NYC Taxi Trip Explorer - Technical Report

1. Problem Framing and Dataset Analysis

Dataset Context

The application uses the official NYC TLC (Taxi & Limousine Commission) data in three parts: (1) yellow_tripdata (fact table) - trip-level records in CSV or Parquet with timestamps, trip_distance, PULocationID, DOLocationID, fare_amount, total_amount, and related fields; (2) taxi_zone_lookup.csv (dimension) - maps LocationID to Borough, Zone, and service_zone; (3) taxi_zones (spatial metadata) - a shapefile of zone polygons. Trips are linked to zones via PULocationID and DOLocationID; zone centroids are computed from the shapefile for heatmaps and clustering.

Data Challenges Identified

• Zone resolution: PULocationID and DOLocationID must exist in taxi_zone_lookup; unknown or missing IDs are excluded.
• Outliers: Trip duration constrained to 60 seconds–24 hours; trip_distance 0–500 miles; fare and total_amount within valid ranges.
• Data quality: Invalid or missing timestamps, passenger counts outside 0–9, and logical anomalies (e.g., negative fares) are rejected.
• Scale: Large trip files are processed with streaming and batched inserts; zone lookup is loaded once into memory.
• Transparency: All excluded records are written to logs/excluded_records.log with a reason code.

Unexpected Observation

The taxi_zones shapefile uses State Plane (New York Long Island, US Feet). Reprojecting to WGS84 with proj4 was required so zone centroids could be used correctly in Leaflet. Using these centroids for the heatmap and K-means gave clearer, zone-based patterns than raw coordinates and aligned the app with TLC's official spatial metadata.

2. System Architecture and Design Decisions

Architecture Overview

Frontend: HTML/CSS/JS
Backend: Express.js
Database: PostgreSQL

Data flow: the browser sends HTTP GET requests to /api/stats, /api/trips, /api/heatmap, /api/clusters, and /api/zones. The server runs parameterized SQL against PostgreSQL and returns JSON. For clusters, the server fetches trip points (zone centroid lat/lon and duration), runs the custom K-means in memory, and returns the cluster lists.

Technology Stack Justification

- Node.js / Express.js: Single language for backend and scripts, good support for CSV/Parquet and streaming, efficient I/O for database and file handling.
- PostgreSQL: ACID compliance, foreign keys (trips → zones), and indexes on datetime, duration, location IDs, borough, and trip_type for fast filtering and aggregation.
- HTML / CSS / JavaScript: No frontend framework; broad compatibility and straightforward deployment.
- Plotly.js: Charts (pie, bar, scatter, histogram) with minimal code; Leaflet: Tile map and zone-based markers for heatmap and clusters.

Design Trade-offs

• Memory vs. performance: Streaming read of yellow_tripdata and batched inserts avoid loading the full file into memory; zone lookup is kept in memory for fast joins.
• Simplicity vs. features: One-off import scripts (setupDatabase, importZones, importData) with no scheduler; dashboard focuses on overview, patterns, heatmap, clusters, and insights.
• Zone-based vs. coordinate-based: Trips have no lat/lon; we use zone centroids from the shapefile, so the heatmap and clusters reflect TLC zones and stay consistent with official geography.

3. Algorithmic Logic and Data Structures

Custom K-Means Clustering Implementation

Problem: Group similar taxi trips by pickup location (zone) and trip duration to reveal mobility patterns and hotspots, without using built-in clustering libraries.

Approach: Manual K-means in the backend (server.js). Each point is (zone centroid lat, zone centroid lon, trip_duration_sec). Distance is Euclidean in this 3D space with duration scaled (e.g., divided by 1000) so units are comparable.

Pseudo-code:

```
FUNCTION kMeans(data, k, maxIterations):
  IF data.length == 0 OR k <= 0:
    RETURN empty array
  centroids = initializeCentroids(data, k)   // k random points from data
  FOR iteration = 1 TO maxIterations:
    clusters = createEmptyClusters(k)
    FOR each point IN data:
      distances = calculateDistances(point, centroids)  // custom distance, no libs
      nearestCluster = indexOfMinimum(distances)
```

```
        clusters[nearestCluster].add(point)
    newCentroids = for each cluster: (avg lat, avg lon, avg duration)
    IF max |centroids[i] - newCentroids[i]| < threshold:
        BREAK
    centroids = newCentroids
RETURN filterNonEmptyClusters(clusters)
```

Time complexity: O(n × k × i) where n = number of points, k = number of clusters, i = iterations.
Space complexity: O(n + k) for the point list and centroid list.

