

DATA 236 Distributed Systems

Uber Simulation Project

Group-5

Aishwarya Gulab Thorat- 017557579

Sheetal Patnaik - 017526678

Leon Corriea- 017410965

Lincy Rebello- 017414072

Dhruv Shetty- 017506541

[Github Link](#)

Individual Contributions:

- Aishwarya Gulab Thorat: She focused on Customer microservices and fare calculation algorithm. She worked on customer registration, profile management, and integrated the ML-based dynamic pricing model for accurate fare predictions.
- Sheetal Patnaik: Her main focus was on the driver microservices which included multiple things like driver login, signup, profile management, availability status and also checking for the real time tracking. In order to establish a live driver customer communication she also implemented the web socket connections.
- Leon Corriea: He focused on the overall system design, focusing on the API Gateway and microservices integration. Implemented the Kafka message broker for event streaming and coordinated the deployment using Docker and Kubernetes.
- Lincy Rebello: She focused on the ride service microservice, including ride status management, route optimization, and ride completion processes. Also implemented the Redis caching layer for improved system performance.
- Dhruv Shetty: He focused on the admin dashboard and corresponding admin service. Implemented system configuration management, analytics reporting, and the ticketing system for customer and driver support.

Object management policy:

Our approach was separation of each entity that is driver customer ride and billing. Is managed by its respective Django model and view set. this insured the independence capability of services

Strict validation applied during object creation and updation

- Driver IDs and customer IDs must confirm to the US social security number pattern.

- Feels like email phone number and vehicle license plates are unique and validated before saving

Caching frequently accessed objects such as driver and ride details are cached using redis to minimize database queries and enhance performance. Cached objects are updated or invalidated upon significant changes ie. location updation or ride acceptance.

Instead of permanently deleting entities **soft deletes** was implemented by marking objects as inactive. this preserves historical data and prevents any accidental data loss.

To handle race conditions for eg: two drivers Accepting the same ride. **Atomic transactions** was used in Django views which ensured consistency.

Handling “heavyweight” resources:

Ride Fair prediction

- A machine learning mode (XGBoost with the accuracy of 79%) calculates and predicts the fair during right creation.
- To reduce latency preprocessing and prediction occur in an optimized pipeline using a lightweight ML model.
- The results are cached to reuse in subsequent queries e.g during ride confirmation or billing.

Websocket connections

- Websocket is used for real-time communication between driver front end and backend
- Each active driver maintains a persistent websocket connection to receive a ride request and updates

Load balancing

- To prevent overloading websocket connections are balanced across multiple server instances.

Timeouts

- Inactive connections are terminated after fixed period to free up resources.

File uploads

- Multimedia files like profile photos and introduction videos are stored in the database
- large files are streamed to prevent excessive memory consumption during upload

Database queries

- Complex queries, such as find nearby drivers, use geospatial indexing e.g post GIS for efficiency .
- Batch processing is used for bulk operations like retrieving ride histories.

The policy we used to decide when to write data into the database:

Few things which was in consideration during the decision of writing data to our database is as follows:

Immediate writes for critical transactions

- Actions impacting the systems state e.g ride booking, driver acceptance are written immediately to ensure consistency. these operations are wrapped in atomic database transactions to prevent partial updates

Deferred writes for non-critical updates

- Non-critical updates such as driver location updates or customer profile edits are done and processed asynchronously using celery workers.

Validation before writing

- Frontend: basic validation e.g required Fields, email format
- Backend: comprehensive validation e.g SSN unique constraint before saving.

Caching before writing

- Frequently accessed objects are cached for instance ride request, driver status. changes are written to the cache first and asynchronously synced to the database
- For example: driver locations are updated in redis first and written into the database in bulk every 5 minutes.

Event driven updates

- Billing creation or ride status updates trigger database writes only when the event completes successfully.
- Example: a bill is generated and saved only after the ride is marked as complete.

Archiving and soft deletes

- Historical data e.g completed rides, old driver records are archived periodically to Separate Tables or external storage to reduce primary database load.

For example:

Ride booking

- The customer submits a booking request with pickup and drop off details the request is validated for accuracy.
- The fare is predicted and the ride is created immediately in the database to ensure other drivers do not see the same request.
- Ride details are cached to reduce subsequent read operations.

