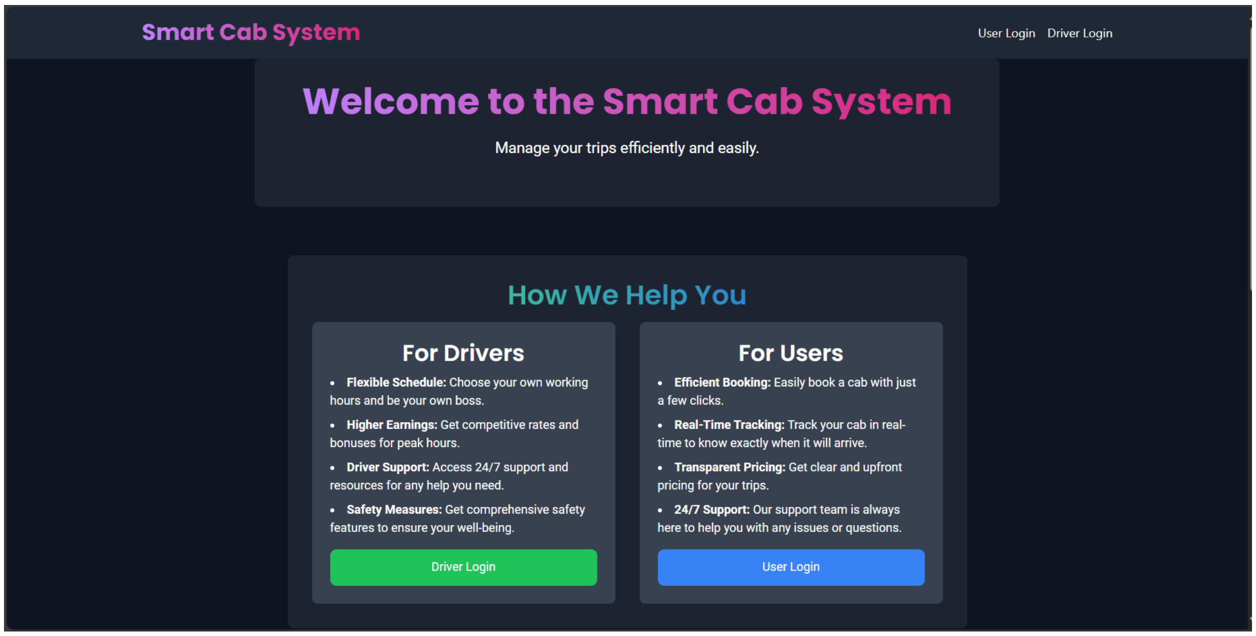
CASE STUDY

Smart Cab Allocation System for Efficient Trip Planning

Tharakadatta D Hegde
B22ES007
IIT Jodhpur



Welcome to the Smart Cab System

Manage your trips efficiently and easily.

How We Help You

For Drivers

- **Flexible Schedule:** Choose your own working hours and be your own boss.
- **Higher Earnings:** Get competitive rates and bonuses for peak hours.
- **Driver Support:** Access 24/7 support and resources for any help you need.
- **Safety Measures:** Get comprehensive safety features to ensure your well-being.

For Users

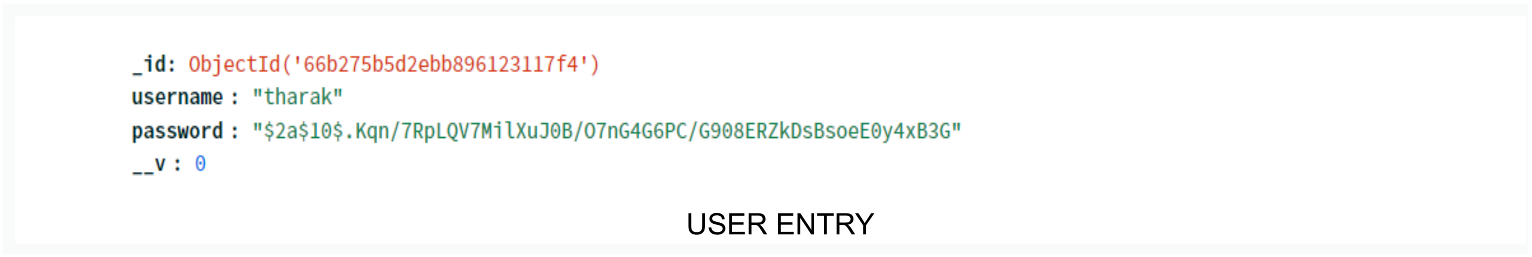
- **Efficient Booking:** Easily book a cab with just a few clicks.
- **Real-Time Tracking:** Track your cab in real-time to know exactly when it will arrive.
- **Transparent Pricing:** Get clear and upfront pricing for your trips.
- **24/7 Support:** Our support team is always here to help you with any issues or questions.