Details: Empty clusters are reinitialized with a random data point. Distance uses (lat, lon, duration/1000) with no external math libraries. Convergence is when each centroid moves by less than a fixed threshold or max iterations is reached. The frontend receives an array of clusters (each cluster is an array of points) and plots them by lat/lon with Plotly.

4. Insights and Interpretation

Insight 1: Trip distribution by borough

Derivation: Aggregate query on trips grouped by pickup_borough (from zone lookup), counting trips per borough.
Visualization: Pie chart on the Overview tab (trip count and percentage per borough).
Interpretation: Manhattan typically dominates yellow taxi pickups because of the central business district and tourism. Other boroughs show smaller shares; EWR (Newark Airport) appears when present in the zone lookup. The distribution reflects where demand is concentrated and supports planning and policy discussions.

Insight 2: Hourly trip patterns

Derivation: Query grouping by hour_of_day (derived from pickup timestamp, with hour taken from the CSV string to avoid timezone issues).

Visualization: Bar chart of the number of trips per hour (0–23) on the Overview tab.
Interpretation: Peaks in the morning and evening correspond to commute times; midday and late night show different demand. This highlights when the system is under the most load and when pricing or supply might be adjusted.

Insight 3: Speed and distance by trip type

Derivation: Filtering trips by borough and trip type (Within Borough vs Cross Borough); scatter or bar charts of speed_kmh, distance_km, and duration from the trips API.
Visualization: Patterns tab (duration histogram, speed–distance scatter) and Insights tab (e.g., average duration by borough, rush-hour comparison).
Interpretation: Shorter trips often have more variable speed (congestion, signals); longer or cross-borough trips can show more stable speeds. Borough-level averages reveal where trips tend to be longer or slower, supporting discussions of congestion and network performance.

5. Reflection and Future Work

Technical challenges

• Timezone and hour: The initial hourly chart was skewed because hour_of_day used the server's local time. Fix: extract the hour from the CSV datetime string so it matches TLC's Eastern time regardless of the server timezone.
• Spatial data: Shapefile in State Plane required reprojection to WGS84 for the map; proj4 was used so zone centroids plot correctly in Leaflet.
• Data volume: Large CSV/Parquet files are handled by streaming and batch inserts; an exclusion log keeps cleaning decisions auditable.

Lessons learned

• Using official TLC zone IDs and spatial metadata (lookup + shapefile) keeps the app aligned with published geography and improves interpretability.

• Storing a few derived fields (duration, speed_kmh, fare_per_km, tip_rate, trip_type, hour_of_day) avoids repeated computation and keeps API responses fast.
• A single custom K-means in the backend, with no clustering libraries, met the assignment requirement and made the grouping logic easy to explain and modify.

Future enhancements

• Real-time or incremental data: WebSocket or polling for new trip data; incremental import instead of full reload.
• Predictive models: Use historical trips and time of day to predict duration or demand by zone.
• Mobile and accessibility: Further responsive and a11y improvements for small screens and assistive technologies.
• Time range and filters: Date range picker and filters by payment type or rate code for deeper analysis.

Production considerations

• Connection pooling (e.g., pg pool) and timeouts to handle concurrent users.
• Caching for heavy or repeated queries (e.g,. stats, hourly aggregates).
• API rate limiting and optional authentication for public deployment.
• Environment-based config for database and ports; no secrets in the repo.

Technical specifications

• Data: Official TLC yellow_tripdata + taxi_zone_lookup + taxi_zones; trips table with FKs to zones; derived fields (trip_duration_sec, speed_kmh, fare_per_km, tip_rate, trip_type, hour_of_day, etc.).
• Processing: Streaming import with batch inserts; exclusion log for invalid or suspicious records.
• API: REST GET endpoints; typical response times under a few hundred milliseconds with indexed queries.
• Frontend: ES6 JavaScript, Plotly.js, Leaflet; runs in modern browsers.