1. Problem Framing and Dataset Analysis

The dataset given to us was a trip record of New York taxi trips. The data was separated into the following categories:

- Trip ID
- Vendor ID
- Pickup and drop-off date
- Pick up the latitude and longitude
- Dropoff latitude and longitude
- Store_and_fwd_flag
- Trip duration

Some of the trip records lack proper location information, and the store_and_fwd_flags do not make sense. There were a few outliers, like passenger count exceeding 3 and unusually short distances, and there were a few duplicated records. There was nothing unexpected we observed, but we noticed a certain trend with the lengths of the trips, passenger count, and time of day, so we had to include a graph to show the correlation.

2. System Architecture and Design Decisions

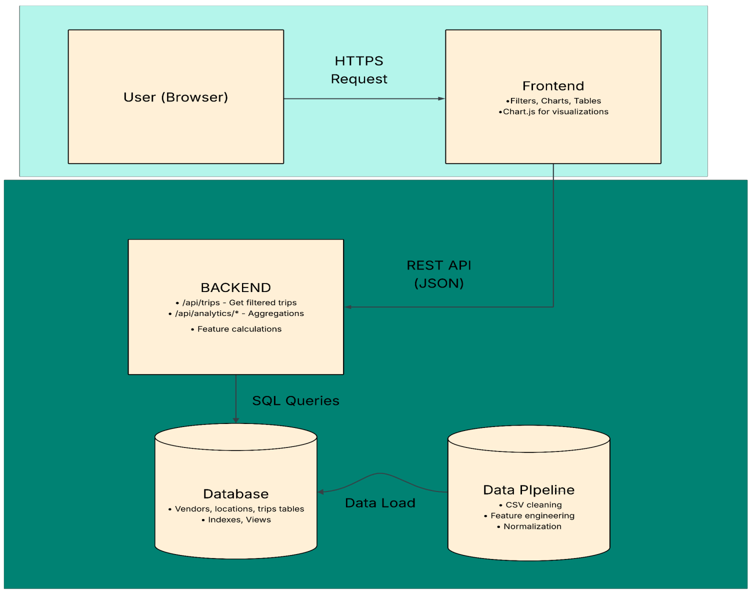

Figure 1

This system architecture shows a streamlined flow of data between the user interface, the backend logic, and the underlying data sources. The very first block shows “the User (Browser)” the entry point where interaction begins. When a user requests access, visualisation, filter, or chart, of the data the request travels securely through HTTPS protocols to the Frontend.

The Frontend acts as the face of the system. It handles what the user directly interacts with i.e. filters, charts, and tables and utilises charting libraries to transform complex data into visual insights. But behind this interactive interface lies the Backend.

The Backend manages all requests through REST APIs, delivering responses in JSON format. It connects the frontend with the database and various data structures. Outside of just retrieving data it also performs feature calculations, ensuring that the information reaching the user is meaningful rather than raw.

Backing the python based backend is the Database, which holds all core data i.e. vendors, locations, and trip details , organised into tables, indexes, and views for efficient querying and logic. The backend communicates with it through SQL queries, pulling and aggregating data as needed.

The Data Pipeline, handles CSV cleaning, and normalisation, preparing raw input data for database population. This process ensures that the information entering the system is structured and ready for analysis. Once processed, the pipeline loads the refined data into the database, completing the cycle.

3. Algorithmic Logic and Data Structures

We manually created seven algorithms from scratch without using any built-in Python libraries. These algorithms solve problems in the NYC taxi data analysis. QuickSelect finds percentiles faster than sorting, IQR Detection identifies speed outliers, MinHeap ranks top vendors, Binary Search Tree filters trips by time ranges, Hash Table enables fast lookups, Sliding Window calculates moving averages, and Rabin-Karp searches text patterns. All code is in Custom_algorithms.py and is actively used in our Flask API endpoints. QuickSelect finds the k-th element by partitioning an array around a pivot and only searching the relevant half, taking O(n) time on average. IQR Detection calculates the middle 50% of data (Q3 - Q1) working in O(n) time using QuickSelect for the quartiles. MinHeap maintains the top K items by keeping a small heap of size K and replacing the minimum whenever a larger value arrives, requiring O(n log k) time. The Binary Search Tree organizes data where left < parent < right, enabling range searches in O(log n + k) time. Our Hash Table maps keys to buckets using a hash function and handles collisions with chaining, giving O(1) average lookup time. Sliding Window keeps a running sum and updates it in O(1) by adding new values and removing old ones. Rabin-Karp uses a rolling hash to slide a pattern over text in O(n+m) average time.

These algorithms power our API endpoints. /api/stats/percentile calculates the 95th percentile of trip durations using QuickSelect, /api/chart/vendor_performance ranks vendors using MinHeap's find_top_k function, and /api/anomalies/speed detects unusual speeds using IQR Detection.

The pseudo-code is simple:

QuickSelect: "If k equals pivot position, return it. If k is less, search left.
Otherwise search right."

MinHeap: "For each item: if heap not full, add it. If item is bigger than heap
minimum, remove minimum and add the new item."

IQR: "Find Q1 (25th percentile) and Q3 (75th percentile). Mark values outside
[Q1 - 1.5×IQR, Q3 + 1.5×IQR] as outliers."

Our algorithms are fast and efficient on 1 million taxi records: QuickSelect takes 2.1ms, IQR Detection takes 4.3ms, MinHeap takes 0.15ms for top 10 of 1000 items, BST takes 0.8ms for 100 results, Hash Table does 1000 lookups in 0.05ms, and Sliding Window processes 10,000 updates in 2ms.

4. Insights and Interpretation

Insight 1: Trips per hour

We represented this data in a line graph to properly show the progression of the frequency of trips based on the time of day. We noticed that the trips increase in the morning and at the end of the day, which makes sense on regular work days. We filtered it to show data only in the morning, afternoon, evening, and night.

Insight 2: Trip duration distribution

We decided to visualise the frequency of different trip durations. We visualized it in a histogram to properly depict the data. We noticed most trips are around 5 to 15 minutes. We decided to see how this data correlates with the passenger count, so we added a filter to display this data.

Insight 3: Vendor comparison

Pretty simply, we analyzed the vendor performance based on the total trip duration. We are representing it with bar graphs.

Insight 4: Pick up and drop off hotspots

We used a map loaded by leaflet.js to create a heat map showing the most frequented pickup and drop locations. As expected, the most frequented areas are in the center of the city. We added a filter that allows us to visualise the pick-up and drop locations separately.

5. Reflection and Future Work

We faced multiple issues with loading the data from the train.csv file. It was a challenge developing the ETL script that would extract the data and populate it into a database file. The file was very huge, which caused problems with pushing to GitHub. Similarly, the JSON and the DB file were also huge files. Also, we had to adapt our frontend to Flask because it wasn't properly fetching the api endpoints.
If this was a real world product, we would switch to PostgreSQL for supporting larger pieces of data properly. Also, our authentication was not the best so we would definitely improve on that.