Tech Stack Used
React JS
Express JS
Node JS
MongoDB

Approach

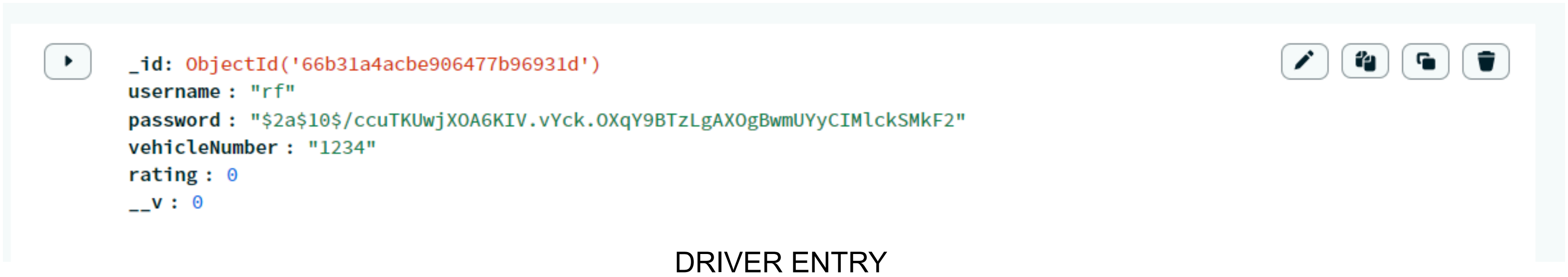
1) User Registration and Authentication

- User Registration: User would register using Basic information like Name , Email, Phone number, password
- Driver registration: Driver would also register using Basic information like name , phone number and password.
- The registered Data would be stored in MongoDB as JSON object.
- Each user would also be allotted a user id and the password would also be stored in the encoded format using JWT Token.



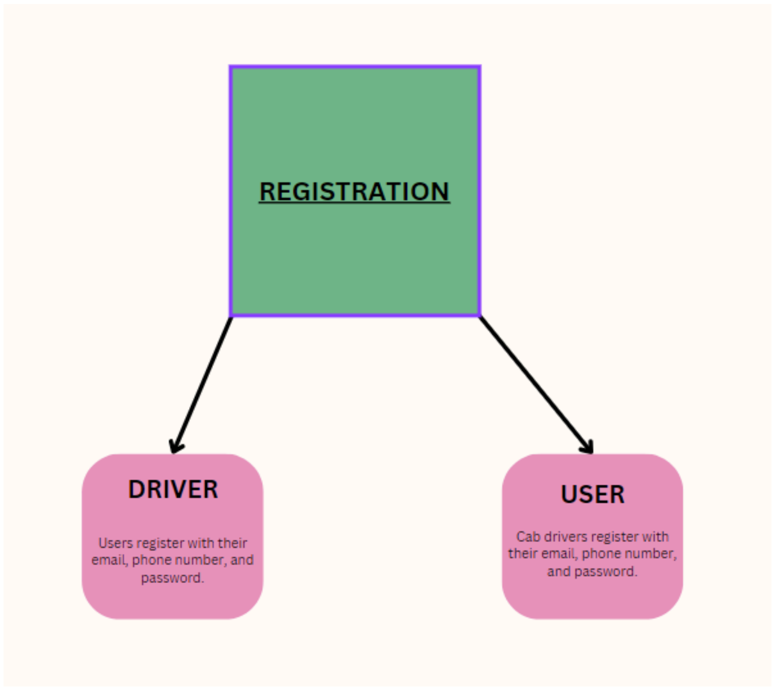
```
_id: ObjectId('66b275b5d2ebb896123117f4')
username : "tharak"
password : "$2a$10$.Kqn/7RpLQV7MilXuJ0B/O7nG4G6PC/G908ERZkDsBsoeE0y4xB3G"
__v : 0
```

