Urban Mobility Data Explorer

Technical Documentation Report

NYC Yellow Taxi Trip Analysis | February 2026

GitHub: github.com/FabriceMbarushimana/urban-mobility-data-explorer

Dataset	Records Cleaned	API Endpoints	Tables
NYC Yellow Cab Jan 2019	~14,684 valid trips	13 RESTful endpoints	4 MySQL tables

1. Problem Framing and Dataset Analysis

1.1 Dataset Description and Context

The primary dataset consists of NYC Yellow Taxi trip records from January 2019, sourced from the NYC Taxi & Limousine Commission (TLC). The raw dataset contained approximately 7.67 million trip records, making it one of the most detailed publicly available urban mobility datasets. Each record captures a complete trip lifecycle including timestamps, geo-spatial zone identifiers, fare components, passenger counts, and payment methods.

To make the dataset tractable for development and analysis, we worked with a 15,000-record sample. The dataset was enriched by joining three auxiliary sources:

- taxi_zone_lookup.csv — maps 265 zone IDs to borough and neighborhood names
- taxi_zones.json — geographic zone boundaries for spatial analysis
- NYC TLC Shapefiles — polygon data for geospatial visualizations

1.2 Data Challenges and Quality Issues

During exploratory analysis, we identified several data quality issues that required systematic handling:

Issue Category	Description	Resolution

Duplicate Records	Repeated trip IDs and timestamps	Removed via deduplication
Missing Fields	Null fare, distance, or location values	Filtered out entirely
Invalid Ranges	Trips >100 miles, fares >$500, pax outside 1-6	Range validation applied
Temporal Anomalies	Dropoff before pickup timestamp	Records excluded
Extreme Outliers	Unusual speed/fare combinations	IQR-based detection

In total, 267,314 records were excluded from the full dataset — representing a 9.02% exclusion rate. Within our 15,000-record sample, approximately 316 records were removed after cleaning, yielding 14,684 valid trips.

1.3 Assumptions Made During Data Cleaning

- Passenger counts outside the range of 1-6 were considered erroneous, consistent with NYC TLC regulations on cab capacity.
- Trip distances exceeding 100 miles were flagged as invalid given the geographic constraints of NYC metro area.
- Fare amounts above $500 were treated as outliers or data entry errors, not reflecting actual trip costs.
- Records with dropoff timestamps preceding pickup timestamps were assumed to be system logging errors.
- Zero-distance trips with positive fares were retained if other fields were valid, as these may represent short waits or cancellation fees.

1.4 Unexpected Observation That Influenced Design

During profiling, we discovered that a significant portion of trips were logged with identical pickup and dropoff zones — suggesting either very short local trips or GPS logging failures. This observation influenced our design decision to include a distance_category feature (short, medium, long, etc.) rather than relying solely on raw mileage for analysis. It also prompted us to add an excluded_data_log table in our schema to maintain a full audit trail of rejected records, enabling future review and data quality improvement.

2. System Architecture and Design Decisions

2.1 System Architecture Overview

The Urban Mobility Data Explorer follows a classic three-tier architecture: a HTML/JavaScript frontend, a Python Flask REST API backend, and a cloud-hosted MySQL database via Aiven. Each layer has clearly separated responsibilities, enabling independent development, testing, and scaling.

Architecture Layers

Layer	Technology	Responsibility
Frontend	HTML5, CSS3, JavaScript	Data visualization, user filtering, chart rendering
Backend / API	Python 3.8+, Flask 3.1.2	Business logic, RESTful routing, query orchestration
Database	MySQL 8.0 on Aiven Cloud	Persistent storage, indexing, referential integrity
ETL Pipeline	Python, pandas, geopandas	Data loading, cleaning, feature engineering
Config	python-dotenv (.env files)	Secure credential management

2.2 Backend — Flask API

We selected Flask over heavier frameworks like Django because of its minimal footprint and native compatibility with the scientific Python ecosystem. Flask allowed us to integrate pandas, SQLAlchemy, and geopandas without the overhead of ORM layers or admin scaffolding. This was critical for maintaining query performance over a 7.6 million-row production-scale dataset.

The API exposes 13 RESTful endpoints organized into four logical groups:

- Health & Status — /api/status
- Statistics — /api/stats/summary
- Trip Filtering — /api/trips/list with 10 query parameters
- Analysis — 8 endpoints for hourly patterns, borough breakdown, fare distribution, speed, tips, and weekend comparison
- Custom Insights — /api/routes/top (QuickSort), /api/insights/custom (IQR outlier detection)

2.3 Database — MySQL on Aiven Cloud

We chose MySQL over PostgreSQL for its broad compatibility and simpler configuration for team collaboration. The database is hosted on Aiven's managed cloud platform, which provided built-in backups and connection pooling. To migrate the 14,684-record dataset efficiently, we split the

upload into 150+ chunks of 150,000 records each — significantly faster than traditional single-command migrations.

