LessGo:
Safe, Smart, and Cost-Saving Carpooling Platform

Project Workbook

By
Johnny To
Shyam Kannan
Spencer Davis
Sri Ram Mannam

October 30, 2025

Advisor: Dr. Younghee Park

[

The **Project Workbook** is used to collect research, proposals, requirements, architecture, design, implementation, and project planning information. This assignment is not intended to be used as a writing assignment but as a technical assignment that collects a vast amount of information about your project. You will ultimately select a subset of the information from this workbook to include in your Project Report. The Project Report will require strict attention to the detail required for a writing assignment.

The information collected in this workbook includes a problem description, a state-of-the-art description, literature search, project justification, requirements, architecture, design (including tradeoffs, UML artifacts, UI mockups, and database design), QA, performance validation planning, implementation/prototype development, project management planning (task assignment and schedule), and conclusions. This information is organized in nine chapters of this document. You can add additional chapters if you think your project has other information that needs to be collected.

Information in this document will be collected and evaluated across two assignments in CMPE 295A (referred to in this document as Workbook Assignment 1 and 2). The chapter descriptions below indicate which of the chapters are evaluated in each of the workbook assignments. You should continue to use this workbook as a collection point for information related to your project until your project is completed.

All members of your project team should contribute to this document. Each member of your project team should identify their individual contributions in this document.

]

1. Literature Search, State of the Art

Literature Search

Carpooling Systems employ a matching algorithm that is used to assign riders and drivers together. This problem is crucial to ensure a quick and optimal route for riders and to minimize the unnecessary miles that the driver has to cover. As a result, a significant amount of research is involved in developing novel algorithms and models to perform such tasks. In the carpooling space, there are 2 methods of carpooling: static carpooling and dynamic carpooling [1]. In static carpooling, riders and drivers are established ahead of time, whereas dynamic carpooling involves selecting riders or drivers in real-time in an ad-hoc manner [1][2]. Graziotin [1] discusses the issues of the adoption of dynamic ridesharing systems and calls for further research in various topics, including matching algorithms.

To improve the speed of matches and the system's efficiency in selecting rider and driver pairs, researchers have pursued two methods to optimize: (i) reducing candidate sets before full scoring, and (ii) optimizing route planning once candidates are identified [3]. Research has been conducted to develop algorithms for mining GPS trajectories to facilitate intelligent routing [2], create data-driven matching heuristics for pooling [4], and formalize user preference modeling to enhance match acceptance and satisfaction [8].

Recent research has also been done to incorporate machine learning techniques into the matching algorithm. Specifically, representation learning and reinforcement learning [3][5][6][7]. Papers [3] and [5] use representation learning to embed orders, vehicles, and contextual signals to refine candidate pools. Papers [6] and [7] explore reinforcement learning to make sequential, system-level assignment decisions under uncertainty and multi-objective trade-offs.

Cost algorithms are central to the design of carpooling platforms because they define how travel costs are allocated between drivers and riders. A primary challenge is balancing transparency and fairness for riders while maintaining adequate incentives for drivers to participate. Early approaches to cost sharing often relied on static methods where costs were split evenly or in proportion to distance traveled, but these models overlook important factors such as detours, waiting time, or variations in user flexibility [9][10].

Researchers have also investigated methods to incorporate route optimization directly into fare allocation. By reorganizing pickup and drop-off sequences, systems can reduce unnecessary mileage while assigning higher costs to riders who create larger detours. Linking fares to detour ratios in this way ensures that disruptions to the route are fairly compensated [10][12]. Other models focus on temporal flexibility, offering small discounts to passengers willing to shift departure times slightly. These incentives help smooth demand, increase vehicle occupancy, and improve driver earnings without relying on surge pricing [11].

More advanced frameworks apply multi-objective optimization and reinforcement learning to cost allocation. Multi-objective models balance minimizing operating costs with

maintaining service quality, often resulting in flat seat pricing schemes that give riders certainty while still covering driver minimum thresholds [12]. Reinforcement learning approaches, such as DeepPool, treat vehicles as agents that learn when to accept new riders or reposition themselves to maximize occupancy. By rewarding high utilization and penalizing idle driving, these systems continuously improve cost efficiency, leading to lower average fares and greater revenue stability for drivers [13].