USER ENTRY

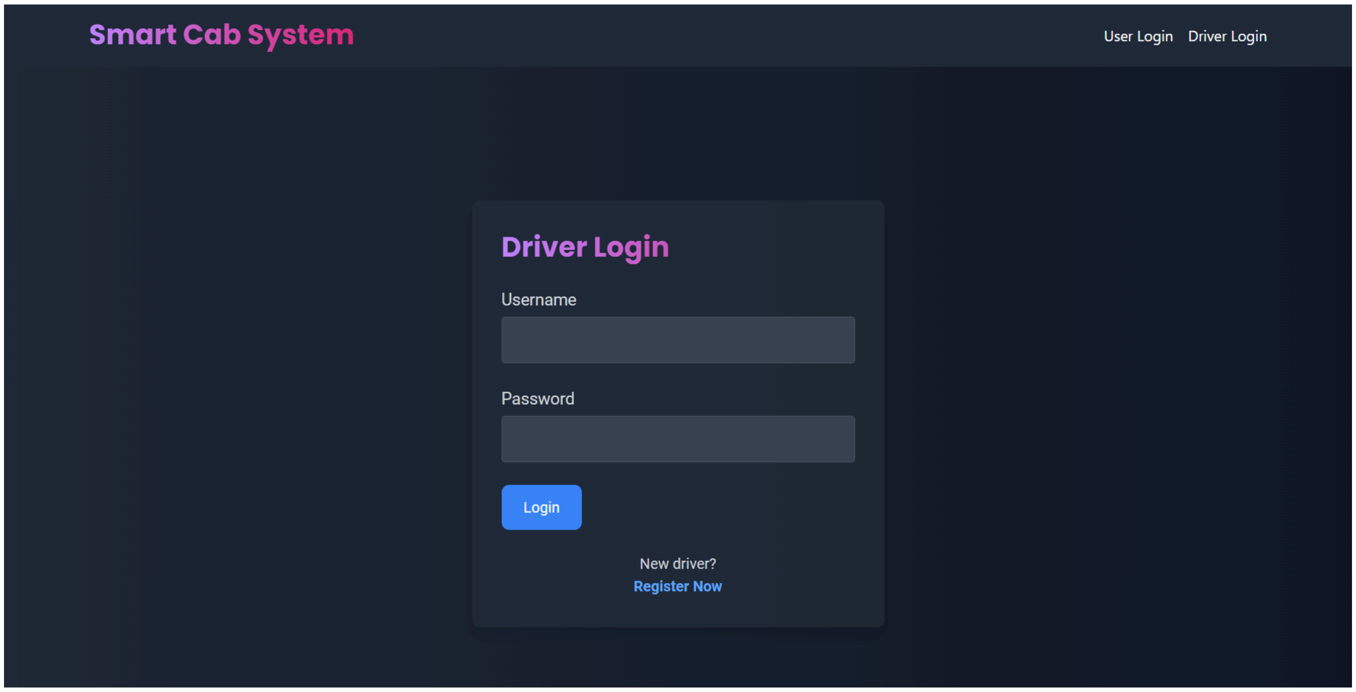


```
_id: ObjectId('66b31a4acbe906477b96931d')
username : "rf"
password : "$2a$10$/ccuTKUwjXOA6KIV.vYck.OXqY9BTzLgAXOgBwmUYyCIMlckSMkF2"
vehicleNumber : "1234"
rating : 0
__v : 0
```

DRIVER ENTRY


REGISTRATION
DRIVER
Users register with their email, phone number, and password.
USER
Cab drivers register with their email, phone number, and password.

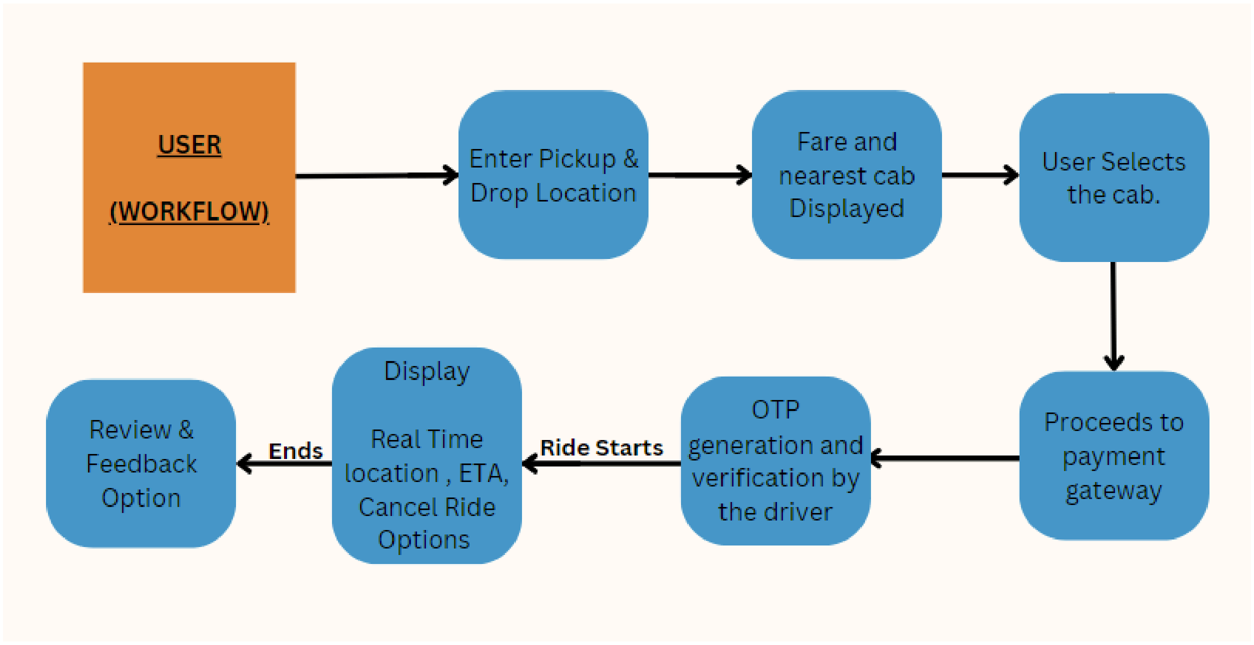
- Separate Login Panels for both User and Driver to login



