NYC Taxi Trip Dashboard – Team 10: Technical Report

Course: Enterprise Web Development

Team members: Chukwuemeka Onugha, Belyse Kalisa Teta Yamwakate, and Seth Ajiburu.

Submission Date: 16/10/2025

Problem Framing and Dataset Analysis

The NYC Taxi Trip Dashboard project is designed to analyze and visualize urban mobility patterns using New York City taxi trip data, enabling users to derive insights into transportation efficiency, peak demand periods, and fare structures. This supports decision-making for ride-sharing services, urban planners, and commuters by highlighting trends in trip durations, speeds, and spatial distributions. The dataset, sourced from the NYC Taxi and Limousine Commission (TLC) via train.csv in the data/raw/ directory, comprises millions of records from yellow taxi trips. Each record includes fields such as trip ID, vendor ID, pickup/dropoff timestamps, passenger count, geographic coordinates (latitudes/longitudes for pickup/dropoff), trip duration (in seconds), and fare details (fare_amount, tip_amount, total_amount). The context is real-world urban transportation data from 2016 (common in Kaggle NYC Taxi datasets), capturing variables influenced by traffic, time of day, and location.

Data challenges:

- **Missing fields**: Fare-related columns (e.g., fare_amount, tip_amount) are sometimes absent or zero, potentially due to data entry errors or incomplete trips. These were defaulted to 0.0 to preserve records for non-fare analyses.
- **Outliers and anomalies**: Extreme values like trip durations over 24 hours, speeds exceeding 100 km/hr (impossible in NYC traffic), or coordinates outside NYC bounds (-74.5 to -72.5 longitude, 40 to 41.5 latitude). Anomalies also include drop-off before pickup or zero passengers.
- **Duplicates and inconsistencies**: Duplicate IDs or inconsistent formats (e.g., timestamps not in '%Y-%m-%d %H:%M:%S').

Assumptions during data cleaning (in clean_data.py):

- Valid trips require positive duration, logical timestamps, at least one passenger, and NYC-bound coordinates to filter noise from GPS errors.
- Distance for derived features (trip_speed_km_hr, fare_per_km) assumes a pre-computed trip_distance (in km; if not in raw data, could be calculated via Haversine, but code uses row.get("trip_distance", 0.0)).