Recent advancements in ride-sharing algorithms have focused on scalable and efficiency-driven solutions for dynamic urban carpooling platforms. Cao et al. [14] introduced the SHAREK system, which utilizes a three-phase pruning algorithm that efficiently matches riders and drivers by filtering candidate pools using spatial indexing and skyline logic, thereby reducing computational overhead and supporting richer matching variables for scalability. Wang et al. [15] developed a Demand-Supply Balance Score (DSBScore), incorporating future demand prediction to enable proactive regional request assignment and optimal driver routing, significantly improving long-term fleet efficiency.

Tong et al. [16] proposed a dynamic programming-based insertion strategy, enabling scalable real-time route updates for large-scale shared mobility systems. Yuen et al. [17] designed algorithmic models that predict the likelihood of future pick-up requests, maximizing compatible rider aggregation and departing from traditional shortest-path paradigms. Ta et al. [18] presented an efficient ride-sharing framework that maximizes the percentage of shared routes, supporting both join-based and search-based algorithms for optimal matching and coverage. These works collectively highlight the importance of demand forecasting, spatial optimization, and computational scalability in the design and implementation of modern ride-sharing platforms, driving improvements in occupancy rates, route efficiency, and system-wide adaptability.

Many dangers of ridesharing are pretty clear to see; getting into a stranger's car will always involve some amount of risk, as will accepting a stranger into your own car. For this reason, there have been many actions taken to improve rider-driver verification and accountability [19]. Studies have also been conducted regarding general driver health and safety and provide input on ways they could best benefit from the platforms, including occupational health and safety programs designed for them and greater agency on rides that they give [20][21].

For carpooling specifically, some unique privacy and safety issues are more relevant than in private rideshares. Riding with potentially several additional strangers means it is that much harder to thoroughly vet each individual in the ride, which, according to [22], is one of the major holdups that people have with adopting carpool ridesharing, along with convenience and privacy. Furthermore, to calculate your route into a carpool, your location and routing data would be in a fairly accessible state for many more people than normal, which could be a major privacy concern if the proper steps are not taken. In order to mitigate this, [23] recommends the use of homomorphic encryption, which improves security by performing mathematical operations on the encrypted data.

State-of-the-Art Summary

In the current industry, two popular ridesharing systems exist: Uber and Lyft. Both of these systems optimize for single parties with multi-rider trips as a secondary focus. These ridesharing services involve simple pricing calculations depending on distance and time, with small detour adjustments. This pricing system is also very opaque and difficult to determine if a fair price is being calculated. Our smart carpool matching platform builds upon preexisting ridesharing systems by focusing on three concepts.

Safety Validation System

- Using anomaly detection algorithms, we plan to implement a real-time safety check of active trips. These safety checks allow riders to feel safer during their trip.

ML Enhanced Grouping Systems

- Utilizing machine learning techniques to reduce the size of candidate sets allows for quicker and higher-quality groupings between riders and drivers.

Fair Cost Allocation Algorithms

- Incorporating detours and disruptions to routes allows for a fairer estimate of cost per rider.
- We also would like to implement multi-factor considerations when estimating cost to be fair for drivers as well (Car MPG, estimated wear, etc.)

Privacy using Encryption

- Homomorphic encryption (HE) is used to improve privacy throughout the system, using addition and multiplication to protect data.
- 3 main types of HE
    - Partial homomorphic encryption, which allows either multiplication or addition, but not both at the same time
    - Somewhat homomorphic encryption, which allows both but limits the number of times they can be performed
    - Fully homomorphic encryption, which allows both addition and multiplication an unlimited number of times

References

[1] D. Graziotin, “An Analysis of issues against the adoption of Dynamic Carpooling,” doi: https://doi.org/10.48550/arXiv.1306.0361. [Online]. Available: https://arxiv.org/abs/1306.0361.

[2] W. He, K. Hwang, D. Li, "Intelligent Carpool Routing for Urban Ridesharing by Mining GPS Trajectories," *IEEE Trans. Intell. Transp. Sys.*, vol. 15, no. 5, pp. 2286–2296, May 2014, doi: 10.1109/TITS.2014.2315521.

[3] Z. Liu, J. Lin, Z. Xia, et al., "Towards Efficient Ridesharing via Order-Vehicle Pre-Matching Using Attention Mechanism," *2024 IEEE Inter. Conf. Data Mining*, Abu Dhabi, United Arab Emirates, 2024, pp. 141–150, doi: 10.1109/ICDM59182.2024.00033.