2) Proposed WorkFlow Post Authentication

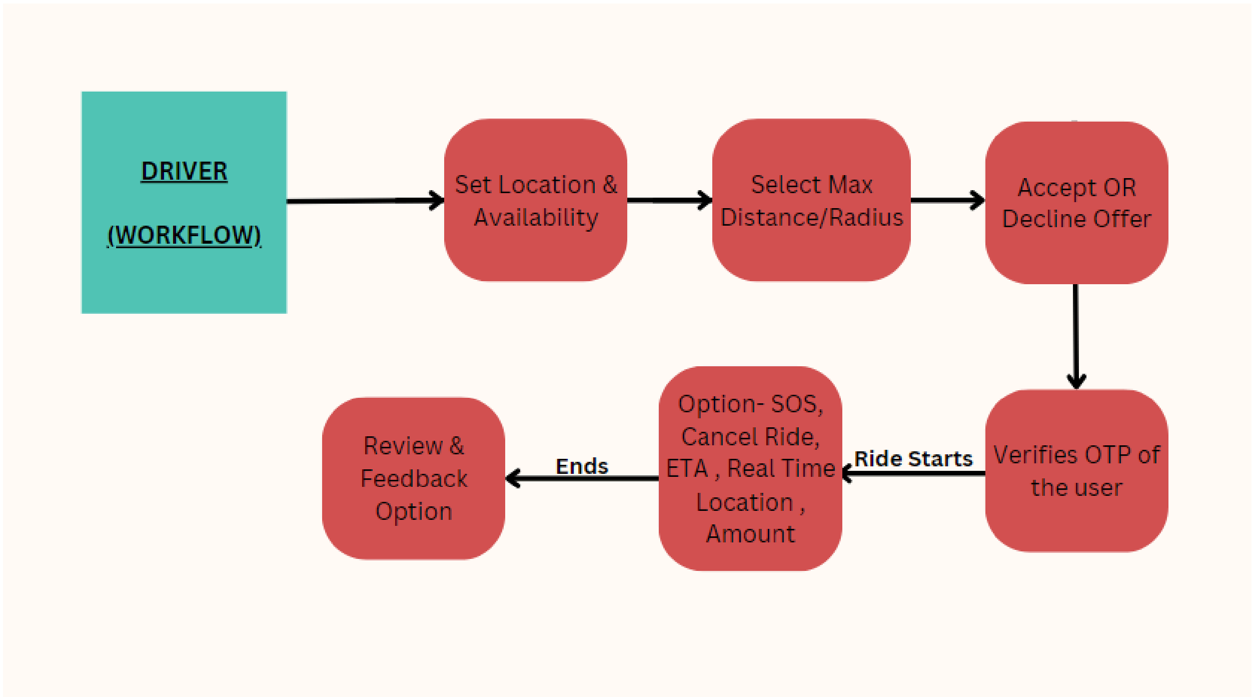
Users:

1. Users can enter their location and request a ride (can book for now or later). They can also book a cab for someone else.
2. During booking, the app shows the nearest cabs and presents the fare and cancellation policies.
3. Users can select the type of cab they wish to ride, such as standard, premium, etc., and their respective price based on the algorithm should be displayed.
4. Once the user has selected their preferred type of ride, they proceed to the payment method (if it's an online payment).
5. The app generates a random OTP (One-Time Password) which needs to be verified by the driver.
6. Users have access to features like SOS button, ETA (Estimated Time of Arrival), fare amount, real-time location, and the option to cancel the ride.
7. After the ride is completed, users can provide reviews and feedback for the driver.


USER (WORKFLOW)
Enter Pickup & Drop Location
Fare and nearest cab Displayed
User Selects the cab.
Proceeds to payment gateway
OTP generation and verification by the driver
Ride Starts
Display Real Time location , ETA, Cancel Ride Options
Ends
Review & Feedback Option

Drivers:

1. Drivers can choose the areas they wish to operate in and set their availability.
2. They can specify the maximum distance they are willing to travel to pick up users.
3. Upon receiving a ride request, drivers have the option to accept or decline it.
4. Drivers receive an OTP that needs to be verified by the user.
5. They have access to features such as canceling a ride (SOS button), real-time location tracking, estimated time of arrival (ETA), and fare amount
6. After the ride is completed, drivers can also receive reviews and feedback from users.


