NYC - URBAN MOBILITY WEB APPLICATION

Introduction: This documentation describes the analysis of the Yello Taxi August 2025 NYC dataset and the implementation of a web application that displays the analysis findings.

1. Problem Framing and Dataset Analysis

A. The dataset and its context

Two different datasets were provided for this project. The first dataset was from the Yellow Taxi Trip Records (August 2025), and the second dataset was taken from the New York City Taxi Trip Dataset. We decided to use the Yellow Taxi Trip Records dataset for this project because it contained much more raw data from which we could draw meaningful insights. We avoided using the New York City Taxi Trip Dataset because it had already been cleaned for another specific purpose, thereby reducing the amount of raw data available for analysis. From the website that hosts the New York City Taxi Trip Dataset, the description explicitly states:

"The competition dataset is based on the 2016 NYC Yellow Cab trip record data made available in BigQuery on Google Cloud Platform. The data was originally published by the NYC Taxi and Limousine Commission (TLC). The data was sampled and cleaned for the purposes of this Playground competition. Based on individual trip attributes, participants should predict the duration of each trip in the test set."

B. Data challenges

Although the dataset contained a large amount of raw data, it was not straightforward to understand and analyze. The major challenge we faced was the lack of context for the fields included in the tables. We had to conduct additional research to make this data more comprehensible. The second problem was the fact that the file was in Parquet format, which we had never worked with before. It took us some research to figure out how to manipulate this format using pandas and pyarrow.

C. Assumptions made during data cleaning

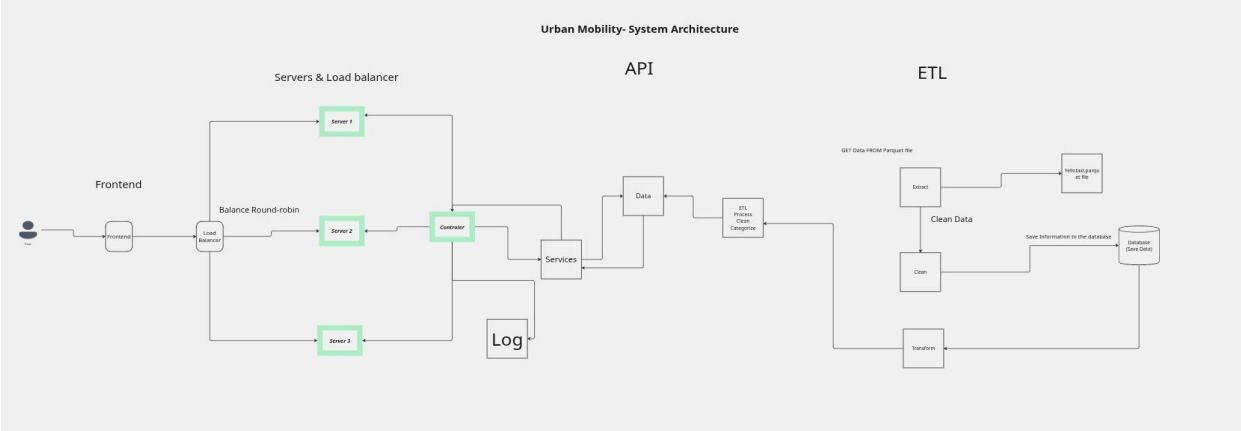
During data cleaning, we decided that any row with a missing entry was incomplete and therefore removed it. Since we were working with a large dataset, removing a few incomplete rows would not affect the integrity of the analysis we would later perform.

D. Unexpected observation that influenced our design

The factor that most affected our design was the size of the dataset. Working with over three million rows of data required a lot of computational power. We initially looped through each column of the dataset and attempted to add it to the database, but this was not only slow but also failed because the computer could not process all that memory simultaneously. We therefore decided to use a chunking technique, which drastically improved data processing.

2. System Architecture and Design Decisions

A. System architecture diagram



B. Stack choices and schema structure:

1. Client & Frontend: The client interacts with our platform through a user interface built in HTML5, CSS, and JavaScript
2. Load Balancer and servers: The user makes a request to the server acting as the load balancer of our platform. The load balancer will direct the request to the appropriate server for processing. We choose the balanced round-robin to equilibrate each request and avoid overloading any server. In the event that one server is unavailable, the next one can take over.
3. API:

   Programming language used: FastAPI

   - Controller layer: once a request reaches to the server, the controller acts as a middleman to send the request to the appropriate endpoint, which then redirects to the appropriate service to be processed

- Service layer: the service layer where all our business logics and computations happen. The service layer is divided into sections: extract and load. The service layer calls the sub-layer load, which in turn calls the extract method. After the data is clean and the computation is complete, the data is transferred to the service layer, then to the controller layer, which returns it to the frontend.
- Data Layer: The data layer is responsible for the ETL (Extract, Transform, Load) process. The data layer loads the Parquet file for studies, cleans the data, and prepares it for saving. The data is transferred to the sub-layer save, which saves different data to the appropriate table in our database.
- Logs layer: The log layer is responsible for storing each request & response that the API has made, along with the appropriate timestamp. This layer allows tracking the speed and frequency of requests & responses. It serves as the entry point for all debugging processes we need to make.
- Internal server (Uvicorn): Uvicorn is responsible for serving our API services

C. ETL: The Extract, Transform, Load process manages the entire process of obtaining NYC study data, cleaning it, saving it, transforming it, and loading the data.

