Distributed Systems

Class Group Project – UBER Simulation

Project Due Date (Presentation and Demo): **9th December 2024**

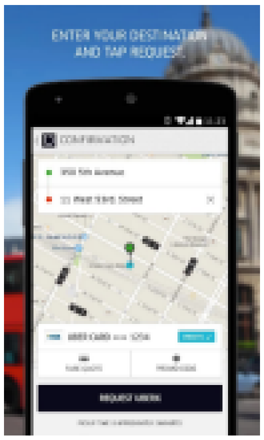
Turn in the following on or before November 10th , 2024. No late submissions will be accepted!

- An API design document of services provided by the application.

You are allowed to use GenAI for this group project only.

Project History

Uber Technologies Inc. is a transportation network company headquartered in San Francisco, California. The company develops, markets and operates the Uber mobile app, which allows consumers with smartphones to submit a trip request, which is then routed to Uber drivers who use their own cars
https://play.google.com/store/apps/details?id=com.ubercab&hl=en



Back in early 2012, Uber's Boston team noticed a problem. On Friday and Saturday nights, around 1am, the company was experiencing a spike in "unfulfilled requests." The root cause was that drivers were clocking off the system to go home, just before the weekend partygoers were ready to venture home themselves. There was a supply-demand imbalance, and the result was a lot of very unhappy customers. So the Boston team had an idea. What if they offered the drivers a higher price to stay on the system longer (until around 3 AM)? Would more take-home dollars for drivers increase supply? In just two weeks, they had a resounding answer. By offering more money to drivers, they were able to increase the on-the-road supply of drivers by 70–80% and, more importantly, eliminate two-thirds of the unfulfilled requests. The supply curve was highly elastic. Drivers were indeed motivated by price.

Pricing

Uber predicts fares and implements dynamic pricing using a combination of advanced algorithms, historical data, and machine learning techniques. Here's a breakdown of how it works:

1. **Base Fare Calculation**: Uber starts by calculating a base fare, which is determined by several factors such as the distance between the pickup and drop-off locations, estimated travel time, and local rates. This basic fare is computed before any dynamic pricing adjustments.

2. **Historical Data & Demand**: Uber leverages historical data such as the number of available drivers in an area, the current demand for rides, and traffic conditions. This is combined with historical data on peak travel times, popular routes, and seasonal trends. When demand for rides exceeds available drivers (e.g., during rush hour or major events), prices can increase dynamically to balance the demand with the supply of drivers.

3. **Machine Learning:** Uber uses machine learning models to **predict current demand patterns and optimize pricing strategies**. These models analyze historical data to anticipate when and where demand will spike, allowing Uber to adjust prices ahead of time or in real-time to match the forecasted conditions.

By combining these **data-driven** techniques, Uber ensures fares are fair while also leveraging dynamic pricing to balance supply and demand, ultimately improving efficiency and reliability for both drivers and passengers.

Make use of the following Kaggle data set when determining your price:
https://www.kaggle.com/datasets/yasserh/uber-fares-dataset

- key - a unique identifier for each trip
- fare_amount - the cost of each trip in usd
- pickup_datetime - date and time when the meter was engaged
- passenger_count - the number of passengers in the vehicle (driver entered value)
- pickup_longitude - the longitude where the meter was engaged
- pickup_latitude - the latitude where the meter was engaged
- dropoff_longitude - the longitude where the meter was disengaged
- dropoff_latitude - the latitude where the meter was disengaged

Explain in detail how your pricing decision algorithm, machine learning techniques, and the use of Kaggle data for prediction have been applied.

Review score
Users of the app may rate drivers on a scale of 1 (bad) to 5 (good); in turn, drivers may rate users. A low rating might diminish the availability and convenience of the service to the user.

Project Overview

In this project, you will design a 3-tier application that will implement the functions of **the Uber System** for cab services. You will build on the techniques used in your lab assignments to create the system.

In the Uber System, you will manage and implement different types of objects:

- Drivers
- Customers
- Billing
- Admin

For each type of object, you will also need to implement an associated **database schema** that will be responsible for representing how a given object should be stored in a relational database.

Your system should be built upon some type of distributed architecture. You have to use web services as the middleware technology. You will be given a great deal of “artist liberty” in how you choose to implement the system.