[4] A. Şahin, İ. Sevim, E. Albey, M. G. Güler," A data-driven matching algorithm for ride pooling problem," *Comput. & Oper. Res.*, vol. 140, Apr. 2022, Art. no. 105666, doi: https://doi.org/10.1016/j.cor.2021.105666.

[5] L. Tang, Z. Liu, Y. Zhao, Z. Duan, and J. Jia, "Efficient Ridesharing Framework for Ride-matching via Heterogeneous Network Embedding," *ACM Trans. Knowl. Disc. Data*, vol. 14, no. 3, pp. 1–24, Jun. 2020, doi: https://doi.org/10.1145/3373839.

[6] Y. Liang, "Fairness-Aware Dynamic Ride-Hailing Matching Based on Reinforcement Learning," *Electron.*, vol. 13, no. 4, p. 775, Feb. 2024, doi: https://doi.org/10.3390/electronics13040775.

[7] H. Abdelmoumene, S. Boussahoul, "Towards Optimized Dynamic Ridesharing System Through Multi-Objective Reinforcement Learning," in *2024 IEEE Inter. Multi. Conf. Smart Sys. Green Process*, Djerba, Tunisia, Oct. 31–Nov. 3, 2024, pp.1–6, doi: 10.1109/IMC-SSGP63352.2024.10919539.

[8] Z. Dastani, H. Koosha, H. Karimi, and A. M. Moghaddam, "User preferences in ride-sharing mathematical models for enhanced matching," *Sci. Rep.*, vol. 14, Nov. 2024, Art. no. 27338, Nov. 2024, doi: 10.1038/s41598-024-78469-1.

[9] S. Hu, M. M. Dessouky, N. A. Uhan, and P. Vayanos, "Cost-sharing mechanism design for ride-sharing", *Transp. Res. Part B: Methodological*, vol. 150, pp. 410–434, Aug. 2021, ISSN 0191-2615, doi: 10.1016/j.trb.2021.06.018.

[10] Y. Cao, S. Wang, and J. Li, "The Optimization Model of Ride-Sharing Route for Ride Hailing Considering Both System Optimization and User Fairness," *Sustainability*, vol. 13, no. 2, p. 902, Jan. 2021, doi: 10.3390/su13020902.

[11] E. Yuan and P. Van Hentenryck, "Real-Time Pricing Optimization for Ride-Hailing Quality of Service" in *Proc. of the 30th Int. Joint Conf. on Artif. Intell.*, Main Track, pp. 3742–3748, doi: 10.24963/ijcai.2021/515.

[12] M. Zheng and G. Pantuso, "Trading off costs and service rates in a first-mile ride-sharing service," *Transp. Res. Part C: Emerg. Technol.*, vol. 150, 2023, 104099, ISSN 0968-090X, doi: 10.1016/j.trc.2023.104099.

[13] A. Alabbasi, A. Ghosh, and V. Aggarwal, “DeepPool: Distributed Model-free Algorithm for Ride-sharing using Deep Reinforcement Learning,” *IEEE Trans. on Intell. Transp. Syst.*, vol. 20, no. 12, pp. 4714–4727, Dec. 2019. doi: 10.1109/TITS.2019.2931830.

[14] B. Cao, L. Alarabi, M. F. Mokbel, and A. Basalamah, "SHAREK: A Scalable Dynamic Ride Sharing System," in *2015 16th IEEE Inter. Conf. Mobile Data Manage.*, Pittsburgh, PA, USA, 2015, pp. 4–13, doi: 10.1109/MDM.2015.12.

[15] J. Wang, P. Cheng, L. Zheng, et al, “Demand-aware route planning for shared mobility services,” in *Proc. VLDB Endow. 13*, 2020, pp. 979–991, doi: 10.14778/3384345.3384348.

[16] Y. Tong, Y. Zeng, Z. Zhou, L. Chen, J. Ye, and K. Xu, “A unified approach to route planning for shared mobility,” in *Proc. VLDB Endow. 11,* 2018, pp. 1633–1646, doi: 10.14778/3236187.3236211.

[17] C. Yuen, A. Singh, S. Goyal, S. Ranu, and A. Bagchi, “Beyond Shortest Paths: Route Recommendations for Ride-sharing,” in *WWW '19: The World Wide Web Conf.*, pp. 2258–2269, doi: 10.1145/3308558.3313465.