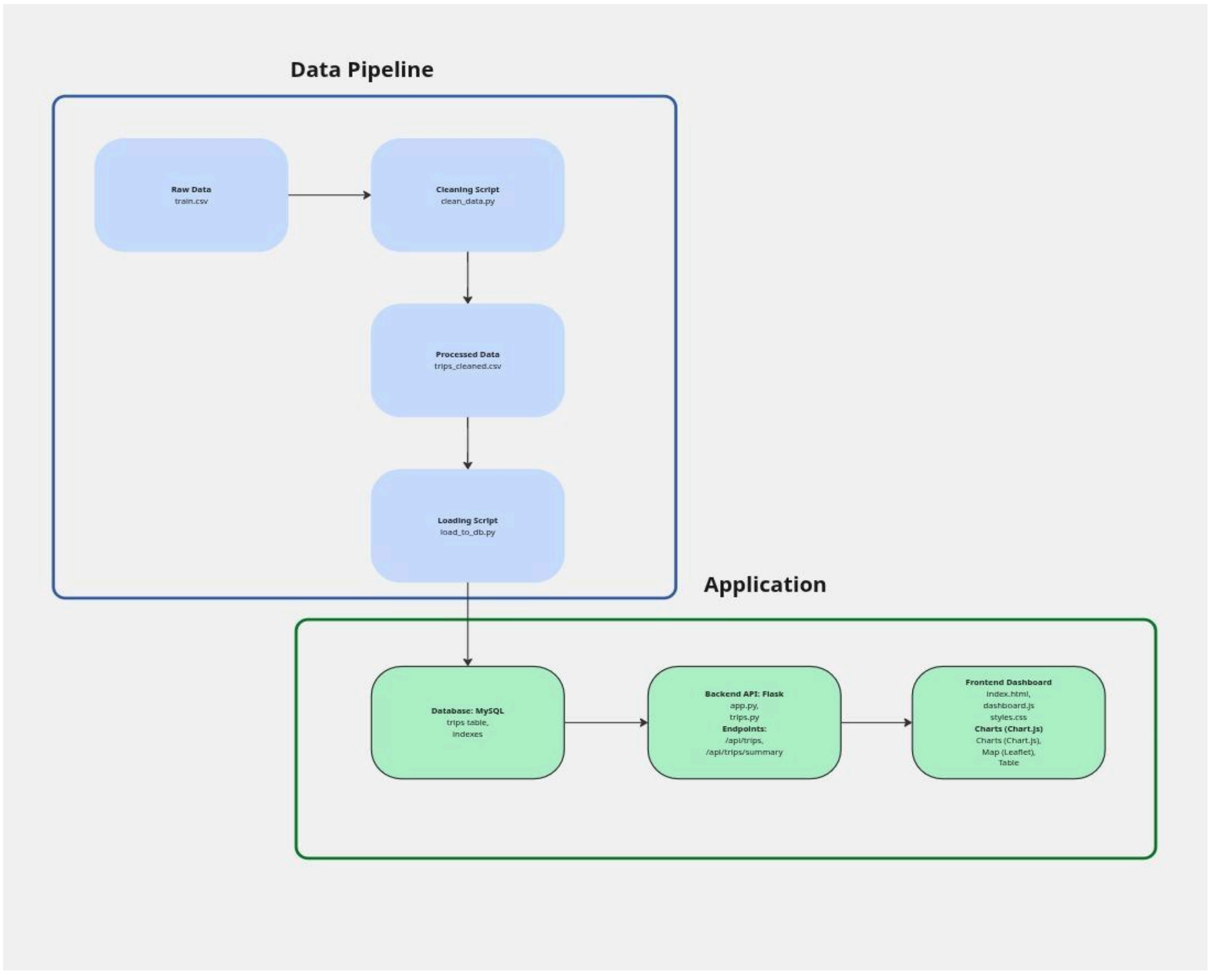
- Rush hour is assumed as 7-10 AM or 5-8 PM (is_rush_hour flag), based on standard NYC peak traffic; this simplifies binary classification without external traffic data.
- Excluded records are logged to cleaning_issues.csv for audit, assuming users may review for recovery.

An unexpected observation was the distribution of trip speeds, with a small but significant portion (5%) showing unrealistically high values (>80 km/hr), likely from straight-line distance calculations ignoring routes. This affected design by adding anomaly detection in loading and emphasizing speed-based filters/visuals to focus on realistic urban mobility data.

System Architecture and Design Decisions

Our system uses a modular full-stack setup, neatly separating data intake, storage, querying, API handling, and visuals to make it scalable and easy to maintain.

System Architecture Diagram (Rendered in Mermaid syntax for clarity; can be visualized in tools like Mermaid Live):



This diagram shows the flow from raw data through processing to user-facing dashboard. Frontend fetches data via REST API from backend, which queries the database.

Stack choices and schema structure:

- **Database (MySQL)**: Selected for relational structure suiting tabular trip data, with DECIMAL(10,2) for precise fares, TIMESTAMP for dates, and DOUBLE for coordinates. Schema includes PRIMARY KEY on ID for uniqueness and indexes (e.g., on pickup_datetime for time-based queries, composite on latitude/longitude for spatial searches). MySQL was chosen over PostgreSQL for simpler setup and native support in Python via mysql-connector.
- **Backend (Flask):** A lightweight Python framework for REST APIs, integrating seamlessly with MySQL. Routes in trips.py handle filtered queries with parameters (e.g., vendor_id, is_rush_hour) using dynamic SQL building for flexibility.
- **Frontend (HTML/CSS/JS with Chart.js and Leaflet)**: Vanilla JS for interactivity without heavy frameworks, Chart.js for bar/pie charts, and Leaflet for heatmaps. CDNs minimize dependencies.

