NYC TAXI TRIP DASHBOARD Technical Documentation Report

1. Problem Framing and Dataset Analysis

Dataset Overview

The NYC TLC Yellow Taxi dataset contains millions of trip-level records. Our system integrates three components: yellow_tripdata (.parquet) as the fact table with raw trip records (timestamps, distances, fares, rate codes, payment types); taxi_zone_lookup (.csv) as the dimension table mapping LocationIDs to boroughs and service zones; and taxi_zones (.geojson) providing spatial polygon boundaries served to the frontend Leaflet map.

Data Challenges & Cleaning Assumptions

- **Missing Values:** Records missing PULocationID, DOLocationID, trip_distance, or fare_amount were excluded imputation was not viable for these critical fields.
- **Duplicates & Datetime:** Exact duplicate rows dropped. Unparseable timestamps and dropoff-before-pickup records removed as logically impossible.
- **Physical Outliers:** Excluded: distances > 100 mi, durations < 1 min or > 24 hrs, speeds > 100 mph, fares > $500 as physically implausible for NYC taxi service.
- **Fare Integrity:** total_amount validated against sum of fare components within $1.00 rounding tolerance. Cash trips (payment_type=2) with tip_amount > $1.00 flagged as erroneous per TLC data dictionary.

Unexpected Observation

The volume of records with inconsistent total_amount was larger than anticipated. This influenced our schema design. We store both the raw total_amount and a derived calculated_total, preserving a full audit trail rather than overwriting source data.

2. System Architecture and Design Decisions

Architecture Overview

Three-tier architecture: (1) Data Processing Layer Python + Pandas loads, cleans, and enriches the parquet data and inserts it into MySQL. (2) Flask REST API exposes endpoints for trips, borough stats, zone lookups, and serves the GeoJSON boundary file directly from disk. (3) Frontend HTML/CSS/JavaScript renders an interactive dashboard with filters, a data table, a Chart.js bar chart, and a Leaflet zone map.

Stack Justification & Trade-offs

- Python + Pandas chosen for mature parquet handling and vectorized outlier detection. Flask, preferred over Node.js shared codebase with the data pipeline, allows reuse of db_connection and config modules. Flask-CORS enabled for local frontend cross-origin requests.
- MySQL over PostgreSQL: better team familiarity; PostGIS spatial support not required since GeoJSON is rendered client-side by Leaflet rather than queried server-side.
- Derived features (trip_duration_minutes, average_speed_mph, tip_percentage) stored directly in the trips table rather than computed at query time trades storage for performance at millions-of-rows scale.

Database Schema Design

Fact-dimension model: central trips table references taxi_zones via PULocationID and DOLocationID foreign keys. Pre-computed views (borough_statistics, rate_code_statistics, trip_details) serve common dashboard queries without repeating JOIN logic in the API. Indexes on tpep_pickup_datetime, PULocationID, DOLocationID, and fare_amount for fast filtering.

3. Algorithmic Logic and Data Structures

Custom Top-N Route Ranking Manual Selection Sort

We implemented a two-phase algorithm to find the most frequent pickup-dropoff zone pairs without sort_values(), Counter(), heapq, or any built-in sort. Phase 1 builds a frequency dictionary in O(T) time (one pass over T trips). Phase 2 extracts top-N in O(N x K) via manual selection sort each of N passes scans all K unique route keys to find the current maximum. Space: O(K). With K bounded by ~69,000 zone pairs (263 x 263) and N typically 10–20, performance is well within acceptable bounds.

Pseudo-code

```
FUNCTION top_n_routes(trips, n):
  freq_map = {}
  FOR each trip: freq_map[(PULocationID, DOLocationID)] += 1
  result = []; visited = set()
  FOR i in range(n):
    max_key, max_count = None, -1
    FOR key,count in freq_map:
      IF key NOT in visited AND count > max_count: max_key,max_count = key,count
    result.append((max_key, max_count)); visited.add(max_key)
  RETURN result
```

4. Insights and Interpretation

Peak Trip Hours: Dual-Demand Pattern

Querying /api/stats/by-hour (GROUP BY HOUR(tpep_pickup_datetime), COUNT(*), AVG(fare_amount)) revealed two demand peaks: morning rush 7–9 AM and evening surge 5–8 PM, mirroring NYC commuter patterns. Evening fares are higher due to longer outer-borough return trips. Evening average speed also drops, confirming congestion. Implication: taxi supply should be dynamically scaled around these two windows.

Manhattan Volume vs. Fare Efficiency

The borough_statistics view and /api/locations/top-pickup confirm Manhattan dominates pickup volume. However, the fare_per_mile derived feature (fare_amount / trip_distance) shows outer-borough and airport trips (JFK/EWR rate codes) generate significantly higher revenue per mile than short intra-Manhattan trips. Implication: high pickup volume does not equal proportional driver earnings, airport routes are far more economically efficient per mile.

Tip Behaviour Linked to Payment Method

The /api/tips/distribution endpoint groups tip_percentage brackets by payment_type. Cash trips cluster at 0% tip; card trips cluster at 16–20%. This reflects the psychological friction of cash tipping versus prompted card terminal percentages — not a data reporting artifact. Implication: rising cashless adoption will increase driver tip income independently of fare changes.

