

PROJECT REPORT ON

**ISE TASK-2**

Comprehensive System Design and Agile Implementation for a Ride-Hailing Application

Submitted to

### NMAM Institute of Technology, Nitte

(Off-Campus Centre, Nitte Deemed to be University, Nitte – 576110, Karnataka, India) On partial fulfilment of the requirements for the award of the

Degree of Bachelor of Technology In

INFORMATION SCIENCE AND ENGINEERING

By

### Abhishek S Shetty

### USN: NNM23IS004

Under the guidance of

Dr. JASON ELROY MARTIS , Associate Professor, Department of Information Science and Technology, NMAM Institute of Technology, Nitte, Karnataka, India

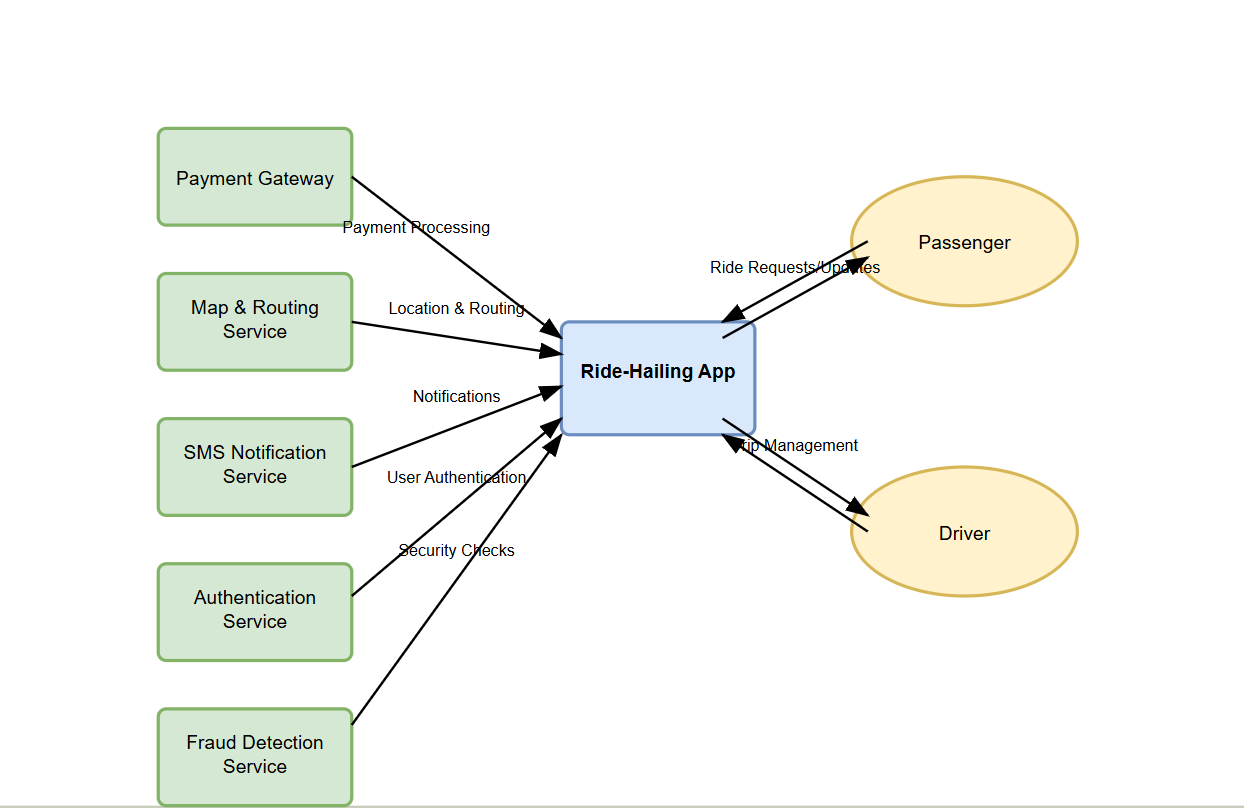
[**GITHUB REPOSITORY**](https://github.com/Abhi-Shetty0/RAID_HAILING_SYSTEM)