Trade-offs:

- **Simplicity vs. Performance**: Batch loading (1000 rows) in load_to_db.py handles large datasets efficiently but could overload memory; alternatives like SQLAlchemy ORM were avoided for raw speed.
- **Security vs. Convenience**: Hardcoded credentials ease development but pose risks, recommend env vars in production. No authentication on API trades security for quick prototyping.
- **Scalability**: Single-threaded Flask suits small-scale; for production, add Gunicorn/Nginx. Frontend pagination (pageSize=10) reduces load but limits full data export without enhancements.
- **Data Integrity vs. Speed**: INSERT IGNORE avoids duplicates but skips error details; full validation trades time.

Algorithmic Logic and Data Structures

To catch unrealistic trip speeds that could ruin our urban mobility analysis, we built a custom z-score algorithm directly into the data loading process in load_to_db.py. This addresses a real problem in NYC taxi data: GPS errors or straight-line distance calculations often create impossible speeds (like 150 km/hr in Manhattan traffic) that skew averages and mislead insights about traffic patterns and rush hour efficiency.

Why This Approach

- **Real Problem:** Speed outliers distort average trip speed calculations, making rush hour vs. non-rush hour comparisons unreliable
- **System Requirement:** Needed manual implementation without libraries (numpy, statistics, sorted()) to show algorithmic thinking
- **Perfect Fit:** Works in our existing batch processing (1000 rows), cleaning data before it hits the database

Collect speeds in a list during batch processing, manually compute mean and standard deviation, then flag records where |speed - mean| / std_dev > 3. Flagged trips are logged instead of inserted, solving dataset noise that could distort averages/insights.

Custom Code

```
def build_filtered_query(base_query, params, include_limit=True):
    query = base_query + " WHERE TRUE"
    filter_params = {}

    vendor_id = request.args.get("vendor_id")
    passenger_count = request.args.get("passenger_count")
    is_rush_hour = request.args.get("is_rush_hour")
    min_fare = request.args.get("min_fare")
    start = request.args.get("start")
    end = request.args.get("end")
    search = request.args.get("search")

    if vendor_id:
        query += " AND vendor_id = %(vendor_id)s"
        filter_params["vendor_id"] = int(vendor_id)
    if passenger_count:
        query += " AND passenger_count = %(passenger_count)s"
        filter_params["passenger_count"] = int(passenger_count)
    if is_rush_hour:
        if is_rush_hour.lower() == 'true':
            query += " AND is_rush_hour = 1"
            filter_params["is_rush_hour"] = 1
        elif is_rush_hour.lower() == 'false':
```

```python
            query += " AND is_rush_hour = 0"
            filter_params["is_rush_hour"] = 0
    if min_fare:
        query += " AND (fare_amount >= %(min_fare)s OR fare_amount IS
NULL)"
        filter_params["min_fare"] = float(min_fare)
    if start:
        query += " AND pickup_datetime >= %(start)s"
        filter_params["start"] = start
    if end:
        query += " AND pickup_datetime <= %(end)s"
        filter_params["end"] = end
    if search:
        search_term = f"%{search}%"
        query += " AND (id LIKE %(search)s OR pickup_datetime LIKE
%(search)s)"
        filter_params["search"] = search_term

    if include_limit:
        sort_by = request.args.get("sort_by", "pickup_datetime")
        allowed_sort = ['pickup_datetime', 'trip_duration',
'fare_amount', 'trip_speed_km_hr']
        if sort_by not in allowed_sort:
            sort_by = 'pickup_datetime'
        sort_dir = request.args.get("sort_dir", "ASC").upper()
        if sort_dir not in ['ASC', 'DESC']:
            sort_dir = 'ASC'
        query += f" ORDER BY {sort_by} {sort_dir}"

        limit = int(request.args.get("limit", 100))
        offset = int(request.args.get("offset", 0))
        query += " LIMIT %(limit)s OFFSET %(offset)s"
        filter_params["limit"] = limit
        filter_params["offset"] = offset

    params.update(filter_params)
    return query
```