Driver location updates

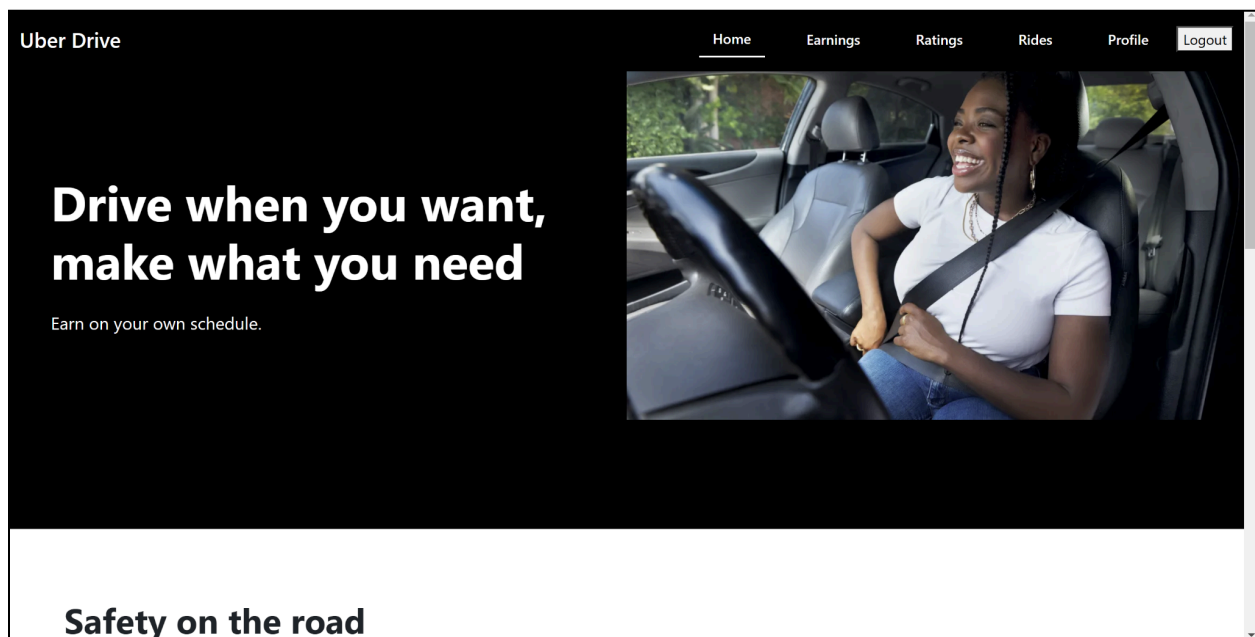
- Frequent updates are cached driver's locations are stored in redis for proximity queries.
- locations are written to database every 5 minutes to reduce Write load.

By using caching and deferred writes we reduce the load on the primary database. Critical transactions are always atomic to prevent race conditions and data corruption. Heavyweight resources like file uploads and machine learning predictions are offloaded or optimized for performance.

Screenshots

These are few of the screenshots which show some of our features from our application:

Driver's Home Page



Book Ride Page for Customer

Uber Ride[Home](#)[Book Ride](#)[My Rides](#)[Profile](#)[Logout](#)

Book Your Ride

Pickup Location

Dropoff Location


Ride Type

XL (4 passengers) ▼

Number of Passengers

1 ▼

MapSatellite




Customer's Home Page

Uber Ride[Home](#)[Book Ride](#)[My Rides](#)[Profile](#)[Logout](#)

Ride with comfort and convenience

Your ride, your way – fast, easy, and safe.



Ride with Confidence

Customer's Ride Details Page with review option

Uber Ride

HomeBook RideMy RidesProfileLogout

Ride Details

Driver ID: 100-00-0002

Pickup Location: 28 N Almaden Ave, San Jose, CA 95110, USA

Dropoff Location: 355 Sunol St, San Jose, CA 95126, USA

Pickup Time: 08:25 PM

Dropoff Time: 08:25 PM

Add Review

Rating (1-5)

5

Content

Good Driver!

Submit Review

Specific Bill based on each ride

Uber Ride

HomeBook RideMy RidesProfileLogout

Bill Details

Billing ID	538-10-9755
Bill Date	2024-12-08
Pickup Time	08:25 PM
Drop-off Time	08:25 PM
Distance Covered	1.12 miles
Total Amount	\$13.32
Source Location	28 N Almaden Ave, San Jose, CA 95110, USA
Destination Location	355 Sunol St, San Jose, CA 95126, USA
Driver ID	100-00-0002
Customer ID	100-00-1005

Driver's Ride History

Uber Drive

Home

Ratings

Rides

Profile

Logout

My Rides

Total Earnings: \$52.66

Ride ID	Created	Customer	Pickup	Dropoff	Fare	Status
1	12/8/2024, 8:23:56 PM	100-00-1005	28 N Almaden Ave, San Jose, CA 95110, USA	355 Sunol St, San Jose, CA 95126, USA	\$13.32	completed
2	12/8/2024, 8:38:04 PM	100-00-1005	665 W Olive Ave, Sunnyvale, CA 94086, USA	The Embarcadero, San Francisco, CA 94133, USA	\$21.42	completed
3	12/8/2024, 8:48:13 PM	100-00-1005	2 Pierce Ave, San Jose, CA 95110, USA	415 Mission St, San Francisco, CA 94105, USA	\$21.42	completed
4	12/8/2024, 9:44:03 PM	100-00-1005	28 N Almaden Ave, San Jose, CA 95110, USA	345 Park Ave, San Jose, CA 95110, USA	\$9.67	completed