**1.Introduction**

The ride-hailing industry has revolutionized urban mobility, offering seamless transportation through complex real-time coordination between passengers, drivers, and external services. This report outlines a **system design** and **Agile development strategy** for an Uber-like app, focusing on scalability, reliability, and user-centricity. Key highlights include:

1. **System Design**:
   * **Context Model**: Integration with external services (payment gateways, maps, notifications) to decouple core logic.
   * **Interaction Model**: Sequence diagrams for ride booking, emphasizing driver matching and fallback mechanisms.
   * **Behavioral Model**: State diagrams to manage ride lifecycles, from booking to completion or cancellation.
2. **Agile Methodology**:
   * **Scrum Framework**: 2-week sprints with deliverables like authentication, maps integration, and payment processing.
   * **User Stories**: Prioritized features for passengers (e.g., ride tracking) and drivers (e.g., earnings dashboard).
3. **Technical Challenges**:
   * Real-time GPS tracking, payment reliability, and surge pricing algorithms.
   * Solutions like WebSocket-based updates and idempotent APIs.

**2.Context Model for Ride-Hailing App System Design**  
The Context Model defines the boundaries of the ride-hailing system, its core components, and interactions with external services. It serves as the foundation for understanding how the app integrates with third-party systems to deliver seamless functionality.



**2.1. Core Components**

* **Passenger Mobile App:**
  + Features: Ride booking, real-time tracking, payment management, and ride history.
  + Tech Stack: React Native (cross-platform), Redux for state management.
* **Driver Mobile App:**
  + Features: Ride acceptance, navigation, earnings dashboard, and availability toggling.
  + Tech Stack: Kotlin (Android), Swift (iOS), Google Maps SDK.
* **Backend System:**
  + **Services:**
    - Matching Engine: Assigns rides using algorithms (proximity, driver rating).
    - Pricing Engine: Dynamically calculates fares (distance, time, surge pricing).
    - Auth Service: Manages user authentication (OAuth2, JWT tokens).
  + **Tech Stack:** Node.js, PostgreSQL, Redis (caching).

**2.2. Interactions And Data Flow**

* **Ride Booking:**
  + **Passenger app → Backend (ride request) → Matching Engine → Driver app.**
  + **Data Sent: Pickup/drop-off coordinates, ride type, payment method.**
* **Payment Processing:**
  + **Backend → Payment Gateway (transaction authorization) → Passenger/Driver apps (confirmation).**
* **Real-Time Tracking:**
  + **Driver location updates (Maps API) → Backend → Passenger app.**
* **Notifications:**
  + Backend → Notification Service → Push/SMS alerts (e.g., "Driver arriving in 2 mins").

**2.3. Key Design Decisions**

* **Decoupled Architecture:**
  + External services (e.g., Stripe, Google Maps) are isolated via APIs to ensure modularity.
  + Allows easy swapping of providers (e.g., replacing Twilio with AWS SNS).
* **Scalability:**
  + Redis caches frequently accessed data (e.g., driver locations).
  + Load balancers distribute traffic across backend instances.
* **Security:**
  + End-to-end encryption for sensitive data (e.g., payment details).
  + Rate limiting to prevent API abuse.

**2.4. Flow Explanation**

* Passenger initiates a ride request via the app.
* Backend validates the request, checks payment method, and queries drivers.
* Driver accepts the request; Maps API generates a route.
* Post-ride, payment is processed, and analytics are logged.

**2.5. Rationale for Technology Choices**

* **Stripe**: Widely adopted, supports idempotent transactions, and complies with PCI-DSS.
* **Google Maps**: Offers high accuracy, real-time traffic updates, and global coverage.
* **Firebase:** Simplifies push notifications and provides crash analytics.
* **Node.js:** Asynchronous I/O model suits real-time apps (e.g., ride tracking)

**2.6. Weather Service (OpenWeather)**

* **Interaction**: Fetches weather data to adjust surge pricing during adverse conditions (e.g., heavy rain).