Your system must support the following types of entities:

• **Drivers** – It represents information about an individual driver registered on Uber. You must manage the following information for this entity:

⇒ Driver ID [SSN Format]

⇒ First Name

⇒ Last Name

⇒ Address

⇒ City

⇒ State

⇒ Zip Code

⇒ Phone number
⇒ Email
⇒ Car details
⇒ Rating
⇒ Reviews
⇒ Own introduction in the form of images and video.
⇒ Rides history

• **Customers** – It represents additional information about an individual customer. You must manage the following state information for this entity:

⇒ Customer ID [SSN Format]

⇒ First Name

⇒ Last Name

⇒ Address

⇒ City

⇒ State

⇒ Zip Code
⇒ Phone number
⇒ Email
⇒ Credit Card details
⇒ Rides History
⇒ Rating
⇒ Reviews

• **Billing Information** – It represents information about billing for rides. You should provide predicted and real prices for rides at different times. You must manage the following state information for this entity:

⇒ Billing ID [SSN Format]

⇒ Date

⇒ Pickup time

⇒ Drop off time

⇒ Distance covered

⇒ Total amount for ride

⇒ Source location

⇒ Destination location

⇒ Driver ID
⇒ Customer ID

• **Administrator** – It represents information about the administrator of the Uber System. You must manage the following state information for this entity:

⇒ Admin ID [SSN Format]

⇒ First Name

⇒ Last Name

⇒ Address

⇒ City

⇒ State

⇒ Zip Code
⇒ Phone number

⇒ Email

- **Rides** – It represents information about a ride. You must manage the following state information for this entity:
  - ⇒ Ride ID [SSN Format]
  - ⇒ Pickup location (Latitude, Longitude)
  - ⇒ Drop off location (Latitude, Longitude)
  - ⇒ Date/Time
  - ⇒ Customer ID
  - ⇒ Driver ID
- **dataset collection**
  - o key - a unique identifier for each trip
  - o fare_amount - the cost of each trip in usd
  - o pickup_datetime - date and time when the meter was engaged
  - o passenger_count - the number of passengers in the vehicle (driver entered value)
  - o pickup_longitude - the longitude where the meter was engaged
  - o pickup_latitude - the latitude where the meter was engaged
  - o dropoff_longitude - the longitude where the meter was disengaged o dropoff_latitude - the latitude where the meter was disengaged

Project Requirements

Your project will consist of three tiers:
- The client tier, where the user will interact with your system
- The middle tier/middleware/messaging system, where the majority of the processing takes place
- The third tier, comprising a database to store the state of your entity objects

Tier 1 — Client Requirements

The client will be a node application that allows a user to do the following:

- **Driver Module/Service:**
  - · Create a new Driver
  - · Delete an existing Driver
  - · List all Drivers known by the system
  - · Change a driver's information (name, address, etc) – *This function must support the ability to change ALL attributes*
  - · Search for drivers based on attributes. You do not have to consider attributes that are not listed above.
  - · Display information about a driver.
  - · Introduction video of the driver.

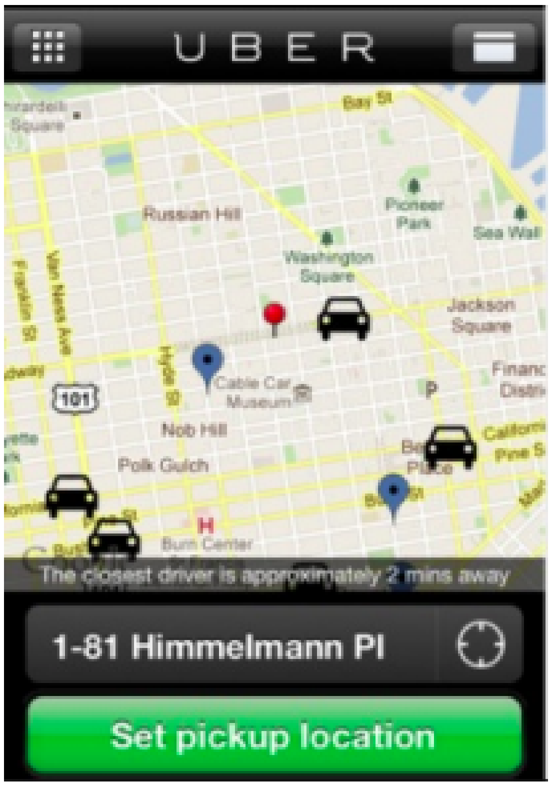
- **Customer Module/Service:**
  · Create a new Customer.
  · Delete an existing Customer
  · List all Customers known by the system
  · Generate Bill for a Customer (Every Ride).
  · Select driver within 10 miles from Customer location.
  · Upload images about events during his ride.

- **Billing Module/Service:**
  · Create a new Bill for each ride.
  · Delete an existing Bill.
  · Search an existing Bill.