The schema was designed around referential integrity. Foreign keys enforce that every trip's pickup and dropoff zone IDs match valid entries in the zones table. Eleven indexes were added strategically on high-cardinality query fields: pickup/dropoff datetime, borough, hour, and location IDs.

Table	Columns	Indexes	Purpose
zones	4	2	Zone ID to borough/neighborhood mapping (265 zones)
taxi_zones	6	3	Geographic zone boundaries with GIS shape data
trips	30	6	Main trip records with enriched computed features
excluded_data_log	7	2	ETL audit trail for rejected records

2.4 Frontend Design

The frontend was intentionally designed with a minimalist aesthetic. Given the complexity of the underlying dataset, visual clutter would hinder rather than help user comprehension. The dashboard is structured with summary metrics at the top, filtering controls in the middle, and data visualizations in a logical flow below.

A key challenge was that frontend development occurred in parallel with backend development. To maintain progress, the team used mock JSON data to simulate API responses. This approach enabled layout design, filter logic implementation, and chart rendering before the backend was finalized. Once real endpoints were live, adjustments were made to align field names and response structures. Post-integration debugging addressed issues such as inconsistent data types, undefined fields, and async timing problems.

2.5 Design Trade-offs

Decision	Trade-off Made	Rationale
Flask over Django	Less built-in tooling	Lighter weight, better library compatibility
MySQL over PostgreSQL	Fewer JSON features	Simpler setup, team familiarity
15k sample over full 7.67M	Reduced statistical coverage	Feasible for dev environment performance
Bubble Sort for IQR	$O(n^2)$ complexity	Demonstrates algorithmic thinking; data size is manageable
Mock data in frontend dev	Alignment effort post-integration	Enabled parallel team workflow

3. Algorithmic Logic and Data Structures

In accordance with the project requirements, all algorithms were implemented manually without relying on built-in library functions such as sort(), sort_values(), heapq, Counter, or SQL ORDER BY. Four distinct custom algorithms were developed, each addressing a real problem in the dataset or application.

3.1 Algorithm 1: Fare Per Mile (Manual Feature Engineering)

Problem

Computing fare_per_mile for every trip record without using pandas vectorized operations, to demonstrate record-level data manipulation at scale.

Approach

We iterate through each trip record in a Python loop, performing a guarded division (distance > 0) and capping the result at $100 to handle outliers. This mimics what a pandas apply() would do internally, but implemented explicitly.

Pseudo-code

```
FOR each trip in dataset:
    IF trip.distance > 0:
        fare_per_mile = trip.fare / trip.distance
        IF fare_per_mile > 100: fare_per_mile = 100
    ELSE:
        fare_per_mile = 0
    trip.fare_per_mile = fare_per_mile
```

Complexity Analysis

Time Complexity: O(n) — single linear pass over all n trip records

Space Complexity: O(n) — new fare_per_mile column stored alongside the dataset

3.2 Algorithm 2: Bubble Sort for Premium Trip Ranking

Problem

Identifying the top 10 highest-revenue trips in a given borough requires sorting by fare_amount. The constraint was to avoid SQL ORDER BY or Python's built-in sort functions.

Approach

We implemented Bubble Sort, which repeatedly compares adjacent elements and swaps them if out of order. While not optimal for large datasets, it satisfies the algorithmic demonstration requirement and operates in-place without auxiliary data structures.

Pseudo-code

```
FUNCTION bubble_sort(trips):
```

```
    n = length(trips)
    FOR i FROM 0 TO n-1:
        FOR j FROM 0 TO n-i-2:
            IF trips[j].fare_amount < trips[j+1].fare_amount:
                SWAP trips[j] AND trips[j+1]
    RETURN trips[0:10]  // top 10
```

Complexity Analysis

Time Complexity: O(n²) — nested loops compare every pair of adjacent elements

Space Complexity: O(1) — in-place sort; only a single temporary pointer for swapping

3.3 Algorithm 3: QuickSort for Route Popularity Ranking

Problem

Ranking the most popular routes (pickup-dropoff zone pairs) by trip count to power the /api/routes/top endpoint, without using sort() or ORDER BY.

