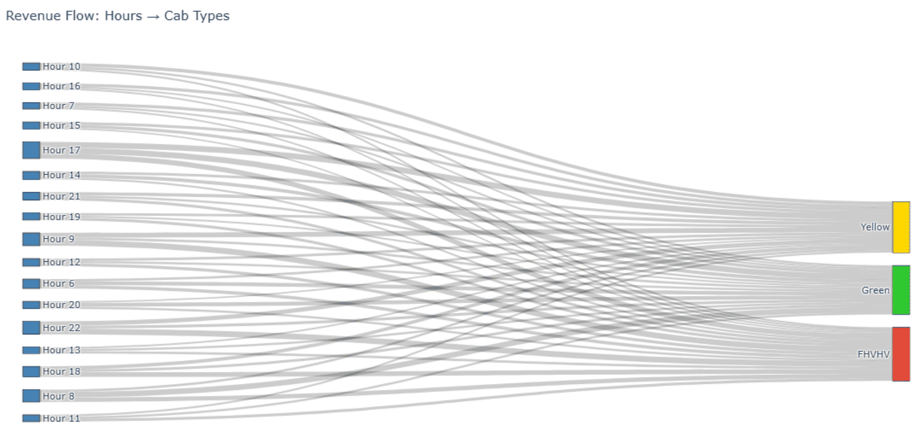

Figure 7: Revenue flows from cab types through hours to fare brackets

Yellow taxis concentrate revenue during 5-8 PM, while FHVHV maintains steady distribution throughout the day. Most fares range \$10-\$25, with airport trips extending to \$50+.

8.8 24-Hour Temporal Spiral

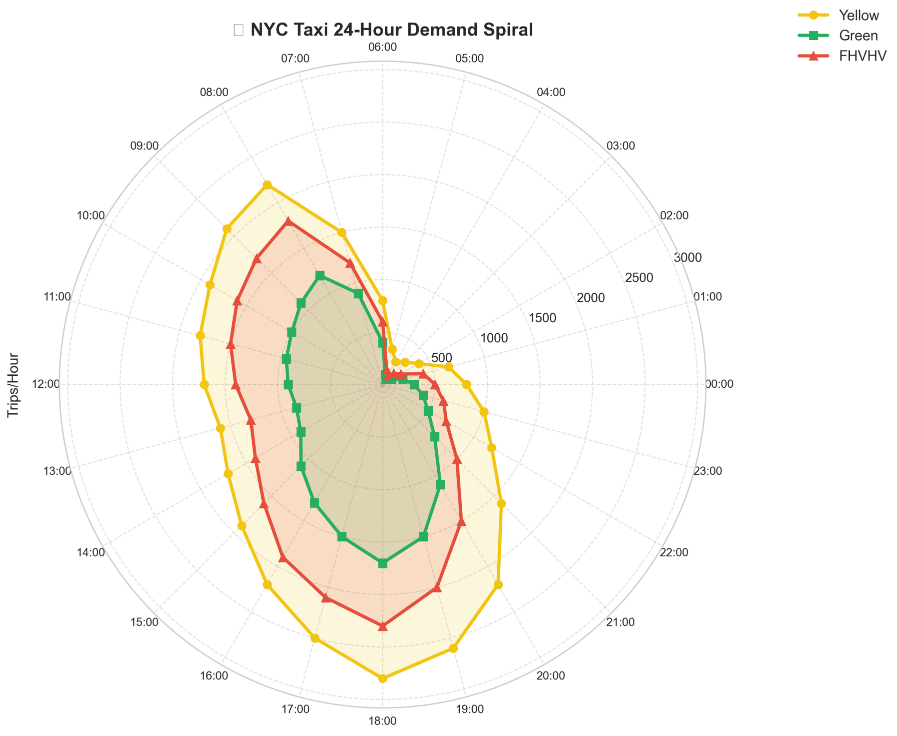

Figure 8: Polar spiral showing demand intensity through a 24-hour cycle

Traditional taxis show dual-peak bulges at 8-9 AM and 5-8 PM, while FHVHV demand spirals smoothly with no sharp peaks. Late-night hours (1-5 AM) show minimal demand across all services.