Driver Profile Details and Update Page

Uber Drive

Home

Ratings


Rides

Profile

Logout

Profile Information

Cancel



Choose Filembappe.jpg

LAST LOGIN

LAST NAME

driver

DATE JOINED

2024-12-09T01:50:06.597731Z

EMAIL

dhruvdriver@example.com

ADDRESS

FIRST NAME

dhruv

IS ACTIVE

true

DRIVER ID

100-00-0002

PHONE NUMBER

4085674531

CITY

7

Admin Dashboard

Admin Dashboard

[Profile](#) [Logout](#)

Choose an action to manage

Manage Drivers

Manage Customers

Manage Billing

View Statistics

Admin Billing Management to search and query bills

Billing Management

Billing ID

Start Date

mm/dd/yyyy

End Date

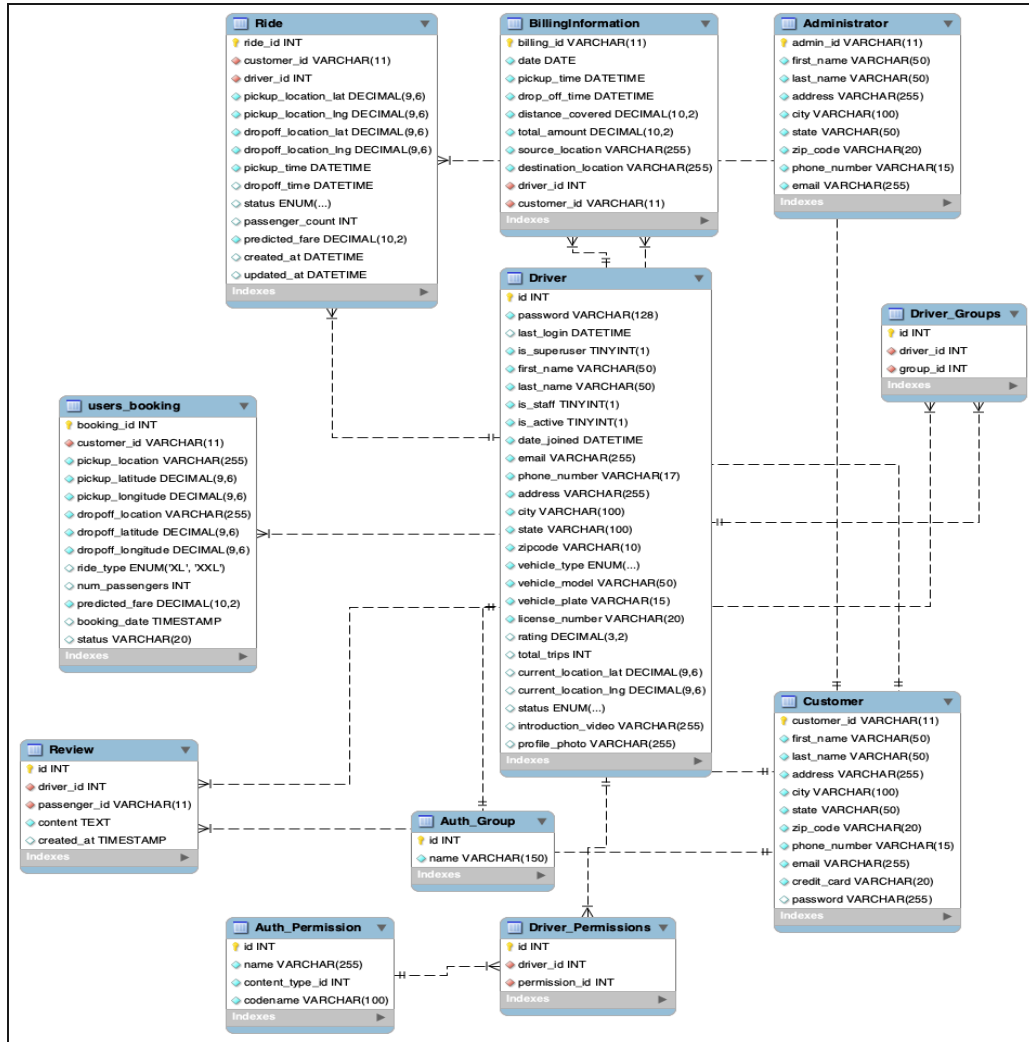
mm/dd/yyyy

Search Bills

Reset

Bill ID	Date	Customer	Driver	Distance	Amount	Source	Destination	Actions
470-49-7360	2024-12-08	100-00-1005	100-00-0002	36.22 miles	\$21.42	665 W Olive Ave, Sunnyvale, CA 94086, USA	The Embarcadero, San Francisco, CA 94133, USA	<div>View Details</div>
538-10-9755	2024-12-08	100-00-1005	100-00-0002	1.12 miles	\$13.32	28 N Almaden Ave, San Jose, CA 95110, USA	355 Sunol St, San Jose, CA 95126, USA	<div>View Details</div>