DRIVER (WORKFLOW)
Set Location & Availability
Select Max Distance/Radius
Accept OR Decline Offer
Verifies OTP of the user
Ride Starts
Option- SOS, Cancel Ride, ETA , Real Time Location , Amount
Ends
Review & Feedback Option

3) Use of Object Oriented Programming (OOPs)

Encapsulation:

- Each user's details (such as name, email, and phone number) and actions (like register, login, and request a ride) are encapsulated within the User class.

Inheritance:

- Create a general Person class with common attributes and methods. The User and Driver classes inherit these attributes and methods, while also having additional functionalities specific to their roles.

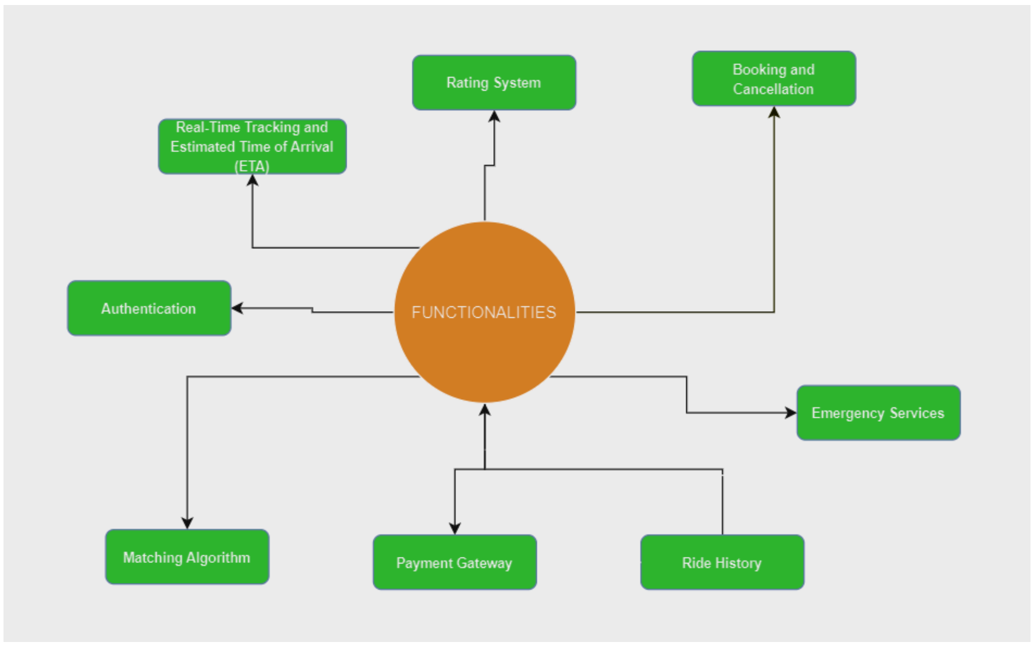
Polymorphism:

- The provide_feedback method is used by both User and Driver, but the content of the feedback differs. This allows the system to handle feedback uniformly while letting each class define its specific behavior.

4) Proposed Key Features of the Product

- **Booking and Cancellation**: Users have the flexibility to book rides and cancel them as needed through a streamlined process.
- **Matching Algorithm**: The app uses a sophisticated matching algorithm to effectively connect users with the nearest available drivers.
- **Real-Time Tracking & ETA**: Provides continuous live tracking of rides and calculates the estimated time of arrival with high accuracy.
- **Payment Integration**: Integrates seamlessly with a payment gateway, ensuring secure and efficient online payment processing.
- **Ride History**: Both users and drivers can view and manage their comprehensive ride history for better tracking and reference.
- **Rating System**: Facilitates a mutual rating system allowing users and drivers to evaluate each other, enhancing service quality.

- **Authentication**: Implements robust authentication mechanisms including email verification, emergency contacts, and two-factor authentication for secure access.
- **Emergency Services**: Features an SOS button for users and drivers to quickly request emergency assistance when needed.
- **Notifications**: Delivers real-time notifications for ride updates, driver arrival, and payment confirmations to keep users informed.
- **User Profiles**: Enables users and drivers to manage and customize their personal profiles and preferences within the app.
- **Support**: Provides dedicated customer support options for resolving issues and addressing inquiries promptly.



Technical Aspects

- **Software Architecture**: Utilizes a microservices architecture to enhance scalability and maintainability by breaking down the system into independently deployable services.
- **Scalability**: Supports independent scaling of microservices based on demand, allowing the system to handle varying loads efficiently.
- **Redundant Mechanisms for Failures**: Implements redundancy and fault tolerance measures to ensure high availability and reliability in case of system failures.
- **Monitoring and Alerting**: Provides comprehensive monitoring with real-time dashboards for continuous system health and performance tracking.

- **Logging and Tracing**: Centralized logging and tracing facilitate effective troubleshooting and performance analysis across the system.