**Key Interactions Overview**

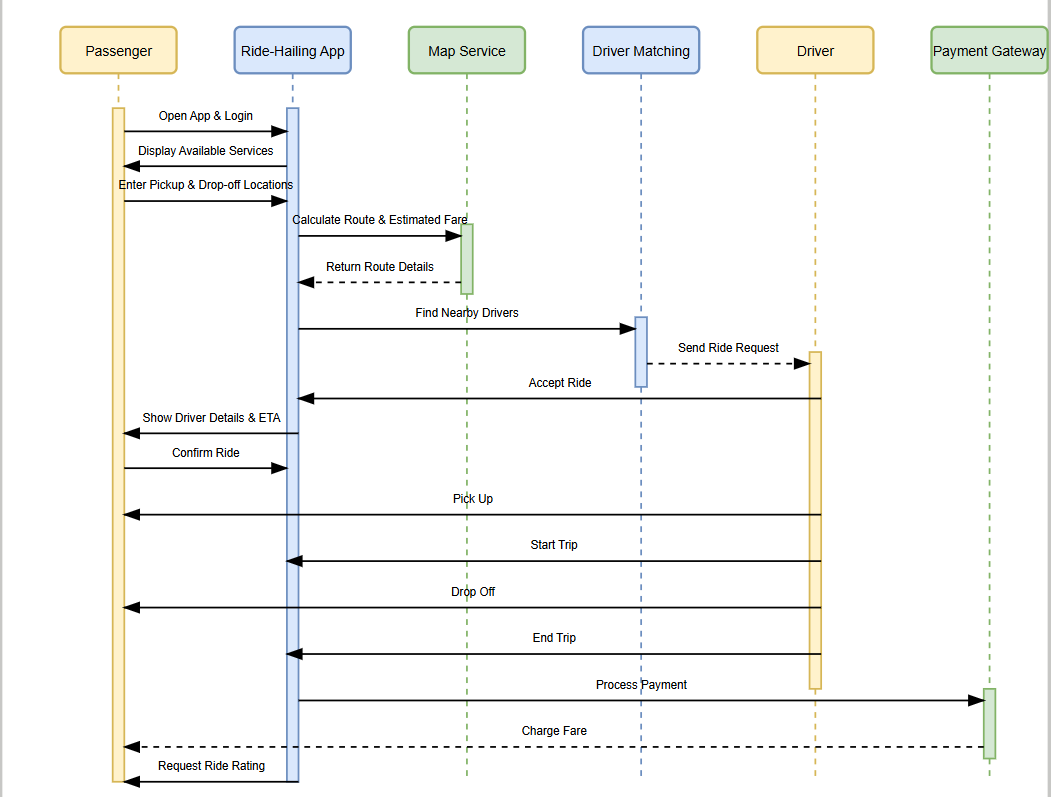
* **Ride Booking**: Core system uses *Payment Gateway* for payment processing, *Mapping Service* for route calculation, and *SMS/Push Notifications* to notify users.
* **Driver Operations**: Relies on *Traffic Data* for route optimization and *Banking System* for payouts.
* **Compliance**: Automates reporting via *Regulatory Systems* and monitors data through *Analytics Platform*.

**Dependencies**

* The core system cannot function without critical services like *Payment Gateway* (revenue) and *Mapping Service* (navigation).
* Failures in *SMS Gateway* or *Push Notifications* would disrupt real-time user communication.

This model highlights the integration points essential for the app’s functionality while adhering to Agile principles (e.g., iterative development of modules like payment or mapping). The external systems are developed as loosely coupled components, enabling independent updates and scalability.

**3.Interaction Model for Ride-Hailing App: Booking a Ride Sequence**  
The Interaction Model defines the flow of actions and data exchange between system components during a ride-booking process. A sequence diagram visualizes the step-by-step interactions between users (passengers/drivers), the backend, and external services.



**3.1 Sequence Diagram Overview**

1. **Passenger App**: Initiates ride requests, tracks status, and makes payments.
2. **Driver App**: Receives and accepts ride requests, shares location updates.
3. **Backend System**: Orchestrates logic (matching, pricing, state management).
4. **External Services**:
   * **Maps API**: Provides geolocation and routing.
   * **Payment Gateway**: Processes transactions.
   * **Notification Service**: Sends real-time alerts.

