NYC Taxi Mobility System — Technical Documentation

1. Problem Framing and Dataset Analysis

Dataset Overview

This project uses the New York City Taxi Trip dataset, a public dataset containing millions of trips recorded by the NYC Taxi and Limousine Commission (TLC). Each record includes details such as pickup and drop-off times, passenger count, trip distance, duration, and fare amounts.
Our goal was to transform this raw, inconsistent data into meaningful insights about urban mobility patterns.

Data Challenges

While exploring the raw dataset, we identified several data quality issues:

- **Missing and invalid fields:** Many trips lacked coordinates or fare amounts.
- **Outliers:** Some trips reported unrealistic speeds (>150 km/h) or extremely short/long durations.
- **Inconsistent formats:** Datetime and numeric fields appeared in mixed formats (strings, timestamps).
- **Duplicates:** Overlapping trip IDs existed due to merges from multiple data files.

Data Cleaning and Assumptions

We developed a **Python-based cleaning pipeline** using Pandas and NumPy to standardize and refine the dataset:

- Removed duplicate trip IDs.
- Converted timestamps to consistent datetime objects.
- Replaced invalid numeric values (negative, zero, or null) with NaN.
- Filtered out trips with unrealistic speed (>150 km/h) or zero distance.
- Normalized distances to **kilometers** and durations to **minutes**.

Assumptions:

- Trips missing key information (e.g., duration or coordinates) were excluded.
- The reasonable average speed range was defined as **0–150 km/h**.
- Missing fares were ignored in derived metrics to prevent bias.

Unexpected Observation

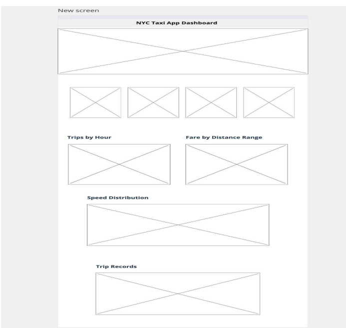
During cleaning, we found that many short trips (<2 minutes) had abnormally high fares.
This anomaly inspired us to create a derived metric — fare_per_km — and use it in our backend anomaly detection algorithm to highlight potential pricing irregularities.

2. System Architecture and Design Decisions

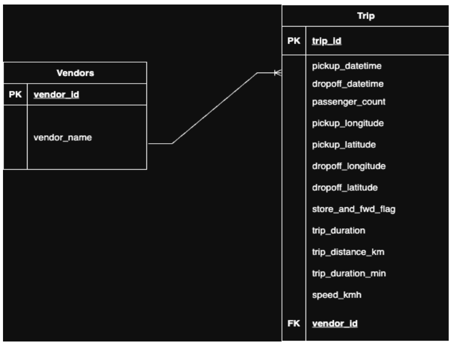
Architecture Overview

The system follows a three-tier architecture:

Frontend (React)



Backend (Flask REST API)
↓
Database (MySQL)



- **Frontend:** Interactive dashboard for visualizing trip patterns, speeds, and anomalies.
- **Backend:** Flask API that handles data cleaning, anomaly detection, and queries (/api/trips, /api/insights).
- **Database:** MySQL stores cleaned and feature-engineered trip data.

Technology Choices and Justifications

Component	Technology	Reason
Backend	Flask (Python)	Lightweight, integrates well with data libraries like Pandas.
Database	MySQL	Reliable for large structured datasets and supports analytical queries.
Frontend	React	Enables responsive, dynamic visualizations.

Schema Structure

Table: trips

Column	Type	Description
id	VARCHAR(255)	Unique trip ID
pickup_datetime	DATETIME	Start time
dropoff_datetime	DATETIME	End time
passenger_count	INT	Number of passengers
trip_distance_km	DOUBLE	Distance in kilometers
trip_duration_min	DOUBLE	Duration in minutes
speed_kmh	DOUBLE	Derived feature (distance/duration)
fare_per_km	DOUBLE	Derived feature for efficiency analysis

Design Trade-offs

- MySQL vs MongoDB: We chose MySQL for relational consistency over flexibility.
- Batch vs Real-time Processing: Batch cleaning was simpler for the dataset scale.
- Server vs Client Processing: Performed feature engineering on the backend to reduce browser computation.