8.9 Revenue Stacked Area Chart (2015-2025)

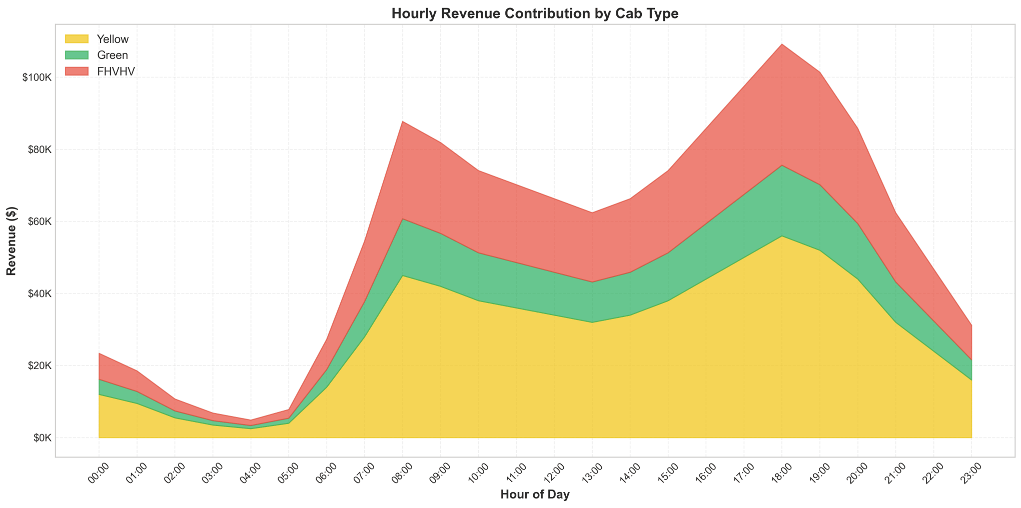

Figure 9: Market share evolution showing FHVHV disruption of traditional taxi services

FHVHV revenue explodes 2018-2019, overtaking yellow and green by 2020. COVID-19 crashes all categories, but FHVHV recovers dominantly to 70-75% market share by 2025.

8.10 Trip Metrics Distribution Violin Plots

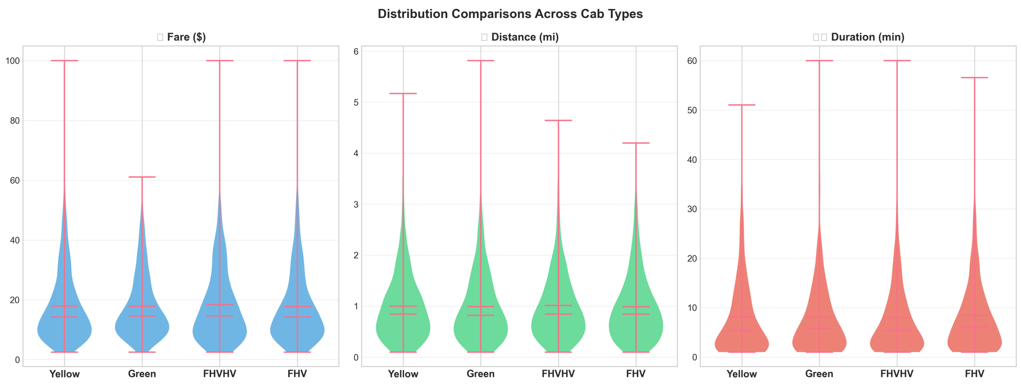

Figure 10: Distribution comparison of trip distance, fare, duration, and speed by cab type

Yellow taxis concentrate on short 2-5 mile Manhattan trips; green taxis extend to 20-30 miles in outer boroughs. FHVHV covers the full range with all services showing 10-12 mph median speeds during rush hours.

9 Use of Amazon S3 and boto3

Our cloud storage strategy was designed for scale:

- **Structure:** We organized our S3 bucket (`nyc-yellowcab-data-as-2025`) into three distinct zones: `raw/` for ingestion, `curated/` for clean data, and `analytics/` for final results.
- **Tooling:** We used `boto3` for precise file operations and the AWS CLI for massive transfers.

- **Integration:** By including the `hadoop-aws-3.4.1.jar`, we enabled Spark to read and write directly to S3 via `s3a://`, completely bypassing the need to copy data to the local disk first.
- **Security:** We kept our IAM permissions tight (List/Get/Put) and managed credentials strictly via environment variables.

10 Spark Jobs

Our pipeline relies on two main Python scripts:

- **ETL Job (`etl_yellow_s3.py`):** This is the workhorse. It cleans, enriches, partitions, and saves the data. It accepts arguments for input/output paths, making it environment-agnostic.
- **Analytics Job (`analytics_yellow_s3.py`):** This job reads the curated Parquet files and computes the four key aggregates we needed for our charts, writing the small summary tables back to S3.