D. Trade-offs made during the design process:

1. FastAPI (speed & developer ergonomics) vs. heavier frameworks

- Choice & reason: FastAPI offers high performance (with async endpoints), automatic documentation (OpenAPI), and rapid developer iteration.
- Trade-off: Smaller ecosystem for some specialized plugins compared to, say, Django. Also requires careful async handling to avoid concurrency bugs.
- Impact: Faster dev and good runtime throughput; slightly more manual work for auth/ORM patterns.
- Mitigation: Use proven libraries (SQLAlchemy and async patterns)

2. Uvicorn as an internal server vs. full-featured application servers

- Choice & reason: Uvicorn is lightweight, async-capable, and pairs easily with FastAPI.

- Trade-off: Uvicorn isn't a full process manager; it requires a process manager (such as systemd, Docker, or Kubernetes) and a reverse proxy for production concerns.
- Impact: Minimal overhead and good latency, but additional operational setup is required.
- Mitigation: Deploy behind a reverse proxy (NGINX) or an orchestration platform (Kubernetes) with process supervision.

3. Balanced round-robin load balancing vs. smarter (health-aware / least-connections)

- Choice & reason: Round-robin is simple and easy to implement/configure.
- Trade-off: Round-robin may not optimally distribute load when servers have uneven load or response times.
- Impact: Possible suboptimal utilization under heterogeneous server performance or long-running requests.
- Mitigation: Combine round-robin with health checks and sticky session avoidance; consider switching to least-connections or a load balancer with request-weighting when real traffic patterns are measured.

4. Data layer separation (extract/load/save) vs. monolithic scripts

- Choice & reason: Clear responsibilities improve maintainability and testing.
- Trade-off: More files and interfaces to manage; slightly higher integration overhead.
- Impact: Better modularity and testability; initial development overhead.
- Mitigation: Keep interfaces small and well-documented; add integration tests and shared utilities.

5. Storing detailed logs for every request vs. performance & storage cost

- Choice & reason: Comprehensive logs help with debugging, performance monitoring, and audits.
- Trade-off: High volume of logs increases storage costs and can impact performance if synced/blocking.
- Impact: Better observability but higher cost and potential I/O overhead.
- Mitigation: Use asynchronous/non-blocking logging, structured logs, log rotation, sampling for high-frequency endpoints, and ship logs to a centralized service with retention policies.

6. **Using SQLite3 database (relational) for saved tables vs NoSQL and other Relational Database alternatives**

For this assignment and due to time constraints, we used sqlite3 database file

- Choice & reason: Relational DBs fit structured schema (Taxizones, trips, Ratecode, etc.) and transactional guarantees.
- Trade-off:

  Less flexible schema evolution and potentially less horizontal scalability for certain workloads compared with NoSQL

No security is provided since SQLite3 creates a database file that is accessible to everyone but can be secured through file permissions, unlike MySQL and PostgreSQL, which offer a database server and require permission before accessing the database.

- Impact:

  Strong consistency and ease of joins; scale limits on write-heavy workloads.

  A SQLite3 database is good for testing and prototyping. The user can clone our current project, build the database, and interact directly with the platform without the need to set up a database server.

- Mitigation: Optimize schema and indexing, use read replicas or sharding when needed, and consider hybrid patterns (caching in Redis) for high-read endpoints.

  Our next step is to convert our database, created using SQLite3, to one that runs inside a database server.

Conclusions:

We chose simplicity, clarity, and development velocity for a complex task for the initial design while keeping an eye on production concerns (observability, modularity, maintainability). For each development choice, we documented the associated risks and provided concrete mitigations, enabling the system to evolve toward robustness and scalability without requiring major rewrites.

3. Algorithmic Logic and Data Structures

For the data structures and algorithms, we decided to implement searching for a single element by its ID. The goal of our two algorithms is to compare the performance of the two manually implemented search algorithms — a Linear Search and a Key-Value Search — for finding a TaxiZone records by its location ID in the trips database. We are loading the TaxiZones table of our database.

Problem solved:

In large datasets, searching for a specific record can be slow unless it is optimized. This algorithm illustrates how various search approaches perform when retrieving data, enabling the identification of efficient methods for handling lookups in real-world applications.

Linear search:

We implemented a search algorithm with linear search (looping into each index to find the element with the specific ID. The algorithm iterates through each TaxiZone record to find the one matching the given ID.