Approach

QuickSort uses divide-and-conquer: a pivot is selected, and elements are partitioned into those less than and greater than the pivot. This is applied recursively, achieving O(n log n) average performance — significantly better than Bubble Sort for larger datasets.

Pseudo-code

```
FUNCTION quicksort(routes, low, high):
    IF low < high:
        pivot_index = partition(routes, low, high)
        quicksort(routes, low, pivot_index - 1)
        quicksort(routes, pivot_index + 1, high)

FUNCTION partition(routes, low, high):
    pivot = routes[high].trip_count
    i = low - 1
    FOR j FROM low TO high-1:
        IF routes[j].trip_count >= pivot:
            i += 1
            SWAP routes[i] AND routes[j]
    SWAP routes[i+1] AND routes[high]
    RETURN i + 1
```

Complexity Analysis

Time Complexity: O(n log n) average, O(n²) worst case (already-sorted input)

Space Complexity: O(log n) — recursive call stack depth

3.4 Algorithm 4: IQR Outlier Detection

Problem

Detecting anomalous fares, distances, and trip durations during the ETL pipeline without using scipy.stats or pandas quantile methods.

Approach

We first sort the data using our custom Bubble Sort, then manually compute Q1 (25th percentile) and Q3 (75th percentile) by index lookup. The IQR is Q3 - Q1. Any value below Q1 - 1.5*IQR or above Q3 + 1.5*IQR is flagged as an outlier and logged to excluded_data_log.

Pseudo-code

```
FUNCTION detect_outliers(values):
    sorted_vals = bubble_sort(values)
    n = length(sorted_vals)
    Q1 = sorted_vals[floor(n * 0.25)]
    Q3 = sorted_vals[floor(n * 0.75)]
    IQR = Q3 - Q1
    lower = Q1 - 1.5 * IQR
    upper = Q3 + 1.5 * IQR
    RETURN [v FOR v IN values IF v < lower OR v > upper]
```

Complexity Analysis

Time Complexity: O(n²) — dominated by the Bubble Sort step

Space Complexity: O(n) — sorted copy of values stored for quartile calculation

3.5 Algorithm 5: Manual Hourly Aggregation (TripAggregator)

Problem

Computing total trips and revenue per hour of day without using pandas groupby() or SQL GROUP BY.

Approach

We initialize a 24-bucket dictionary (one per hour). On a single linear pass, each trip increments its hour bucket's count and accumulates revenue. Averages are computed in a second O(24) = O(1) pass.

Pseudo-code

```
FUNCTION aggregate_by_hour(trips):
    buckets = {0..23: {count: 0, revenue: 0}}
    FOR each trip IN trips:
        h = trip.pickup_hour
        buckets[h].count += 1
        buckets[h].revenue += trip.total_amount
    FOR each hour IN buckets:
        buckets[h].avg = revenue / count  (if count > 0)
    RETURN buckets
```

Complexity Analysis

Time Complexity: O(n) — single pass over all trip records
Space Complexity: O(1) — fixed 24-bucket accumulator, independent of dataset size

4. Insights and Interpretation

The following three insights were derived from direct data analysis using API queries, custom aggregation algorithms, and visualizations. Each is interpreted in the context of urban transportation patterns in New York City.

Insight 1: Rush Hour Congestion Significantly Impacts Trip Speed

How It Was Derived

Using the TripAggregator's manual hourly aggregation, we computed the average speed (avg_speed_mph) per hour across all 14,684 valid trips. The SpeedAnalyzer then flagged hours where average speed fell below a defined threshold (e.g., 10 mph) as congestion periods. Results were surfaced via the /api/analysis/speed endpoint.

Observation

Average taxi speeds drop sharply between 7:00-9:00 AM and 4:00-7:00 PM — the classic commuter rush hours. Speeds during these windows can be 30-40% lower than off-peak hours (e.g., 2:00-4:00 AM), when streets are nearly empty.

Urban Mobility Interpretation

This pattern confirms that NYC's congestion pricing initiatives are well-targeted. Taxi operators during peak hours face significantly longer trip durations for similar mileage, which directly impacts revenue per hour and driver earnings. The data supports infrastructure policies aimed at reducing peak-hour vehicle volume through congestion charges, improved mass transit, or ride-pooling incentives.

Insight 2: Manhattan Dominates Pickup Volume But Not Revenue Per Trip

How It Was Derived

Borough-level analysis was performed via the /api/analysis/borough endpoint, which aggregates trip counts and average fares by pickup borough. Fare distribution data from /api/analysis/fare-distribution supplemented this with per-bracket breakdowns.