3. Algorithmic Logic and Data Structures
Overview

We implemented a custom anomaly detection algorithm in JavaScript to identify suspicious taxi trips — including extremely high fares, impossible speeds, or illogical fare-per-distance ratios.
This was done without using built-in methods such as `sort()` or `filter()`, emphasizing manual algorithmic design.

Approach

1. Extract numerical values (e.g., fares or speeds) from the dataset.
2. Manually sort the data using Bubble Sort.
3. Compute Q1 (25th percentile) and Q3 (75th percentile).
4. Calculate IQR = Q3 - Q1 and determine normal bounds:
   [Q1 - 1.5 × IQR, Q3 + 1.5 × IQR].
5. Flag any record outside this range as an anomaly.
6. Detect additional anomalies in fare_per_km and speed_kmh.

Pseudocode

```
function detectOutliers(data, field):
    values = extract field values
    sorted = bubbleSort(values)
    n = length(sorted)
    Q1 = sorted[floor(n * 0.25)]
    Q3 = sorted[floor(n * 0.75)]
    IQR = Q3 - Q1
    lower = Q1 - 1.5 * IQR
    upper = Q3 + 1.5 * IQR
    outliers = []
    for each record in data:
        if record[field] < lower or record[field] > upper:
            outliers.append(record)
    return outliers
function bubbleSort(arr):
    for i from 0 to n-1:
        for j from 0 to n-i-1:
            if arr[j] > arr[j+1]:
                swap(arr[j], arr[j+1])
    return arr
```

Complexity Analysis

Step	Complexity	Description
Sorting (Bubble Sort)	$O(n^2)$	Manual comparison of all pairs
Outlier Detection	$O(n)$	Linear scan through dataset
Overall	$O(n^2)$	Adequate for moderate dataset sizes

Impact

This algorithm improved data integrity by removing inconsistent records before analytics.
It powers the /insights API, providing users with more accurate and trustworthy statistics.

4. Insights and Interpretation

Insight 1 — Peak Travel Hours

Query:

```
SELECT HOUR(pickup_datetime) AS hour, COUNT(*) AS trip_count
FROM trips GROUP BY hour ORDER BY hour;
```

Observation: Trip counts spike between 7–9 AM and 5–7 PM, mirroring NYC's commute hours.
Interpretation: Reflects daily travel demand — useful for fleet scheduling and surge pricing strategies.

Insight 2 — Speed vs Passenger Count
Query:
```
SELECT passenger_count, AVG(speed_kmh) AS avg_speed
FROM trips GROUP BY passenger_count;
```
Observation: 1–2 passenger trips have higher average speeds than group trips.
Interpretation: Larger groups tend to travel shorter inner-city routes with heavier traffic.
Insight 3 — Fare Efficiency
Visualization: Scatter plot of fare_per_km vs trip_distance_km.
Observation: Short trips (<2 km) show extremely high fare_per_km.
Interpretation: Indicates base fare influence or surge pricing, revealing inefficiencies in pricing fairness.

5. Reflection and Future Work

Technical Challenges

- Processing large CSVs that exceeded in-memory capacity.
- Managing cross-module imports between backend folders.
- Maintaining database consistency during batch inserts.

Team Challenges

- Merge conflicts from parallel Git branch work.
- Aligning frontend visualizations with backend data structure.

Future Improvements

- Implement real-time streaming for live trip monitoring.
- Add geospatial clustering (K-Means) for pickup hotspot analysis.
- Deploy with Docker + AWS RDS for scalability.
- Integrate predictive models to estimate fares or durations.

Summary

The NYC Taxi Mobility System demonstrates a full data-driven pipeline — from cleaning raw TLC data to generating actionable insights through a custom-built API and visualization dashboard.
Our emphasis on algorithmic transparency and data integrity ensures that every analytical result reflects both technical rigor and urban mobility relevance.