660-35-9213	2024-12-08	100-00-1005	100-00-0002	0.26 miles	\$9.67	28 N Almaden Ave, San Jose, CA 95110, USA	345 Park Ave, San Jose, CA 95110, USA	<div>View Details</div>
825-46-2695	2024-12-08	100-00-1005	100-00-0002	42.53 miles	\$21.42	2 Pierce Ave, San Jose, CA 95110, USA	415 Mission St, San Francisco, CA 94105, USA	<div>View Details</div>

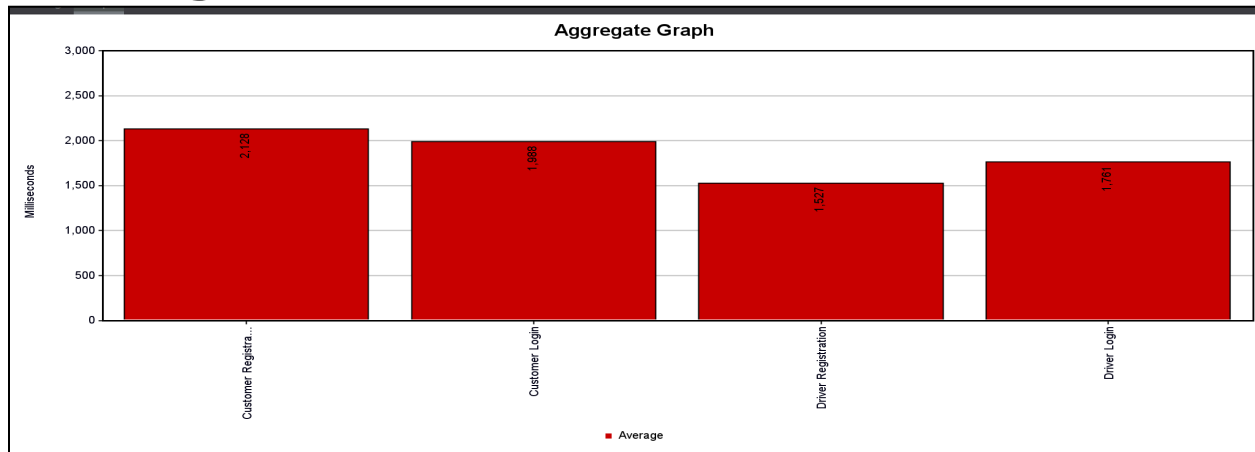
Database Schema Design



Testing:

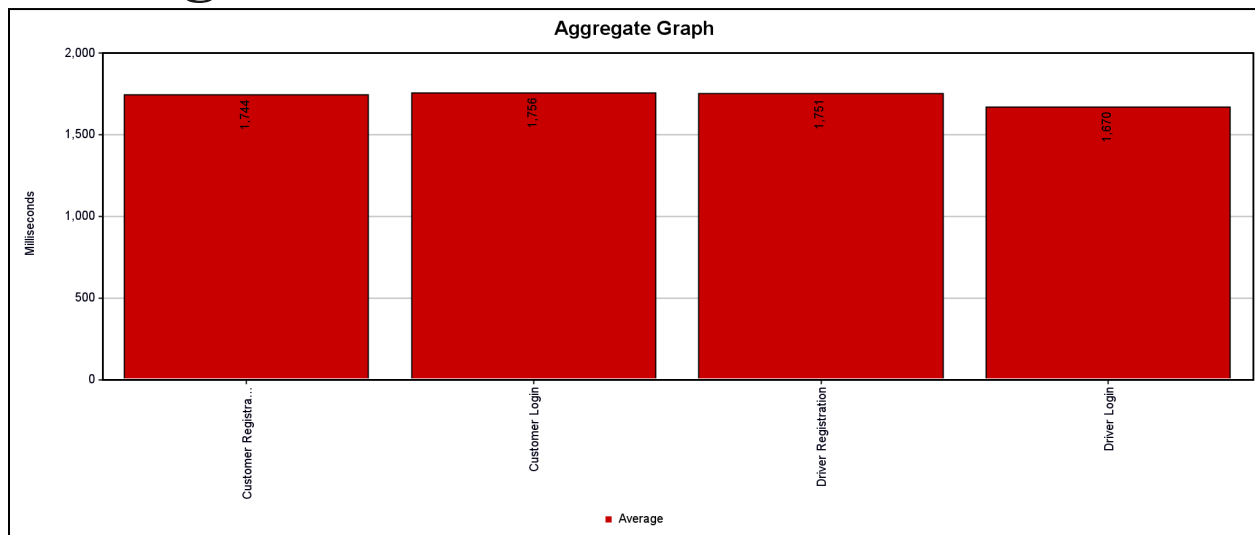
Our project is tested for performance and scalability using Apache JMeter, simulating real-world scenarios with concurrent user activity. Test plans include core functionalities like Driver Registration, Customer Registration, Login, and Ride Booking, configured to simulate up to 100 simultaneous user threads with unique data for each user. The JMeter tests are designed to evaluate the robustness of API endpoints under load, ensuring low latency and error-free performance. Distributed testing is implemented using multiple JMeter servers to assess scalability. Results are analyzed for throughput, response time, and resource utilization, enabling optimization of the backend services for production readiness.

