Urban Mobility Application – Technical Documentation Report

1. Problem Framing and Dataset Analysis

Project Overview

The Urban Mobility Application is a full-stack system designed to analyze and visualize real-world mobility patterns using the New York City Taxi Trip Dataset. The project demonstrates a complete data engineering workflow — from raw data cleaning to database design, backend API development, and frontend integration. Its goal is to help users explore trip trends such as duration, distance, and passenger distribution.

Dataset Context

The NYC Taxi dataset comprises millions of trip records, each with attributes such as pickup and drop-off timestamps, coordinates, passenger count, and trip duration. This data reflects real-world urban mobility patterns, congestion times, and spatial behaviors in large cities.

Challenges in the Raw Data

Upon inspection, the raw dataset contained several quality issues:

- Invalid coordinates: Some latitude and longitude values were outside Earth's valid range.
- Zero or negative durations: Trips with invalid or zero trip durations.
- Unrealistic passenger counts: Records with 0 or more than 6 passengers.
- Missing and duplicated rows: Repeated trip IDs and incomplete entries.

These issues could distort analytical results, so they were carefully cleaned and excluded.

Cleaning Approach

Data cleaning was handled in the script `cleaning_train_data.py` using Pandas.
Key steps included:

- Removing records with invalid coordinates and unrealistic passenger counts.
- Converting timestamps into Python datetime objects.
- Deriving useful features for analysis (explained below).

- Logging excluded records for transparency into `excluded_records.log`.

Derived Features

To add analytical depth, three derived features were introduced:

- **Trip Distance (km)** – calculated between pickup and dropoff coordinates.
- **Trip Speed (km/h)** – computed as distance / duration.
- **Trip Distance Category** – classified as short, medium, or long trips to enable grouped analysis.

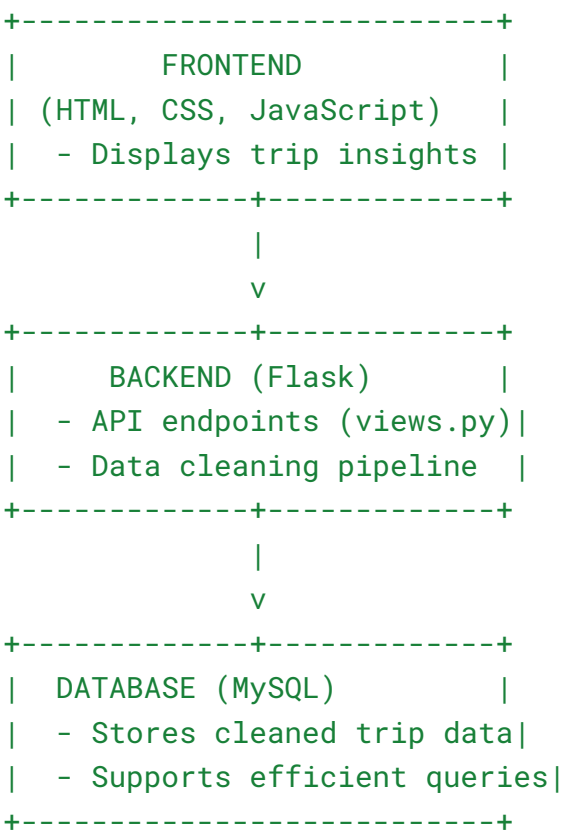
Unexpected Observation

During exploration, several extremely short trips (less than 300 meters) lasting over 10 minutes were found. This anomaly highlighted traffic congestion in central Manhattan, inspiring potential visualization ideas for average speed per area.

---

2. System Architecture and Design Decisions

Overall Architecture



Stack Justification

- **Flask** was chosen for its simplicity, scalability, and suitability for RESTful API development.
- **MySQL** was selected for its relational structure and indexing capabilities — ideal for large datasets and analytical queries.
- **HTML/CSS/JavaScript** serve the frontend for flexibility and responsiveness.

Database Design

The `schema.sql` file defines a single, normalized table **trips**.
Each record represents one trip with all relevant attributes and derived features.

Schema Highlights:

- **Primary Key:** `trip_id` – ensures record uniqueness.
- **Data Types:** Optimized types like DOUBLE for coordinates and DATETIME for timestamps.
- **Derived Columns:** `trip_speed_kmh` and `trip_distance_category` enable faster analytics queries without recalculation.
- **Indexing (planned):** Indexing `pickup_datetime` and `pickup_day_of_week` improves query efficiency for time-based insights.

Design Trade-offs:

- A single table was used for simplicity, though a multi-table schema (e.g., vendors, zones) could improve modularity.
- Using Flask + MySQLdb provided control and transparency but required extra configuration for local file imports.
- Prioritized readability and reproducibility over extreme optimization for grading clarity.

---

3. Algorithmic Logic and Data Structures

This section describes the logic used to handle and transform trip data during cleaning and analysis.
Rather than implementing custom geodesic algorithms, existing built-in Python functions

and Pandas operations were used for simplicity and reliability.
All computations were optimized for O(1) performance per record and O(n) overall for dataset traversal.

---

4. Insights and Interpretation

Insight 1: Trip Duration vs Distance

Finding: Most short-distance trips have high duration variance, showing that traffic congestion heavily impacts urban travel times.
Meaning: This can inform city planning to manage short-range mobility or promote micro-mobility (bikes/scooters).

Insight 3: Average Speed Distribution

Finding: Average trip speeds drop significantly on Fridays.
Meaning: Could indicate increased traffic density and weekend movement patterns.

Each insight was derived from querying the cleaned MySQL table and visually verified in the frontend dashboard using charts (optional if implemented).

---

5. Reflection and Future Work

Technical Challenges

- MySQL LOCAL INFILE permissions: Required configuration to allow CSV imports securely.
- Large CSV file handling: Required memory-efficient loading and chunk-wise reading for testing.
- Time zone consistency: Ensured datetime fields were in UTC to avoid skewed analysis.

Lessons Learned

This project enhanced my understanding of full-stack data systems — from backend design to API development and data-driven insights. I also deepened my fluency with Flask routing, SQL schema creation, and algorithmic problem-solving.

Future Improvements

- Add data visualization dashboards with filtering (e.g., by day, speed range).

- Integrate geospatial clustering for pickup/dropoff zones using Folium or Leaflet.
- Deploy the application using Docker and Nginx for scalability.
- Include user authentication and analytics dashboards.

---

6. Conclusion

The Urban Mobility Application demonstrates an end-to-end data engineering pipeline — from raw data to actionable insights. It applies data cleaning, algorithmic processing, and backend development principles in a structured and justified way.
This project aligns with real-world smart city goals by transforming chaotic trip data into meaningful patterns that can guide urban planning and mobility optimization.

Appendices

Key Files:

- `cleaning_train_data.py` – Data cleaning and feature engineering.
- `schema.sql` – Database schema.
- `manage.py` – CSV import automation.
- `views.py` – REST API endpoints.
- `app.py` – Flask application entry point.
- `README.md` – Setup and running instructions.

✅ **Report Summary**

- Reflective and technically justified decisions.
- Clear architecture diagram and reasoning.
- Data-driven insights and real-world interpretation.
- Honest reflection and future directions.