Observation

Manhattan accounts for the overwhelming majority of taxi pickups (over 70% in the sample). However, trips originating from outer boroughs such as Queens and Brooklyn tend to have higher average fares, reflecting longer distances to destinations like JFK Airport, LaGuardia, or cross-borough business travel.

Urban Mobility Interpretation

The high pickup density in Manhattan reflects the borough's role as NYC's commercial and tourist hub, where short-hop trips are frequent but low-revenue. Outer-borough trips, while fewer in number, generate disproportionate revenue per trip. For taxi operators, balancing time spent waiting in Manhattan vs. servicing longer outer-borough routes represents a real optimization challenge — one that dynamic pricing algorithms attempt to address.

Insight 3: Credit Card Payment Correlates with Higher Tips

How It Was Derived

Payment type analysis was conducted via the /api/analysis/payment and /api/analysis/tips endpoints. Cross-referencing payment_type (1 = Credit Card, 2 = Cash) against tip_percentage revealed a clear behavioral split.

Observation

Credit card trips carry an average tip percentage of approximately 18-20%, consistent with the pre-populated tip prompts on NYC taxi payment terminals. Cash trips, by contrast, show near-zero recorded tips — not because passengers tip less, but because cash tips are not captured by the electronic system.

Urban Mobility Interpretation

This insight highlights a data completeness issue inherent in the TLC dataset: cash tip behavior is invisible to the data pipeline. From a policy perspective, the shift toward cashless payment systems has improved data quality and likely increased driver income transparency. For mobility platforms and driver-assistance apps, understanding this split is critical to accurately benchmarking driver earnings and incentive structures.

5. Reflection and Future Work

5.1 Technical Challenges

The project involved several technical hurdles that required iterative problem-solving:

- Database Migration Scale: Uploading 14,684 trip records to Aiven required chunking the migration into 150+ batches of ~150,000 records each. The naive single-command approach timed out repeatedly before we implemented batch processing.
- Parallel Frontend/Backend Development: Building the frontend against mock JSON data introduced structural mismatches once real API responses were integrated. Field names, data types, and nested structures differed from assumptions, requiring a dedicated debugging phase post-integration.
- Async Timing in the Browser: Asynchronous API calls occasionally resulted in chart rendering before data had fully loaded, producing undefined field errors. We resolved this with proper async/await chains and loading state management.
- Geospatial Data Complexity: Merging trip records with 265 polygon-based taxi zones via geopandas required careful CRS (coordinate reference system) alignment and was computationally intensive on large samples.

5.2 Team Challenges

Coordinating across team members with different familiarity levels with Python, SQL, and JavaScript required establishing clear API contracts early in the project. Using a shared GitHub repository with branch-based development helped prevent merge conflicts. Regular syncs on data schema decisions were essential — schema changes late in development had downstream effects on both the ETL pipeline and frontend rendering logic.

5.3 Suggested Improvements

Area	Improvement	Impact
Algorithm Performance	Replace Bubble Sort ($O(n^2)$) with Merge Sort or HeapSort for outlier detection	10-100x faster for large datasets
Data Coverage	Use full 7.67 M-record dataset instead of 15k sample	More statistically robust insights
API Caching	Add Redis caching layer for frequently called endpoints	Reduce DB load, improve response time
Real-Time Data	Integrate TLC live feed or streaming pipeline (Kafka)	Enable live dashboards
Authentication	Add JWT-based auth for multi-user access control	Production-grade security

Frontend	Migrate to React with charting libraries (Recharts/D3)	Richer interactivity
CI/CD Pipeline	Implement GitHub Actions for automated testing and deployment	Code quality and reliability
Spatial Analysis	Enable polygon-based zone-to-zone heatmap visualization	Richer geographic insights

5.4 Future Work — Real-World Product Vision

If this were a real-world product, the Urban Mobility Data Explorer would evolve into a comprehensive transportation intelligence platform. Key directions include:

- Multi-modal integration: Expanding beyond yellow taxis to include for-hire vehicles (FHV), Citi Bike, subway ridership, and bus data to provide a complete picture of NYC mobility.
- Predictive analytics: Training ML models on historical trip data to forecast demand surges by zone and hour, enabling proactive driver positioning.
- Driver earnings optimization: Building a recommendation engine that suggests optimal shift times and pickup zones to maximize revenue per hour.
- Public policy tooling: Providing borough-level dashboards for city planners to measure the impact of congestion pricing, new transit lines, or infrastructure changes on taxi demand patterns.
- Open data API: Publishing a public API layer so urban researchers, journalists, and civic technologists can build on the cleaned, enriched dataset.