Testing: B



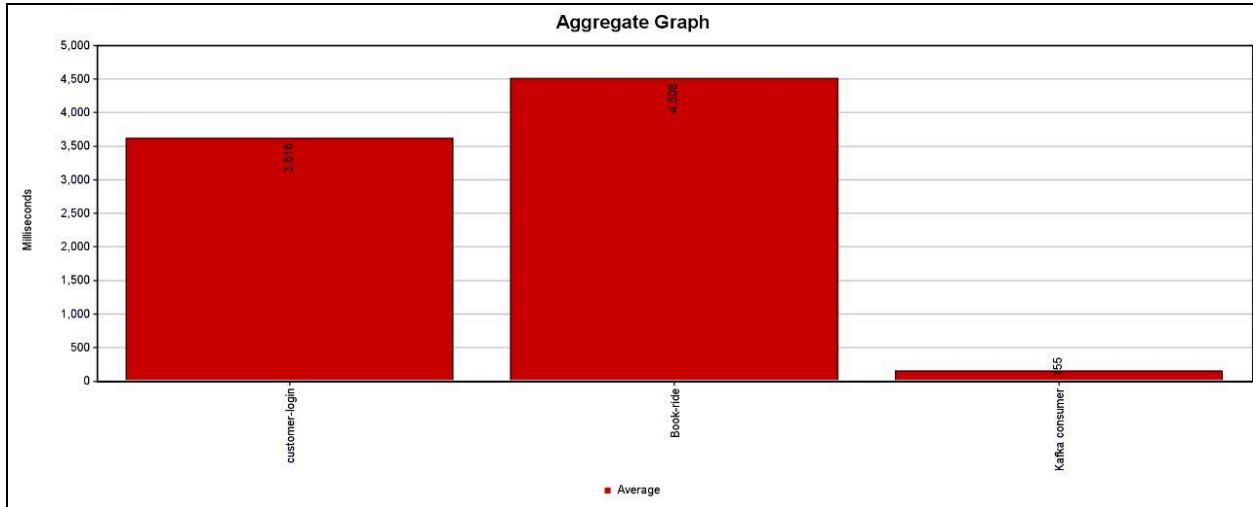
Measured performance of login and registration with direct database queries.

Testing: B+S



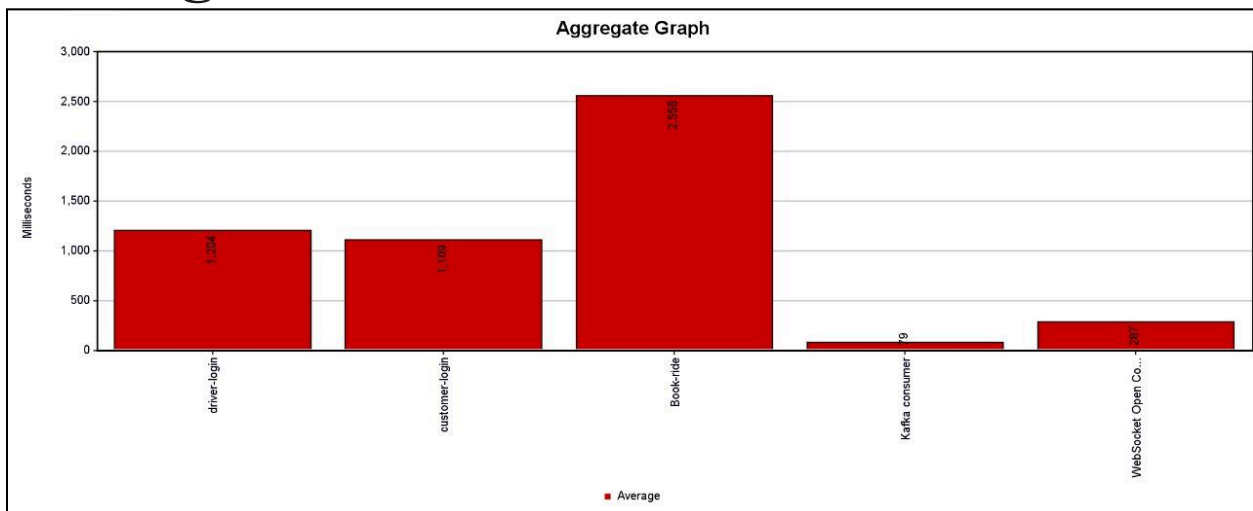
Improved login and registration throughput by reducing database load with caching.