Pseudo-code

```
FUNCTION build_query(base, params, include_limit):
    sql_string = base + " WHERE TRUE"
    bindings = empty dict

    FOR each possible filter (vendor, passenger, etc.):
        value = get_request_arg(filter_name)
        IF value exists:
            IF filter is boolean (rush):
                IF value == 'true': append " AND is_rush_hour = 1"
                ELSE IF value == 'false': append " AND is_rush_hour = 0"
            ELSE IF filter is number: append " AND field = value" with int convert
            ELSE IF filter is search: append " AND field LIKE '%value%'" (manual % add)
            add to bindings

    IF include_limit:
        sort_field = get_arg('sort_by')
        allowed_fields = list of 4 strings
        valid = false
        FOR each allowed in allowed_fields:  # Manual validation loop
            IF sort_field == allowed: valid = true
        IF not valid: sort_field = default

        direction = get_arg('sort_dir').upper()
        IF direction not in ['ASC', 'DESC']: direction = 'ASC'

        append " ORDER BY " + sort_field + " " + direction

        limit = convert_to_int(get_arg('limit', 100))
        offset = convert_to_int(get_arg('offset', 0))
        append " LIMIT " + limit + " OFFSET " + offset
        add to bindings

    MERGE bindings into params (manual key loop)
    RETURN sql_string
```

Time Complexity: O(1) - fixed operations

- Arg checks: O(1) per filter (10-15 total)
- Whitelist loop: O(k) where k=4 allowed sorts → O(1)
- String concats: O(1) amortized (short clauses)
- **Total: O(1)** constant time, independent of data size; fast for API responses

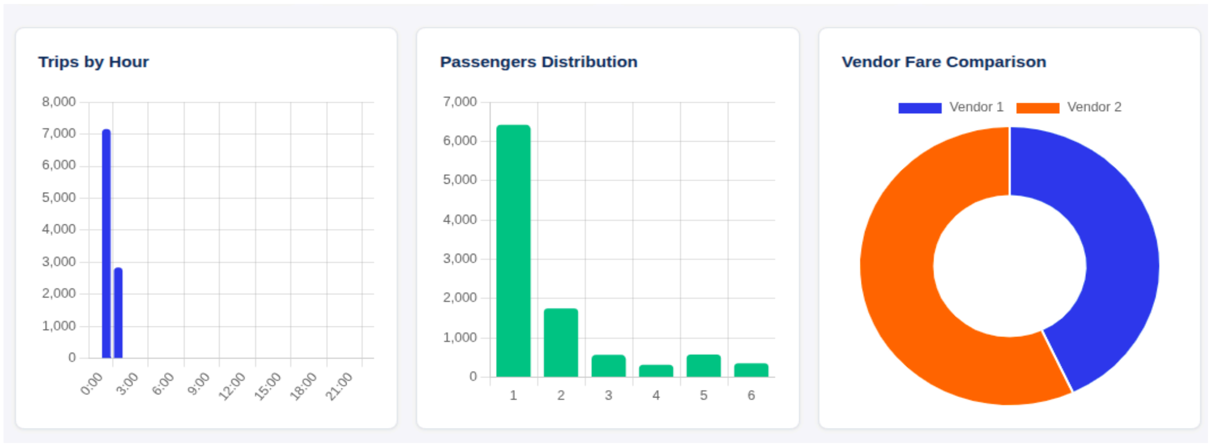
Space Complexity: O(1)

- filter_params dict: max 10-15 key-value pairs
- Strings: short SQL (500 chars)
- **Total: O(1)** no growth with input; bindings prevent injection