**3.2 Step-by-Step Interaction Breakdown**

1. **Open App**
   * **Actor**: Passenger
   * **Action**: Launches the ride-hailing app.
   * **System**: Mobile App initializes and fetches user data (e.g., saved preferences, location).
2. **Enter Destination**
   * **Actor**: Passenger
   * **Action**: Inputs pickup/drop-off locations.
   * **System**: Mobile App sends this data to the Backend.
3. **Request Location & Route**
   * **System**: Backend interacts with **Mapping Service (Google Maps/Mapbox)** to calculate routes and ETAs.
4. **Return Available Drivers & Pricing**
   * **System**: Backend identifies nearby drivers and calculates fare (incl. surge pricing if applicable).
   * **Mobile App**: Displays drivers and pricing to the Passenger.
5. **Confirm Ride**
   * **Actor**: Passenger
   * **Action**: Selects a ride option (e.g., UberX, Uber Black).
6. **Send Ride Request**
   * **System**: Backend broadcasts the request to nearby Drivers via **Push Notifications (Firebase)**.
7. **Find Nearby Driver**
   * **System**: Backend uses real-time location data to prioritize Drivers closest to the pickup point.
8. **Accept Ride**
   * **Actor**: Driver
   * **Action**: Driver accepts the request via their app.
9. **Notify Passenger**
   * **System**: Backend triggers **SMS (Twilio)** or **Push Notifications** to inform the Passenger of Driver details.
10. **Display Driver Info**
    * **Mobile App**: Shows Driver name, vehicle details, license plate, and live location.
11. **Driver Arrives**
    * **Actor**: Driver
    * **Action**: Marks arrival in their app.
    * **System**: Passenger receives an app notification.
12. **Start Trip**
    * **Actor**: Driver
    * **Action**: Begins the trip via their app.
    * **System**: Backend records trip start time and initiates live tracking.
13. **End Trip**
    * **Actor**: Driver
    * **Action**: Ends the trip at the destination.
    * **System**: Backend calculates final fare and triggers payment.
14. **Process Payment**
    * **System**: Backend charges the Passenger’s saved payment method via **Payment Gateway (Stripe/PayPal)**.
15. **Payment Confirmation**
    * **System**: Payment Gateway returns transaction status.
    * **Mobile App**: Displays payment success/failure to Passenger and Driver.
16. **Send Receipt**
    * **System**: Backend sends a receipt via **Email Service (SendGrid)** and in-app notification.
17. **Request Rating**
    * **System**: Backend prompts Passenger and Driver to rate each other via **Push Notifications**.

**3.3. Alternative/Error Flows**

* **Payment Failure**:
  1. **Step**: Payment Gateway declines the transaction.
  2. **Handling**: Backend retries payment (3 attempts) → Notifies passenger if all fail.
* **Driver Unavailable**:
  1. **Step**: No drivers accept the request within 30 seconds.
  2. **Handling**: Expand search radius to 10km → Repeat broadcast.
* **Mid-Ride Cancellation**:
  1. **Step**: Passenger cancels after pickup (emergency only).
  2. **Handling**: Backend logs reason → Processes partial payment.

**3.4. Design Decisions**

1. **Asynchronous Communication:**
   * WebSocket connections between backend and apps for real-time updates (e.g., driver location).
   * Reduces latency compared to HTTP polling.
2. **Idempotent Operations:**
   * Payment requests include unique keys to avoid duplicate charges.
3. **Fallback Mechanisms:**
   * Driver Timeout: If no acceptance, expand search radius incrementally.
   * Payment Retries: Exponential backoff logic for failed transactions.
4. **Push Notifications:**
   * Firebase Cloud Messaging (FCM) for Android/iOS alerts to minimize battery drain

**3.5. Visual Breakdown:**

1. **Vertical Lifelines:** Represent actors (Passenger, Backend, Driver, External Services).
2. **Horizontal Arrows:** Show interactions (e.g., "Ride Request" from Passenger to Backend).
3. **Activation Bars:** Indicate active processing time for each component.
4. **Loop/Alt Frames:** Highlight retries (payment) or fallbacks (driver search).