Hardware:

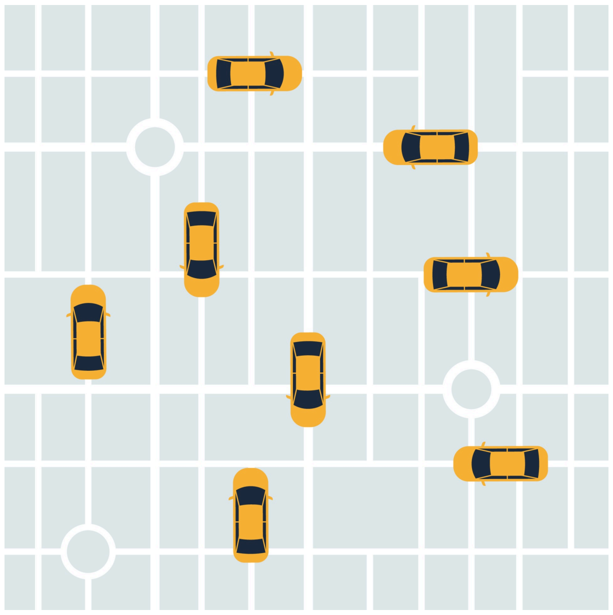
  - **Microservices Deployment**: Deployed across multiple servers to ensure load distribution and redundancy.
  - **Load Balancing**: Employs round-robin load balancing to distribute requests uniformly across servers, optimizing performance and resource utilization.

Detailed Overview

A microservices architecture delivers multiple advantages for a cab booking application, including enhanced scalability, easier maintenance, and the ability to deploy individual services independently.

1. **User Service**:
   - Manages user registration, authentication, profile updates, and user-related data.
   - **Database**: MongoDB, used for storing user profiles and authentication information.
2. **Driver Service**:
   - Oversees driver registration, authentication, availability management, and driver data.
   - **Database**: MongoDB, utilized for driver profiles and availability records.
3. **Ride Service**:
   - Handles ride requests, bookings, cancellations, ride history, and fare computations.
   - **Database**: PostgreSQL, used for storing transactional data related to rides.

4. **Payment Service**:
    - Manages payment processing, transactions, payment methods, and invoice generation.
    - **Integration**: Interfaces with third-party payment gateways.
    - **Database**: MySQL, used for tracking payment transactions and invoices.
5. **Notification Service**:
    - Sends notifications regarding ride status updates, OTPs, and other alerts to users and drivers.
    - **Integration**: Utilizes message queues (RabbitMQ) for asynchronous communication.
    - **Integration**: Interfaces with SMS and email services for notification delivery.
6. **Location Service**:
    - Provides real-time ride tracking, calculates ETAs, and manages GPS data.
    - **Database**: Redis, used for storing real-time location data.
7. **Review Service**:
    - Manages the review and rating system for both users and drivers.
    - **Database**: Elasticsearch, used for indexing and querying review data.
8. **Authentication Service**:
    - Oversees authentication processes, including OTP generation, two-step verification, and security protocols.
    - **Database**: MongoDB, used for storing security tokens and session data.
9. **API Gateway**:
    - Serves as the central entry point for client requests.
    - Routes requests to the appropriate microservices, provides rate limiting, load balancing, and security features.



5) Testing Methods

For the cab booking application, the following testing methods are essential:

Unit Testing:

- Purpose: Validate that individual components and services operate correctly in isolation.
- Tools: JUnit for Java-based services.
- Example: Testing the functionality of the OTP generation in the Authentication Service.

Integration Testing:

- Purpose: Confirm that different services interact and function together as expected.
- Tools: Postman for API testing.
- Example: Verifying the integration between the User Service and Ride Service during the ride booking process.
- Functional Testing:
- Purpose: Ensure the application meets its functional requirements.
- Tools: JUnit for backend functional tests.
- Example: Testing the entire ride booking process, including user registration, ride requests, OTP verification, and payment handling.

Performance Testing:

- Purpose: Evaluate the application’s responsiveness, stability, and scalability under various conditions.
- Types:
- Load Testing: Simulates a high volume of users to test system performance under typical load conditions.
- Stress Testing: Tests the system’s behavior under extreme conditions to determine its breaking point.
- Tools: Apache JMeter.
- Example: Simulating 10,000 simultaneous ride requests to assess system performance and capacity.