Testing: B+S+K



Measured booking ride performance with asynchronous notifications via Kafka.

Testing: B+S+K+C



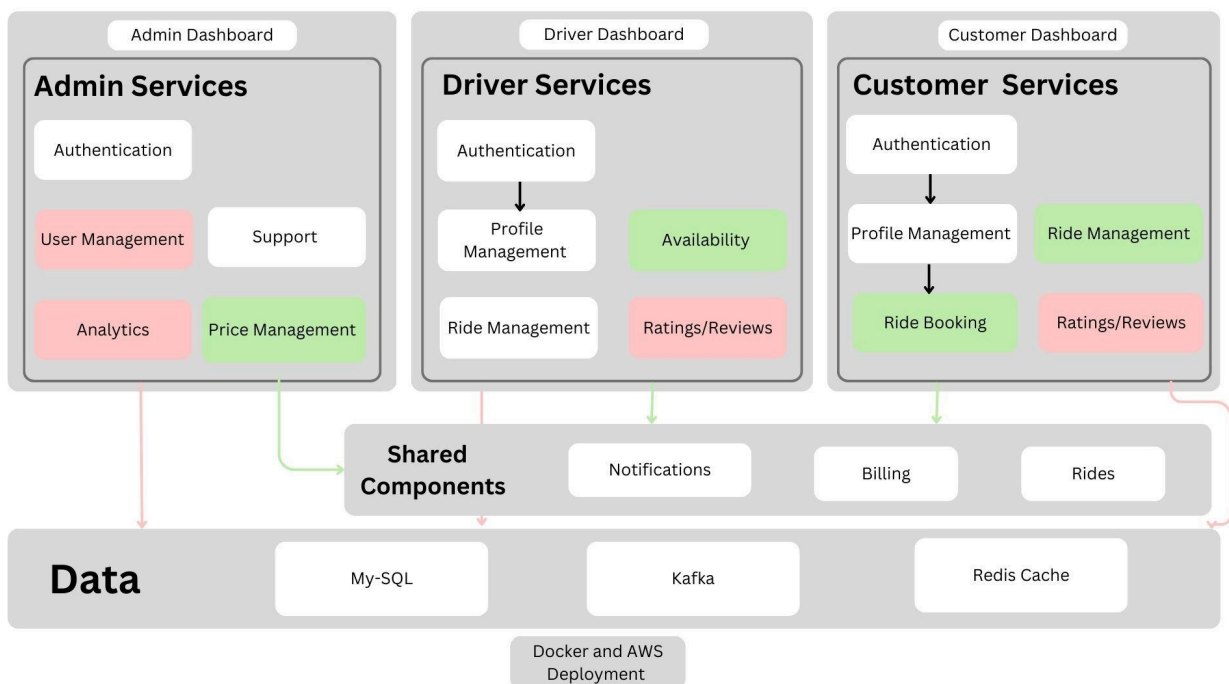
Measured real-time booking notifications between customers and driver using Redis-backed WebSocket messaging.

Deployment

Upon successful completion of the project we deployed our project in AWS. Our project is deployed using Docker and Kubernetes for scalability and efficient resource management. Each backend module, including Driver, Customer, Billing, and Rides, is containerized using Docker, and images are stored in a private container registry. The deployment leverages Kubernetes for orchestration, ensuring high availability and fault tolerance. Redis is deployed as a Kubernetes service for caching, with configurations dynamically loaded from .env files for flexibility.

Persistent Volumes are used for media storage, including customer and driver profile images. The database is managed via a MySQL StatefulSet, ensuring persistent and reliable storage of transactional data. The system is hosted on AWS, utilizing EKS for Kubernetes, S3 for static assets, and CloudWatch for monitoring and logs.