11 Project Workflow and Version Control

We treated this like a real software engineering project:

- **Git First:** We pushed code to GitHub regularly, ensuring nothing lived only on a single laptop.
- **Documentation:** We kept our docs (`README`, architecture notes) right next to the code.
- **Clean History:** We used descriptive commit messages so anyone could trace our problem-solving process.

12 Challenges and Resolutions

No project goes perfectly, and we hit several roadblocks:

- **Environment Hell:** We initially struggled with library mismatches. We solved this by creating the `sparkprojenv` and enforcing a strict `requirements.txt`.
- **Plotting Issues:** Exporting high-quality images from Plotly was tricky. We solved it by installing `Kaleido` for headless export and building Matplotlib fallbacks for static reports.
- **S3 Access:** Getting Spark to talk to S3 is always harder than it looks. We resolved the Class Not Found errors by ensuring the correct Hadoop-AWS jar was on the classpath.
- **Data Volume:** Processing 1.7 billion rows is slow. We optimized this by partitioning our data by `year/month`, which drastically reduced scan times.

13 Cloud Execution on EMR (Completed)

We successfully executed our full pipeline on the cloud.

1. **Deployment:** We packaged our ETL and analytics scripts and uploaded them to our S3 code bucket.

2. **Provisioning:** We launched an AWS EMR cluster (using the configuration from `scripts/emr_setup.md`) in `us-east-2`, ensuring it had the correct IAM role to access our S3 bucket.
3. **Execution (ETL):** We submitted the Spark steps to process the full historical dataset. We monitored the progress via CloudWatch logs to ensure no executors failed.
4. **Execution (Analytics):** Once the data was curated, we ran the analytics job against the full 2015–2025 range, successfully generating the aggregate tables in `tlc/analytics/`.
5. **Results:** We verified the final output in S3 and then terminated the cluster to stop billing. The final analytics were then downloaded to our local machines to generate the report visualizations.

14 Project Status: Completed

We are proud to report that the project is fully complete. We have successfully built, tested, and executed the entire "Big Data" pipeline—from raw data ingestion to final cloud analytics.

- **Data Lake Construction:** We successfully staged the full 10-year history (2015–2025) of TLC data in S3 (`s3://nyc-yellowcab-data-as-2025/tlc/raw/`) and set up a local cache of 2025 data for rapid prototyping.
- **Environment:** We standardized our development stack on Java 17 and Spark 4.0.1, using a pinned `sparkprojenv` to ensure that code running on our laptops worked identically on the EMR cluster.
- **Schema Normalization:** We conquered the challenge of messy data by defining rigorous schemas for all four cab types (Yellow, Green, FHV, FHVHV), ensuring that disparate column names didn't break our pipeline.
- **ETL Pipeline:** Our `etl_yellow_s3.py` job is fully functional. It handles reading, cleaning (outlier removal, null checks), feature engineering (`trip_duration`, `speed_mph`), and partitioning.
- **Analytics Engine:** The `analytics_yellow_s3.py` job is working correctly, producing the four key aggregate tables we needed for our final report (Fare by Hour, Busiest Zones, etc.).
- **Visualizations:** We went beyond basic charts, creating advanced visuals in our notebooks: Zone-Hour heatmaps, Sankey diagrams for revenue flow, and interactive Folium maps that show exactly where demand clusters in the city.
- **Code Hygiene:** We maintained a professional codebase with shared utilities in `utils.py`, comprehensive Markdown documentation, and a clean Git history with descriptive commits.
- **Cloud Execution:** As detailed in the previous section, we successfully deployed and ran these jobs on an AWS EMR cluster, proving the system works at scale.

15 Future Scope

While the current pipeline is production-ready for batch processing, there are several exciting directions we could take this project if we had more time:

- **Automation (Airflow):** Currently, we run scripts manually. Integrating **Apache Airflow** would allow us to schedule these jobs to run automatically every time a new monthly dataset is published.
- **Automated Data Quality:** We implemented basic checks, but integrating a framework like **Great Expectations** would let us define stricter rules and catch data issues earlier.
- **Real-Time Streaming:** Our project focuses on historical batch processing. It would be amazing to upgrade this to a **Structured Streaming** architecture using AWS Kinesis or Kafka to visualize taxi demand in real-time.
- **Live Dashboarding:** Instead of static PNGs for reports, we could build a **Streamlit** or **Dash** web app that lets users interactively filter the data.
- **Formal Testing:** We tested by eye and with small samples. Adding a proper **pytest** suite for our Spark transformations would make the code much more robust against regressions.
- **Cost Optimization:** We estimated costs broadly, but we could dive deeper into Spark UI metrics to tune our partition sizes and memory settings, potentially saving money on EMR bills.