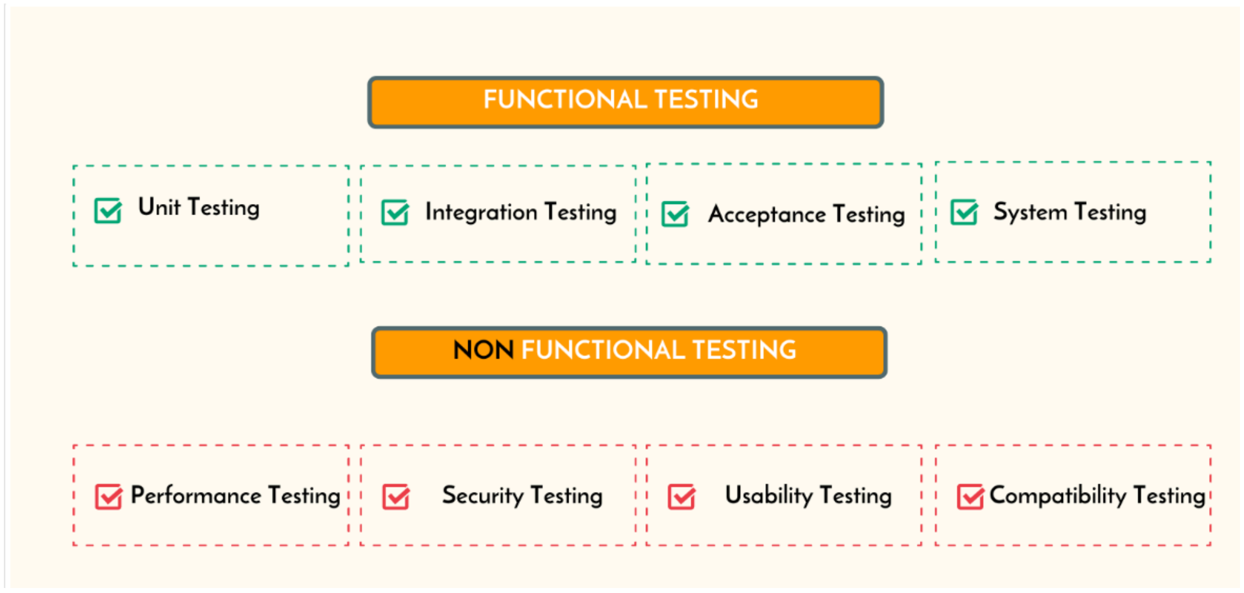
Security Testing:

- Purpose: Detects and addresses security vulnerabilities to ensure the application’s safety.
- Tools: Burp Suite for security assessments.

  - Example: Performing penetration tests to uncover potential security issues, such as SQL injection or cross-site scripting (XSS).

User Acceptance Testing (UAT):

  - Purpose: Validate that the system meets user requirements and is ready for deployment.
  - Conducted By: End-users or clients.
  - Example: End-users test the complete ride booking workflow and provide feedback prior to the final release.



6) Trade-offs in the System

Performance vs. Scalability:

- Decision: We chose a microservices architecture to enhance scalability at the expense of some performance overhead due to inter-service communication.
- Rationale: This decision allows each service to be scaled independently based on demand, improving overall system flexibility and resilience.

Complexity vs. Maintainability:

- Decision: Using multiple microservices increases system complexity but improves maintainability.
- Rationale: Each service is easier to understand, test, and deploy independently, reducing the risk of introducing bugs when making changes.

Consistency vs. Availability:

- Decision: Implement eventual consistency in some parts of the system, such as ride status updates.
- Rationale: This approach improves availability and performance but might lead to temporary inconsistencies, which are acceptable for our use case.

Security vs. Usability:

- Decision: Implementing two-factor authentication for both users and drivers.
- Rationale: Enhances security but adds an extra step in the login process, slightly reducing usability

7) Evaluate and Communicate the Rationale

Performance:

- Evaluate the impact of network latency and inter-service communication overhead in a microservices architecture.
- Communicate the benefits of independent scaling and fault isolation.

Scalability:

- Assess the ability to scale individual services horizontally to handle increased load.
- Highlight the ease of scaling specific parts of the application based on traffic patterns.

Maintainability:

- Consider the ease of updating, testing, and deploying services independently.
- Emphasize the reduced risk of changes affecting unrelated parts of the system.

8) Caching

Integrate Caching Mechanisms

Tools:

- Use Redis for caching frequently accessed data.

Caching Strategies:

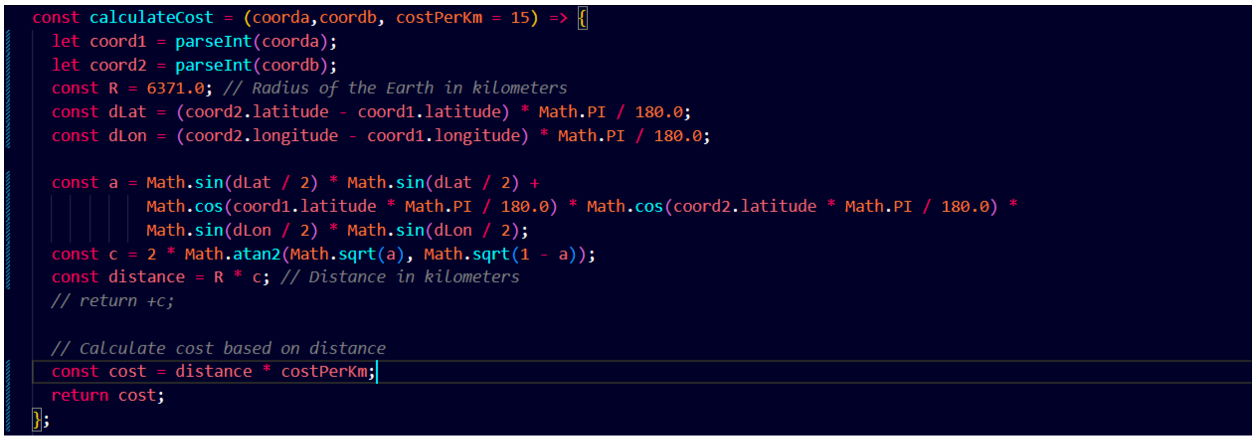
- Cache ride fare calculations, driver availability, and user session data to reduce database load.
- Use content delivery networks (CDNs) to cache static assets, such as images and scripts, for faster content delivery.

9) Implement Cache Eviction Policies

Eviction Policies:

- Least Recently Used (LRU): Evict the least recently used items when the cache reaches its maximum capacity.
- Time-to-Live (TTL): Set expiration times for cached items to ensure stale data is removed.
- Custom Policies: Implement custom eviction strategies based on specific application requirements, such as prioritizing eviction of less critical data

Admin's Cab Allocation Optimization:

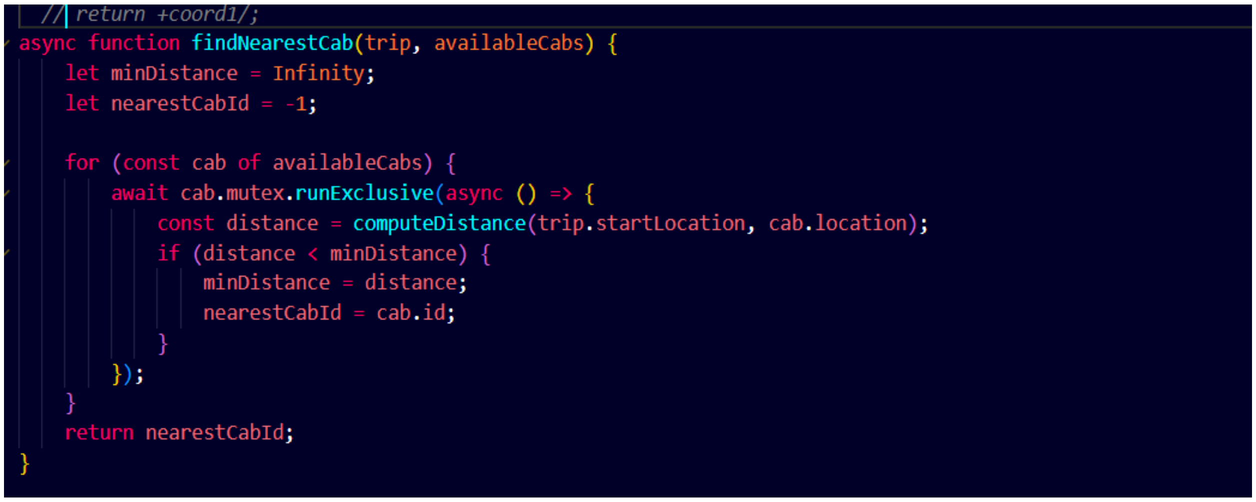


```
const calculateCost = (coorda,coordb, costPerKm = 15) => {
  let coord1 = parseInt(coorda);
  let coord2 = parseInt(coordb);
  const R = 6371.0; // Radius of the Earth in kilometers
  const dLat = (coord2.latitude - coord1.latitude) * Math.PI / 180.0;
  const dLon = (coord2.longitude - coord1.longitude) * Math.PI / 180.0;

  const a = Math.sin(dLat / 2) * Math.sin(dLat / 2) +
            Math.cos(coord1.latitude * Math.PI / 180.0) * Math.cos(coord2.latitude * Math.PI / 180.0) *
            Math.sin(dLon / 2) * Math.sin(dLon / 2);
  const c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1 - a));
  const distance = R * c; // Distance in kilometers
  // return +c;

  // Calculate cost based on distance
  const cost = distance * costPerKm;
  return cost;
};
```

Employee's Cab Search Optimization



```
    // return +coord1/;
async function findNearestCab(trip, availableCabs) {
    let minDistance = Infinity;
    let nearestCabId = -1;

    for (const cab of availableCabs) {
        await cab.mutex.runExclusive(async () => {
            const distance = computeDistance(trip.startLocation, cab.location);
            if (distance < minDistance) {
                minDistance = distance;
                nearestCabId = cab.id;
            }
        });
    }
    return nearestCabId;
}
```