- **Admin Module/Service:**
  · Add drivers to the system.
  · Add customers to the system.
  · Review driver/customer account.
  · Show statistics (revenue/day) and total rides (area wise).
  · Show graphs/charts for different rides per area, per driver, per
  customer. · Search for a Bill based on attributes
  · Display information about a bill.

- **Rides Module/Service:**
  · Create a new ride.
  · Edit an existing ride.
  · Delete an existing ride.
  · List all rides of a customer.
  · List all rides of a driver.
  · Show ride statistics as per location.

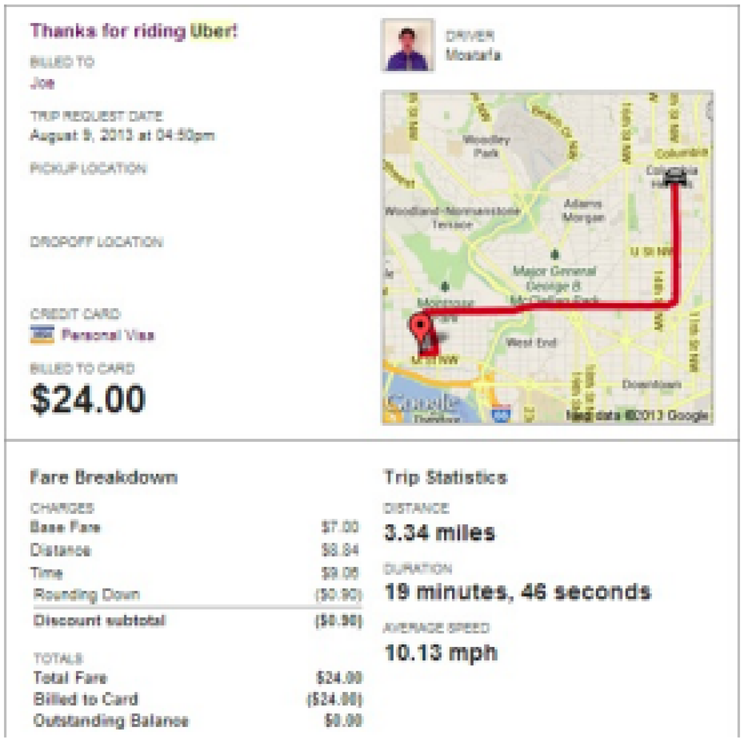
- Display the location of drivers within 10 miles of radius from customer location.



The closest driver is approximately 2 mins away

1-81 Himmelmann Pl

- Sample billing report



Thanks for riding Uber!

BILLED TO
Joe

TRIP REQUEST DATE
August 9, 2013 at 04:50pm

PICKUP LOCATION

DROPOFF LOCATION

CREDIT CARD
Personal Visa

BILLED TO CARD
$24.00

DRIVER
Mostafa

Map data ©2013 Google

Fare Breakdown

CHARGES	
Base Fare	$7.00
Distance	$8.84
Time	$9.06
Rounding Down	($0.90)
Discount subtotal	($0.90)
TOTALS	
Total Fare	$24.00
Billed to Card	($24.00)
Outstanding Balance	$0.00

Trip Statistics

DISTANCE
3.34 miles

DURATION
19 minutes, 46 seconds

AVERAGE SPEED
10.13 mph

- Sample Admin Analysis Report

You may add any extra functionality you want (optional, not required), but if you do so, you must document and explain the additional features. You still must implement all the required features, however.

The client should have a pleasing look and be simple to understand. Error conditions and feedback to the user should be displayed in a status area on the user interface.

Ideally, you will use an IDE tool to create your GUI.