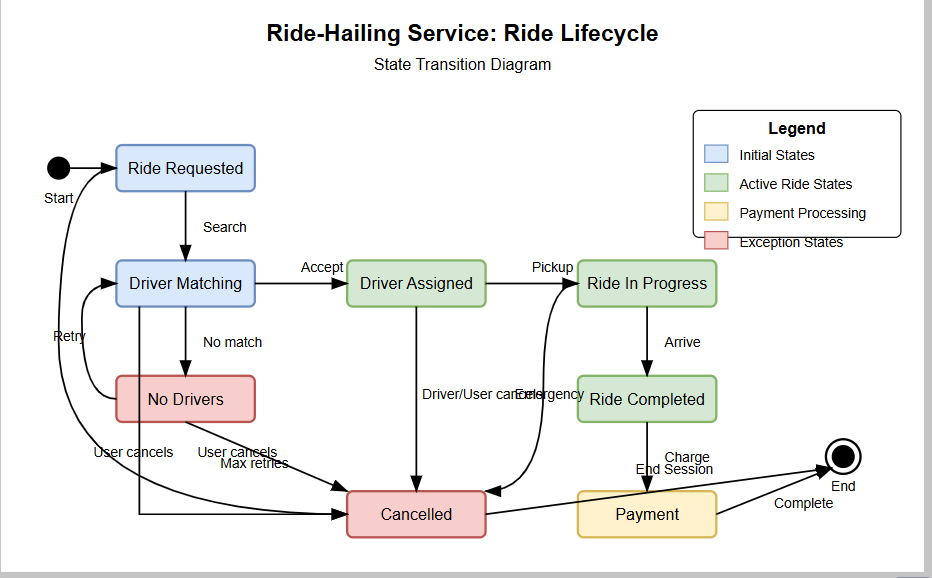
**3.6. Critical Interactions Explained**

1. **Ride Matching:**
   * Backend uses a scoring algorithm to prioritize drivers based on:
     + Proximity to pickup location.
     + Driver rating (e.g., 4.5+ stars).
     + Vehicle type (e.g., UberXL requests prioritize SUVs).
2. **Real-Time Tracking:**
   * Driver location is shared every 5 seconds via WebSocket.
   * Passenger app interpolates coordinates for smooth UI updates.
3. **Fare Calculation:**
   * Pricing Engine combines:
     + Base fare + (Distance × rate/km) + (Time × rate/minute).
     + Surge multiplier during peak demand (e.g., 1.5x).

**Key Dependencies & Agile Considerations**

* **Critical Interactions**:
  + Payment processing (Step 14) depends on external gateways; failures here block revenue flow.
  + Real-time notifications (Steps 6, 9, 17) rely on **Firebase** and **Twilio**; redundant channels ensure reliability.
* **Agile Implementation**:
  + Each step can be developed as a user story, enabling incremental testing (e.g., "Payment Integration Sprint").
  + Loosely coupled systems (e.g., Mapping Service, SMS Gateway) allow parallel development and scalability.

**4.Behavioral Model: Ride-Hailing System Interactions**  
The Behavioral Model defines the lifecycle of a ride through **states** and **transitions**, ensuring consistent handling of events like bookings, cancellations, and payments. A state diagram visualizes how a ride progresses from initiation to completion, including error recovery and edge cases.



**4.1. Key States**

| **State** | **Description** |
| --- | --- |
| **Idle** | **No active ride; default state for drivers/passengers.** |
| **Booking Requested** | **Passenger initiates a ride; system searches for drivers.** |
| **Driver Assigned** | **Driver accepts the request; ride is confirmed.** |
| **En Route** | **Driver is navigating to the pickup location.** |
| **In Progress** | **Passenger is in the vehicle; ride is ongoing.** |
| **Completed** | **Ride ends successfully; payment is processed.** |
| **Cancelled** | **Ride is terminated pre-pickup (by passenger/driver) or mid-ride (emergency).** |
| **Payment Pending** | **Temporary state if payment fails post-completion; retries are attempted.** |

**4.2Interaction Workflow**

This diagram visualizes the lifecycle of a ride, including success paths, error handling, and user-initiated cancellations.