Pseudocode:

```
function linear_search(list, target):
    for each element in list:
        if element.id == target:
            return element
    return None
```

Key-value search (Manual dictionary simulation):

This algorithm builds a list of (key, value) pairs before performing the search. It simulates a dictionary lookup structure but is implemented manually (without Python's built-in `dict` or hashing).

Pseudocode:

```
function key_value_search(list_of_pairs, target):
  for (key, value) in list_of_pairs:
    if key == target:
      return value
  return None
```

Complexity Analysis:

Linear Search	Key-Value Search:
Time Complexity: O(n)	Time Complexity: O(n)
Space Complexity: O(1)	Space Complexity: O(n)

Although both are O(n) in time, the key-value search introduces extra space usage due to creating an intermediate list of tuples. This demonstrates trade-offs between preprocessing and lookup performance.

Result:

Testing with ID = 200

Found data

Key-Value Search Elapsed time: 0.00000358 seconds

Found data

Linear Search Elapsed time: 0.00002265 seconds

Testing with ID = 225

Found data

Key-Value Search Elapsed time: 0.00000429 seconds

Found data

Linear Search Elapsed time: 0.00003862 seconds

Testing with ID = 250

Found data

Key-Value Search Elapsed time: 0.00000405 seconds

Found data

Linear Search Elapsed time: 0.00004292 seconds

Testing with ID = 300

Data not found

Key-Value Search Elapsed time: 0.00000501 seconds

Data not found

Linear Search Elapsed time: 0.00002766 seconds

4. Insights and Interpretation

From the Yellow Taxi August 2025 dataset, we gained various insights, including the most visited regions of New York based on the number of trips and population. We also analyzed regions in New York where vendor VeriFone Inc. We used Pandas.

- **Most visited places by the number of trips**: To find the most visited places based on the number of trips, we group the destination location ID with the passenger count, then count each row, as one row represents a trip. Then, we reset the index to have a well-formatted dataframe.
- **Most visited places by population:** To determine the most visited places based on population, we employed the same approach as the first one. After grouping, we summed the passenger counts by region. We reset the index to have a well-formatted dataframe.
- **Regions where Verifone Inc. dominates the most:** To find regions where Verifone dominates the most, we grouped the pickup location ID with the vendor ID, then resized the new dataframe and reset the index to create a visually appealing dataframe. Then, inside the newly created dataframe, we grouped the pickup location ID by trip count, sorted it in descending order, and identified the most dominant regions (top 10).

We have provided charts to display the results on our web pages

5. Reflection and Future Work

This summative assessment enabled us to apply the various concepts learned during the first half of the trimester and build upon our previous skills.

Skills learned:

Programming & Development

- Backend development using FastAPI and Python
- Implementing asynchronous programming for improved I/O performance
- Writing ETL scripts for data extraction, transformation, and loading
- Error handling and logging for better debugging and observability
- Testing and debugging backend services to ensure data consistency and reliability
- Version control using Git and GitHub for source management and collaboration

System Design & Architecture

- Building and selecting the right system architecture design for scalability and maintainability
- Designing Entity Relationship Diagrams (ERDs) for database modeling
- Applying the Model–Service–Controller pattern to separate concerns

- Designing and implementing RESTful APIs with clear endpoints and structure
- Understanding load balancing concepts (balanced round-robin) for distributed systems
- Implementing logging layers for monitoring and system diagnostics

Data Engineering & Analysis

- Handling large datasets efficiently using chunking techniques
- Data cleaning, transformation, and analysis using *pandas* and *pyarrow*
- Loading and saving structured data with SQLite3 and understanding database schema design
- Implementing data pipelines (ETL) for real-time or batch processing
- Visualizing insights and trends using charts and tables

Frontend & User Experience

- Developing web interfaces using HTML5, CSS, and JavaScript
- Ensuring the separation of frontend and backend for maintainability
- Designing a user-friendly dashboard for displaying analytical results
- Applying UI/UX principles for clear data presentation and smooth navigation

Project Management & Documentation

- Communicating technical decisions and trade-offs through clear documentation
- Collaborating effectively in a team setting with structured workflows
- Managing project lifecycle from data exploration to deployment preparation

We believe that our system architecture, database, API, UI, and UX design provide a solid foundation for building a robust application that can handle multiple concurrent tasks at high speed, while providing a visually appealing dashboard to users.

Next steps:

- Caching system**:** Integrate a caching layer (Redis) to store frequently accessed results and reduce backend request load.
- HTML Templates and Dynamic Loading: We plan to implement dynamic loading on our frontend to enhance the user experience when navigating different resources.

- Full Asynchronous Processing: extend asynchronous handling beyond API endpoints to include data loading, transformation, and logging operations to further improve I/O performance.
- Deployment and containerization: Implement a full deployment with real servers, load balancers, or Dockerize it.
- Database Optimization: Transition from SQLite3 to a production-grade database system such as PostgreSQL or MySQL for better concurrency, security, and scalability.