Workflow:



Our system architecture follows a modern microservices-based distributed architecture with three primary tiers:

1. Client Tier: This is the tier where all the user interactions begin at the client level through the web application. We have divided our client side interface into three distinct dashboards:

A. Customer Dashboard: In this dashboard we mainly focus on the ride booking and management.

B. Driver Dashboard: Here we focused on making a dashboard which allows the driver to accept rides and manage his availability.

C. Admin Dashboard: In order to manage the whole system and monitor everything we have the admin dashboard.

2. Our **API Gateway** functions as a single entry point for all the client requests and it also gives us vital services like request routing, authorization and authentication along with maintaining a limit on the rate. Along with this we have also made sure we cover load balancing also.

3. **Core Business Layer** or the Microservice layer: Here each of the microservice is independent and handles its own specific business functionality.

Below are a few of the main services we focus on.

Driver location tracking

Google Maps API is used to track and update driver locations in real time through the browser this approach allows for accurate and up-to-date positioning of drivers.

Ride booking process

Customer location based booking

customer can only place booking request if there are drivers available within a 10 mile radius of the location. this restriction and shows that rides are feasible and reduces the likelihood of long wait times for customers.

Driver eligibility for ride requests

Ride request are only sent to drivers who meet the two critical criterias

1. They are within 10 miles of customers pickup location.
2. they do not have any ongoing rides.

This criteria system optimizes driver customer matching and improves overall service efficiency.

Ride management

Single Active Ride policy

To prevent overbooking customers are restricted from booking another ride while they have an ongoing ride. this ensures that each customer is associated with only one Active Ride at a time.

Ride completion and billing

Upon marking a ride as completed the system automatically generates a bill. this bill along with detailed ride information is made available to the customers in the 'My Rides' section of the account. These allows customers to review their trip history.

Driver rating and review system

The system has a rating mechanism for drivers based on customer reviews. After each completed ride Customers have the option to review their driver. these reviews contribute to the driver's overall ratings.

Driver earnings and ride history

Drivers have access to ride history including

- Details of each completed ride
- The total Fair of individual rides
- Cumulative earning calculation

The driver's earnings are calculated as 80% of the total fare, reflecting the standard Uber Commission of 20%. This system allows drivers to track their performance and earnings effectively.

4. Data Management Layer: Here the system uses the multiple data storage solutions for different purposes:

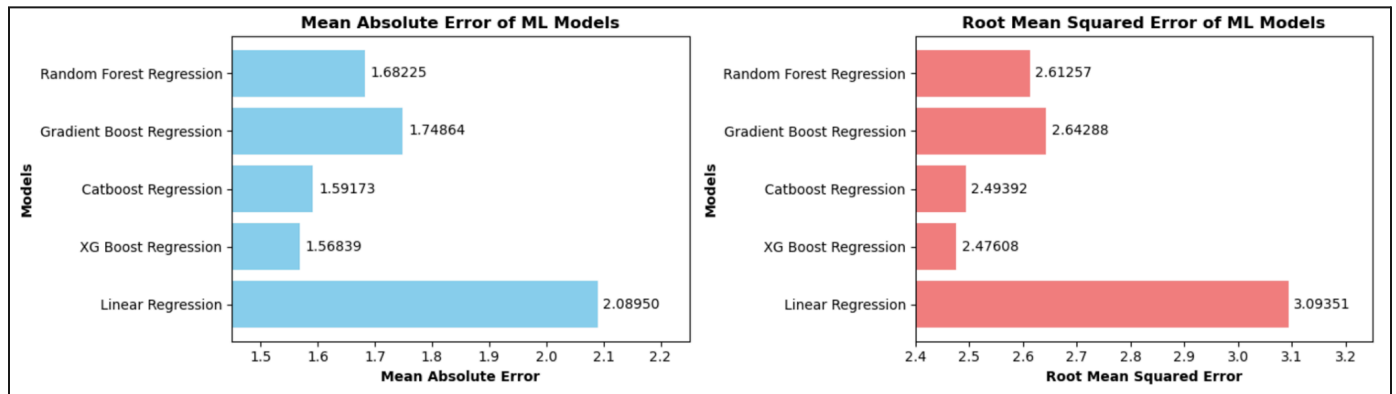
- **Kafka :** This is the Message Broker in our application which handles the real-time event streaming and also manages the service to service communication. It helps in processing the location updates by allowing the location at the time of usage of our application and it handles the ride status changes by giving real time notifications. Ride creation events and other crucial updates are handled through Kafka ensuring reliable and scalable event processing. This approach allows for efficient handling of high volume real time data streams.
- **Redis Cache:** This stores the most frequently accessed data by caching the driver location and maintaining active ride information and by storing the temporary session data. It also improves response times for common queries. Caching frequently access data, such as driver locations and Active Ride information. this significantly reduces database load and improves system response time.
- **MySQL Database:** We have used MySQL Database which stores our application data and maintains the user profiles along with the transaction history and ride records.
- **Websocket notifications:** Real time notifications to both customers and drivers are done using websocket connections. This enables instant updates on ride status, driver arrival and other time sensitive information.