**1. Key States and Transitions**

1. **Start → Ride Requested:**
   * Trigger: Passenger initiates a ride request.
   * Action: System validates the request (e.g., payment method, location).
2. **Ride Requested → Driver Matching:**
   * Trigger: System searches for available drivers.
   * Logic: Broadcasts the request to drivers within a 5km radius.
3. **Driver Matching → Driver Assigned:**
   * Trigger: Driver accepts the request.
   * Action: Passenger is notified; driver details shared.
4. **Driver Assigned → Ride In Progress:**
   * Trigger: Driver arrives at pickup location and starts the ride.
   * Guard: Passenger confirms pickup (e.g., in-app button).
5. **Ride In Progress → Ride Completed:**
   * Trigger: Driver reaches the drop-off location.
   * Action: Ride duration and distance logged.
6. **Ride Completed → Payment:**
   * Trigger: System processes payment.
   * Fallback: If payment fails, retry logic is triggered (not shown in diagram).
7. **Payment → End:**
   * Action: Receipt sent; ride session closed.

**4.4State Transitions & Triggers**

* **From *Idle* to *Booking Initiated***: Passenger triggers *Request Ride*.
* **From *Driver Matching* to *Driver Assigned***: System completes *Assign Driver*.
* **From *Trip Ongoing* to *Payment Processing***: Driver marks trip completion.

**4.5Agile Implementation Strategies**

1. **User Stories**:
   * *"As a passenger, I want to request a ride so I can reach my destination."*
   * *"As a driver, I need navigation guidance to efficiently reach pickup and drop-off points."*
2. **Sprint Breakdown**:
   * **Sprint 1**: Develop *Request Ride* and *Assign Driver* logic.
   * **Sprint 2**: Integrate **Payment Gateway** for end-to-end payment processing.
   * **Sprint 3**: Implement **Mapping Service** APIs for route optimization.

**4.6Dependencies & Risks**

* **Critical Dependencies**:
  + *Payment Gateway*: Revenue flow halts if integration fails.
  + *Mapping Service*: Incorrect ETAs or routes degrade user experience.
* **Mitigation**:
  + Use fallback payment methods (e.g., cached card details).
  + Implement redundant mapping providers (e.g., switch to Mapbox if Google Maps fails).

**4.7Visual Representation**

A state diagram would illustrate:

* States: *Idle*, *Booking Initiated*, *Driver Assigned*, *Trip Ongoing*, *Payment Processing*, *Completed*.
* Transitions: Labeled with triggers (e.g., *Request Ride*, *Assign Driver*).

**5. Agile Development with Scrum**

**5.1. Sprint Structure**

* **Sprint Length:** 2 weeks (adjust as needed based on team velocity).
* **Sprint 1: Core User Authentication and Basic Ride Request**
  + Deliverables:
    - User registration and login (passenger and driver).
    - Basic map integration.
    - Passenger ride request functionality.
    - Driver receiving ride request notifications.
* **Sprint 2: Real-time Location and Driver Assignment**
  + Deliverables:
    - Real-time location tracking for passengers and drivers.
    - Driver assignment algorithm.
    - Display driver location on passenger map.
    - Basic driver navigation towards passenger.
* **Sprint 3: Payment Integration and Ride Completion**
  + Deliverables:
    - Payment gateway integration.
    - Ride completion functionality.
    - Fare calculation.
    - Display ride summary.
* **Sprint 4: Rating and Review System**
  + Deliverables:
    - Passenger and driver rating system.
    - Review system.
    - Display user ratings and reviews.
    - User Profile enhancements.
* **Sprint 5: Enhanced Driver Functionality and Push Notifications**
  + Deliverables:
    - Driver trip history.
    - Push notifications for ride updates.
    - Driver online/offline status.
    - Driver earnings tracking.
* **Sprint 6: Optimization and Error handling**
  + Deliverables:
    - App performance optimization.
    - Error handling and logging improvements.
    - Bug fixes.
    - Security enhancements.