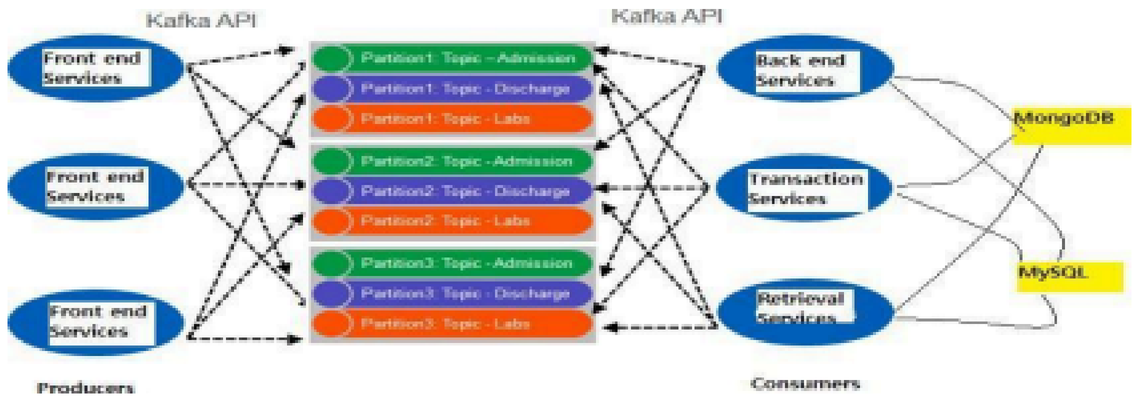
Tier 2 — Middleware

You will need to provide/design/enhance a middleware that can accomplish the above requirements. You have to implement it using REST-based web services as middleware technology. Next, decide on the interface your service would be exposing. You should ensure that you appropriately handle error conditions/exceptions/failure states and return meaningful information to your caller. Make sure you handle the specific failure cases described below.

ON OR BEFORE the date listed below, you must turn in a document describing the interface your server will use, which precisely means that you have to give API request-response descriptions.

Your project will store the state of each driver and customer in a relational database. Your project should also include a model class that uses standard modules to access your database. Your entity objects will use the data object to select/insert/update the data in the relational database.

User Kafka as a messaging platform for communication between front-end channels with backend systems.



Tier 3 — Database Schema and Database Creation

You will need to design a database table that will store your relational data. Choose column names and types that are appropriate for the type of data you are using. You will also need a simple program that creates your database and its table. The MySQL Workbench is a very efficient and easy tool for it.

You will need to use MySQL/MongoDB for storing driver introductions and images of drivers and customer reviews and updates related to the car or driver during his ride.

Scalability, Performance and Reliability

The hallmark of any good project is scalability. A highly scalable system will not experience degraded performance or excessive resource consumption when managing large numbers of objects or when processing large numbers of simultaneous requests. You need to make sure that your system can handle many drivers, customers, and incoming requests.

Pay careful attention to how you manage "expensive" resources like database connections.

Your system should easily be able to manage 10,000 drivers, 10,000 customers, and 100,000 billing records. Consider these numbers as **minimum** requirements.

Further, for all operations defined above, you need to ensure that if a particular operation fails (for example, a guard allocated two buildings at the same time), your system is left in a consistent state. Consider this requirement carefully as you design your project, as some operations may involve multiple database operations. You may need to make use of transactions to model these operations and roll back the database in case of failure.

Testing

To test the robustness of your system, you need to design a test harness that will exercise all the functions that a regular client would use. This test harness is typically a command-line program. You can use your test harness to evaluate scalability (described above) by having the test harness create thousands of drivers and customers. Use your test harness to debug problems in the server implementation before writing your GUI.

Other Project Details

Turn in the following on or before the due date. No late submissions will be accepted!
· A title page listing the members of your group
· A page listing how each member of the group contributed to the project (keep this short, one paragraph per group member is sufficient)
· A short (5 pages max) write-up describing:
  a) Your object management policy
  b) How you handle "heavyweight" resources
  c) The policy you used to decide when to write data into the database
· A screen capture of your client application, showing some actual data
· A code listing of your client application
· A code listing of your server implementations for the entity
objects · A code listing of your server implementation for the
session object · A code listing of your main server code
· A code listing of your database access class
· A code listing of your test class
· Any other code listing (utility classes, etc)
· Output from your test class (if applicable)
· A code listing of your database creation class (or script)
· A screen capture showing your database schema
· Observations and lessons learned (1 page max)

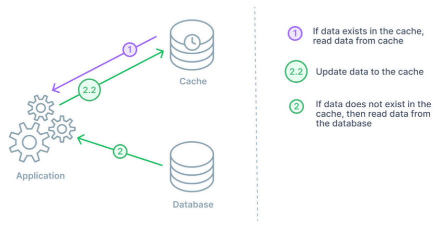
You also need to demo the project and submit a softcopy (via E-Mail to syshim@gmail.com) of all code needed to make your project functional.