Dynamic Pricing Decision Algorithm:

We explored multiple machine learning algorithms to identify the one which was giving the best results.

We evaluated multiple ML algorithms using R2-score, MSE and MAE.

Evaluation of multiple ML algorithms



R2-scores-

1. Linear Regression - 0.67
2. **XGBoost Regressor - 0.79**
3. Catboost Regressor - 0.78
4. Random Forest Regressor - 0.76
5. Gradient Boosting Regressor - 0.76

We have selected XGBoost Regressor as it was performing well as compared to other models.

Features considered in ML model -

1. Passenger Count
2. Distance between pickup and drop off using Haversine formula
- Booking date and time -> Hour, Weekday, Date, Month, Year
 3. Year
 4. Hour -> Early Morning, Morning, Afternoon, Evening, Late Night
 5. Day -> WeekDay or WeekEnd
 6. Day -> Early in Month, In the middle of the month, At the end of the month
 7. Month -> Quarters (1,2,3,4)
- Boxing Coordinates using UTM

To better analyze location-based data, we used the Universal Transverse Mercator (UTM) system to divide the map into 500x500-meter grid squares. Each grid square was assigned a unique ID, and location coordinates were grouped into these squares. This simplified the organization and analysis of spatial data.

8. Pickup box ID

9. DropOff box ID

Performance of the model for the morning ride vs evening ride -

Same Booking details for the morning ride and evening ride -

```
# predicting same ride at early morning
ride_data = [{
    'Unnamed: 0': 24983189,
    'key': '2021-06-17 02:22:06.0001103',
    'pickup_datetime': '2021-06-17 02:22:06 UTC',
    'pickup_longitude': -73.999817,
    'pickup_latitude': 40.738354,
    'dropoff_longitude': -73.999512,
    'dropoff_latitude': 40.723217,
    'passenger_count': 3
}]

ride_data = pd.DataFrame(ride_data)
data = preprocess_data(ride_data)
fare_predicted_emrng = xgb_model.predict(data)
print(f"Total fare for early morning UBER Ride: ${fare_predicted_emrng[0]:.2f}")

Total fare for early morning UBER Ride: $17.25
```

```
# predicting a ride for 3 people in evening
ride_data = [{
    'Unnamed: 0': 24983189,
    'key': '2021-06-17 19:22:06.0001103',
    'pickup_datetime': '2021-06-17 19:22:06 UTC',
    'pickup_longitude': -73.999817,
    'pickup_latitude': 40.738354,
    'dropoff_longitude': -73.999512,
    'dropoff_latitude': 40.723217,
    'passenger_count': 3
}]

ride_data = pd.DataFrame(ride_data)
data = preprocess_data(ride_data)
fare_predicted_evng = xgb_model.predict(data)
print(f"Total fare for evening UBER Ride: ${fare_predicted_evng[0]:.2f}")

Total fare for evening UBER Ride: $18.55
```

We can clearly see from the above screenshots that the model was predicting a higher price for the evening ride and lower price for the morning ride.

Observations and lessons learned:

Developing Uber stimulation with both challenging and insightful here are some of the key observations and lesson learned

Key observations:

Few of the key observations throughout our UBER rides project is as follow:

- **Scalability and Performance:** This application needs to handle a large volume which includes drivers, customers and many billing records. So while we maintain everything we have observed that implementation of SQL Caching using Redis gave us understanding of how different the performances are on different configurations.
- **Distributed Architecture:** This project made us learn how a true distributed system implementation works as we used multiple technologies (Kafka, Docker, AWS Kubernetes) than a simple client server model. This made us aware about the real-world enterprise architecture requirements and modern microservices approaches.
- **Error Handling and Data Validation:** The project places strong emphasis on proper error handling and data validation, particularly for critical fields like SSN format, zip codes, and state abbreviations. This reflects the importance of data integrity and robust error handling in production systems.

Lessons Learned:

The following are few of the important lessons we learned by doing this project:

- **Importance of load balancing**

The critical nature of proper load balancing especially when dealing with resource intensive operations like websocket connections and ml model predictions. Distributing the load across multiple servers instances was essential for maintaining systems stability under high traffic

- **Containerization**

Docker for containerization and kubernetes for orchestration prove to be a wise decision this technology is streamline the development process and made it easier to manage the complex microservices architecture. this approach greatly simplified scaling and version management

- **Machine learning integration**

Incorporating machine learning for fair prediction was a challenge. We learned the importance of optimizing ml models for production environments ensuring quick prediction times without compromising accuracy