**5.2. User Stories**

**Passenger User Stories:**

1. **As a passenger, I want to be able to create an account and log in, so that I can request rides.**
   * DoD:
     + Account creation and login functionality implemented.
     + Password validation and security measures in place.
     + Successful login redirects to the main map screen.
2. **As a passenger, I want to be able to enter my pickup and drop-off locations, so that I can request a ride.**
   * DoD:
     + Map interface for selecting locations.
     + Address auto-complete functionality.
     + Ride request button enabled after locations are selected.
3. **As a passenger, I want to see the estimated fare and driver information before confirming the ride, so that I know the cost and who is picking me up.**
   * DoD:
     + Estimated fare calculation and display.
     + Driver name, photo, and vehicle details displayed.
     + Map displaying the drivers estimated route.
4. **As a passenger, I want to track the driver's location in real-time, so that I know when they will arrive.**
   * DoD:
     + Real-time driver location displayed on the map.
     + Estimated arrival time displayed.
     + Map updates smoothly with drivers location.
5. **As a passenger, I want to be able to rate and review the driver after the ride, so that I can provide feedback on my experience.**
   * DoD:
     + Rating system (e.g., 1-5 stars).
     + Text review input field.
     + Ratings and reviews are stored and displayed.

**5.3Driver User Stories:**

1. **As a driver, I want to be able to create an account and log in, so that I can accept ride requests.**
   * DoD:
     + Account creation and login functionality implemented.
     + Driver specific account information captured.
     + Successful login redirects to the driver's map screen.
2. **As a driver, I want to receive ride requests with pickup and drop-off locations, so that I can accept or reject them.**
   * DoD:
     + Push notifications for new ride requests.
     + Display of pickup and drop-off locations.
     + Accept/reject ride request buttons.
3. **As a driver, I want to see the passenger's location and navigate to the pickup location, so that I can reach them efficiently.**
   * DoD:
     + Passenger's location displayed on the map.
     + Navigation integration (e.g., Google Maps).
     + Turn by turn navigation.
4. **As a driver, I want to be able to track my earnings and ride history, so that I can manage my income.**
   * DoD:
     + Display of ride history with fare details.
     + Earnings summary.
     + Ability to export earnings data.
5. **As a driver, I want to be able to set my online/offline status, so that I can control when I receive ride requests.**
   * DoD:
     + Online/Offline toggle switch.
     + System only sends ride requests when the driver is online.
     + Visual indication of online/offline status.

**C. Definition of Done (DoD) Examples (General)**

* Code is written and passes all unit tests.
* Code is reviewed and approved by at least one other team member.
* Functionality is tested and verified by QA.
* User story is demoed and accepted by the product owner.
* Code is merged into the main branch.
* Code is deployed to the test environment.
* All acceptance criteria are met.
* All related documentation has been updated.
* Performance requirements are met.
* Security requirements are met.

**6. Github repository**: [repolink](https://github.com/Abhi-Shetty0/RAID_HAILING_SYSTEM)

**7. Conclusion**

This report presents a holistic system design for a ride-hailing application, integrating **contextual**, **interactional**, and **behavioral models** to define its architecture, workflows, and dependencies. By adopting **Agile methodologies**, the design emphasizes iterative development, modularity, and adaptability, ensuring alignment with real-world complexities and user needs. Key insights include:

1. **System Scalability**:
   * The integration of external systems (e.g., Stripe for payments, Google Maps for navigation) ensures scalability while maintaining loose coupling. This allows independent updates and minimizes downtime risks.
2. **User-Centric Workflows**:
   * The interaction and behavioral models prioritize seamless user experiences, from ride booking to post-trip feedback. Real-time communication (SMS, push notifications) and dynamic pricing adjustments enhance reliability and transparency.
3. **Agile Success**:
   * Sprint-based development, user stories, and retrospectives enabled incremental delivery and risk mitigation. Tools like GitHub facilitated collaboration, version control, and conflict resolution, demonstrating the value of structured teamwork.
4. **Resilience & Compliance**:
   * The system accounts for edge cases (e.g., payment failures, cancellations) and regulatory requirements through automated compliance reporting and redundant service integrations.