Frontend Dashboard — Design and Implementation

Structure & Dependencies (index.html)

Plain HTML/CSS/JavaScript ES modules, no build step. Two-panel Flexbox layout: 280px sidebar + flex-1 main area. CDN dependencies: Leaflet.js (map), Chart.js (bar chart), Inter font. Sidebar filter controls: Borough dropdown (id: boroughFilter, populated dynamically via fetchBoroughs()), Hour dropdown (id: hourFilter), Fare Range inputs (ids: minFare, maxFare), and Apply Filters button (id: applyFilters). Main area: three stat tiles (totalTrips, avgFare, totalRevenue initialised as 'Loading...'), four tab buttons (Overview, Temporal Patterns, Geographic Analysis, Trip Explorer), trips table (id: tripsTable, columns: Date/Time, Pickup, Dropoff, Distance, Fare), Chart.js canvas (id: boroughChart), and Leaflet map div (id: map, 400px height).

Visual Design (style.css)

Three-level dark colour hierarchy: base #0e1a2b (deep navy body), #111f33 (sidebar), #16263f (cards and inactive tabs). Single accent #3b82f6 (blue) used only for the active tab, Apply Filters button, and bar chart fill. White text with 60–70% opacity for labels and headers creates typographic hierarchy without additional colours. Pill-shaped tabs (border-radius: 20px). Table row separators use rgba(255,255,255,0.05) barely visible, avoiding grid overload on the dark background.

Data Layer (api.js)

Six async fetch functions at BASE_URL (localhost:5000): fetchOverview() for aggregate stats; fetchTrips(params) using URLSearchParams for filtered records; fetchByBorough() for borough aggregations; fetchZonesGeoJSON() returning a GeoJSON FeatureCollection for Leaflet; fetchTopPickups(limit) for ranked pickup zones; fetchBoroughs() which deduplicates zone records via a JavaScript Set to dynamically populate the Borough dropdown at runtime.

Orchestration & Rendering (app.js + charts.js)

app.js registers initDashboard() on DOMContentLoaded, sequentially calling: renderStats(overview), renderTripsTable(top 20 trips sorted by fare_amount DESC), renderBoroughChart(borough stats), renderMap(geojson), and fetchTopPickups(10) logged to console for a future visualisation. Entire sequence wrapped in try/catch for graceful error handling. charts.js holds module-level map and boroughChart variables boroughChart.destroy() is called before each re-render to prevent Chart.js canvas conflicts; Leaflet skips re-initialisation on subsequent calls and instead clears prior GeoJSON layers by removing any layer carrying a feature property before rendering updated data with blue fill (#3498db, 50% opacity) and zone/borough popup tooltips.

To prevent redundant tab re-initialisation and repeated data loading, a module-level state tracker is implemented using: (**const loadedTabs = new Set();**)

This Set stores the names of dashboard tabs that have already been initialised, ensuring each tab's data and rendering logic executes only once per session.

Within the Geographic Analysis tab logic, the guard clause:
(if (loadedTabs.has("geographic")) return;

loadedTabs.add("geographic");) prevents re-rendering of the Leaflet map and associated API calls if the tab has already been loaded, avoiding duplicate map layers and unnecessary network requests.

Similarly, within the Trip Explorer tab logic:
(**if (loadedTabs.has("explorer")) return;**
loadedTabs.add("explorer");)
ensures the trip exploration dataset and table rendering logic execute only once, preventing repeated fetch operations and DOM reflows.

5. Reflection and Future Work

5.1 Technical Challenges

- Pipeline phase ordering: outlier detection before type standardisation caused type mismatch errors in early iterations resolved by enforcing an explicit phase sequence inside clean_data() so type conversion always follows validation.
- Fare reconciliation: optional surcharge columns (e.g., congestion_surcharge) were not consistently present across records handled by dynamically detecting existing columns and zero-filling missing ones before summing fare components.
- GeoJSON file size: resolved by serving from disk on demand rather than caching in memory, keeping the Flask API fully stateless.
- Leaflet layer accumulation: without explicit removal, GeoJSON polygons stacked on each filter re-render fixed by iterating all map layers and removing any carrying a feature property before adding new data.
- Chart.js canvas conflict: 'Canvas is already in use' error on re-render resolved by storing the chart instance in a module-level boroughChart variable and calling .destroy() before recreating.
- Schema versioning: parallel backend/frontend development caused API breakages when the schema changed resolved by establishing a shared schema definition file as the single source of truth.

5.2 Future Improvements

- Wire the four dashboard tab buttons to distinct data views and corresponding API endpoints (currently static).
- Render fetchTopPickups() results as a ranked list or choropleth overlay on the Leaflet map — currently only logged to the console.
- Add Redis caching for expensive aggregation queries (borough stats, hourly patterns) to reduce database load at scale.
- Implement streaming ingestion via Apache Kafka to replace the current batch pipeline and process new TLC records in real time.
- Migrate to PostgreSQL + PostGIS for server-side spatial queries and zone-level heatmap aggregation without client-side GeoJSON rendering.
- Containerise backend with Docker and deploy to AWS or GCP for public access and production readiness.
- Integrate fare prediction and trip demand forecasting ML models as additional Flask API endpoints.