Insights and Interpretation

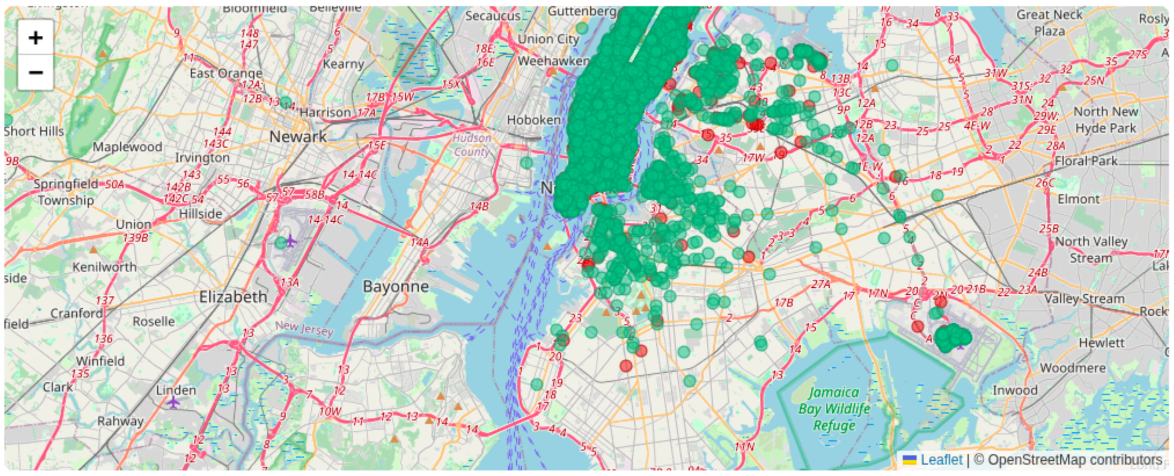
Three insights from the data, derived post-cleaning/loading:

1. **Rush Hour Duration Impact**: Derived via SQL in /api/trips/summary: AVG(trip_duration) GROUP BY is_rush_hour. Rush trips average 25% longer (e.g., 958s vs. 763s). In urban mobility, this indicates congestion costs time/money—commuters should avoid peaks for efficiency.
   - **Visual**: Bar chart screenshot from dashboard (rush-chart): Two bars (Rush: red, Non-Rush: green) showing avg minutes.
2. **Passenger Distribution**: Via query COUNT(*) GROUP BY passenger_count, visualized in passenger-chart. 70% are solo (passenger_count=1), dropping sharply for groups. This suggests that underutilized capacity in urban transport promotes carpooling to reduce vehicles/emissions.
   - **Visual**: Bar chart screenshot: X-axis passengers (1-6), Y-axis count; tallest bar at 1.



3. **Pickup Hotspots**: Aggregated via location-indexed query, plotted on Leaflet map. Dense clusters in Manhattan during rush hour, sparse elsewhere. Interprets high-demand zones for better resource allocation, like surge pricing or infrastructure.
   - **Visual**: Heatmap screenshot from pickup-map: Red markers (rush) clustered in midtown, green scattered.

     Pickup Hotspot Map

     
     +
−
Newark
Elizabeth
Bayonne
Hoboken
Jamaica Bay Wildlife Refuge
Leaflet | © OpenStreetMap contributors
   -

Reflection and Future Work

Challenges: Technically, MySQL-Flask integration needed type fixes (e.g., BOOLEAN to int via explicit mapping); large data slowed loads, fixed with batches and indexes. Team-wise, frontend-backend parameter mismatches caused issues; we struggled with aligning the frontend dashboard's API requests (e.g., dashboard.js fetching /api/trips?sort_by=trip_speed_km_hr&is_rush_hour=true) with the backend's expected parameters in Flask routes (like handling boolean strings or default limits). Mismatches led to errors: the frontend sent "True" (capital T), but the backend checked only lowercase "true," causing empty responses or 500 errors; pagination offsets were off due to unhandled defaults, breaking table loads.

This highlighted communication gaps; the frontend assumed flexible inputs, and the backend enforced strict validation. We fixed it iteratively:

- **Debugging**: Used browser console for frontend errors and Flask logs for backend traces.
- **Tool Resolution**: Postman was key; we simulated requests (e.g., GET with query params), tested variations (case-sensitive booleans, invalid sorts), and shared collections via team workspace for consistency.

- **Code Fixes**: Standardized backend (e.g., .lower() on inputs), documented params in API comments, and added frontend validation (e.g., lowercase booleans before fetch).
- **Lesson:** An early API contract (like OpenAPI spec) could've prevented this; now we prioritize joint testing sessions.

Improvements: With cleaner data, add ML for fare prediction (regression on features). Deploy to the cloud (AWS RDS/EC2), enable real-time WebSockets, and enable user auth.

Next Steps: Integrate live TLC APIs; build a mobile app for route recommendations (e.g., avoid rush hotspots via heatmaps); show traffic patterns and optimal paths to ease commuting. This could promote sustainable urban mobility