The possible project points are broken down as follows:
• **40% for basic operation** – Your server implementation will be tested for proper operation of some aspect of your project. Each passed test is worth some point(s) toward the total score, depending on the test. Each failed test receives 0 points.
• **15% for scalability and robustness** – Your project will be judged on its scalability and robustness. I will test your program with thousands of objects; it should not exhibit sluggish performance, and it definitely should not crash. In addition to performance improvement techniques covered in the class, you are required to implement SQL caching using Radis and show performance analysis.
• **10% for distributed services – Depoly with Docker into AWS (Kubernetes)**. Divide client requirements into distributed services. Each service will be running on a backend connected by Kafka, divide your data into
• **MongoDB or/and MySQL(5%)** and provide performance data with justification on results.
• **5% for dynamic pricing algorithm** – Devise your own dynamic pricing algorithm and explain why it fits the supply and demand Uber model.
• **10% for the client** – As this project primarily stresses server-side development, the client GUI is not as important. However, I do want to see some sort of GUI developed. You must code a client GUI; I will not accept client-less submissions.
• **15%** for your test class and project report

Project Presentation:

1. Group number and team details
2. Database Schema
3. System Architecture Design Diagram
4. Dynamic Pricing Algorithm
4. Scalability/Performance comparison bar graphs that show the difference in performance by adding different distributed features at a time, such as B (Base), S (SQL Caching), and K (Kafka) for 100 simultaneous user threads. (4 bar graphs in total). You can use Apache JMeter: An open-source tool for performance and load testing

SQL Caching: https://www.prisma.io/dataguide/managing-databases/introduction-database-caching


Cache
Application
Database
1 If data exists in the cache, read data from cache
2.2 Update data to the cache
2 If data does not exist in the cache, then read data from the database

Combinations are:
a. B
b. B+S
d. B+S+K
e. B+S+K+ other techniques you used

Note: Populate DB with at least 10,000 random data points and measure the performance.

Hints:

· Cache entity lookups: To prevent a costly trip to the database, you can cache the state of entity objects in a local in-memory cache. If a request is made to retrieve the state of an object whose state is in the cache, you can reconstitute the object without checking the database. Be careful that you invalidate the contents of the cache when an object's state is changed. In particular, you are required to implement SQL caching using Radis.

· Don't write data to the database that didn't change since it was last read.
· *Focus FIRST on implementing a complete project* – remember that a complete implementation that passes all of the tests is worth as much as the performance and scalability part of the project.

· Do not overoptimize. Project groups in the past that have tried to implement complex optimizations usually failed to complete their implementations.

Exceptions/Failure Modes

Your project MUST properly handle the following failure conditions:
- Creating Duplicate Driver/Customer
- Addresses (for Person) that are not formed properly (see below)

For more failure conditions, see Other Notes below

Other Notes

State abbreviation parameters

State abbreviations should be limited to valid US state abbreviations or full state names. A Google search will yield these valid values. Methods accepting state abbreviations as parameters are required to raise the 'malformed_state' exception (or equivalent) if the parameter does not match one of the accepted parameters. You do not need to handle US territories such as the Virgin Islands, Puerto Rico, Guam, etc.

Zip code parameters

Zip codes should adhere to the following pattern:
[0-9][0-9][0-9][0-9][0-9]
or
[0-9][0-9][0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]
Examples of valid Zip codes:
95123
95192
10293
90086-1929
Examples of invalid Zip codes:
1247
1829A
37849-392
2374-2384

Driver/Customer IDs are required to match the pattern for US social security numbers:
[0-9][0-9][0-9]-[0-9][0-9]-[0-9][0-9][0-9][0-9]
You may assume that any SSN matching the above pattern is valid (there are some reserved social security numbers starting with 0 that are not valid in real life, but you may consider them to be valid for the purposes of the project). Any method accepting a driver ID is required to raise the 'invalid_driver_id' exception (or equivalent) if the supplied parameter does not match the pattern above.

Peer Review

Each team member should write a review about other team members except himself/herself. It is confidential, which means that you should not share your comments with other members. Peer review is submitted separately on Canvas.

Appendix

AI Engineer designs and develops scalable solutions using AI tools and machine-learning models. Ensures the scalability, reliability, and performance of AI systems.
An AI engineer is a specialized programmer who designs, develops, and manages artificial intelligence (AI) systems. They are responsible for building AI-based applications, such as chat interfaces and full-stack applications. AI engineers work on a variety of tasks, including:

- Developing algorithms: AI engineers create algorithms that allow machines to learn from data. For example, they might create an algorithm to optimize inventory based on sales trends.
- Building data processing techniques: AI engineers build advanced data processing techniques to handle large-scale data processing.
- Ensuring ethical considerations: AI engineers ensure that AI systems are designed with fairness, privacy, and security in mind.
- Integrating AI systems: AI engineers integrate AI systems with other software applications.
- Optimizing AI algorithms: AI engineers optimize AI algorithms for performance and efficiency.
- Deploying AI models: AI engineers integrate AI models into various applications. For example, they might deploy a trained chatbot model to provide customer service on a website.