[18] N. Ta, G. Li, T. Zhao, J. Feng, H. Ma and Z. Gong, "An Efficient Ride-Sharing Framework for Maximizing Shared Routes," in *2018 IEEE 34th Int. Conf. on Data Eng. (ICDE)*, Paris, France, 2018, pp. 1795–1796, doi: 10.1109/ICDE.2018.00255.

[19] H. Sutton, “Improve Rideshare Safety for Students by Sharing Strategies, Resources,” Stud. Affairs Tod., vol. 25, no. 12, pp. 1,5, Mar. 2023. [Online]. Available: https://research-ebsco-com.libaccess.sjlibrary.org/linkprocessor/plink?id=721f3dec-57a3-374b-90d1-7ef4992e7f27.

[20] P. Louzado-Feliciano et al., “Characterizing the Health and Safety Concerns of U.S. Rideshare Drivers: A Qualitative Pilot Study,” *Sage J.*, vol. 70, no. 7, Apr. 2022, doi: 10.1177/21650799221076873.

[21] M. Y. Almoqbel and D. Y. Wohn, “Individual and Collaborative Behaviors of Rideshare Drivers in Protecting their Safety,” *Assoc. Comput. Mach*., vol. 3, no. CSCW, pp. 1–21, Nov. 2019, doi: 10.1145/3359319.

[22] R. Gangadharaiah et al., “The Development of the Pooled Rideshare Acceptance Model (PRAM),” *Proquest*, vol. 9, no. 3, 2023, doi: 10.3390/safety9030061.

[23] D. Palma, P. L. Montessoro, M. Loghi, and D. Casagrande, “A Privacy-Preserving System for Confidential Carpooling Services Using Homomorphic Encryption,” *Adv. Intell. Syst.*, vol. 7, no. 5, Mar. 2025, doi: 10.1002/aisy.202400507.

2. Project Justification

This semester's surge in campus enrollment has intensified transportation challenges, leading to overcrowded parking garages and mounting frustration among students and staff. The increased volume of single-occupancy vehicles leads to traffic congestion and unnecessary carbon emissions, resulting in delays and a diminished overall campus experience. These issues underscore the pressing need for a more efficient, community-based commuting solution specifically designed to university environments.

LessGo addresses these issues by intelligently pairing commuters with shared rides, helping to reduce the number of vehicles vying for limited campus parking spaces. By optimizing routes, facilitating real-time scheduling, and enabling users to connect based on overlapping travel needs, the platform aims to save time, promote sustainability, and encourage a sense of community. Its innovative algorithms and notification features offer distinct advantages over traditional ride-sharing services, which often overlook the specific demands of campus populations.

Technically, the platform leverages scalable cloud-native architecture, secure user authentication, and rapid matchmaking to handle high volumes typical during peak enrollment. Its user-centric design and advanced features ensure practical viability for university settings. By directly targeting parking congestion and fostering sustainable commuting, this project promises immediate and ongoing benefits for both campus stakeholders and the broader commuter community.

3. Project Requirements

Requirement

3.1 User Stories

3.1.1 User Stories: Rider

- As a Rider, I want to hail a ride so that I can get to my destination
- As a Rider, I want to schedule a ride so that I don't have to constantly hail a random ride
- As a Rider, I want to find a driver so that I can ___
- As a Rider, I want safety features (TBD) so that I feel safe during a drive.
- As a Rider, I want to share my trip status with friends/family for safety.
- As a Rider, I want to track my ride in real time so that I know my driver's location and ETA.
- As a Rider, I want to rate my driver and provide feedback after the ride so that service quality is maintained.
- As a Rider, I want to see an estimated fare before confirming my ride so that I can make an informed decision.

3.1.2 User Stories: Driver

- As a Driver, I want to accept or reject ride requests so that I can control my schedule.
- As a Driver, I want to see the rider's rating before accepting a ride so that I can ensure safety and reliability.
- As a Driver, I want to get navigation and directions to the rider's pickup and drop-off location.
- As a Driver, I want to set my availability status (online/offline) so that I can take breaks as needed.
- As a Driver, I don't want the route to make me late for class

3.2.3 User Stories: Admin & System

- As an Admin, I want to monitor active rides in real time to respond to emergency issues.
- As an Admin, I want to review reports and block users/drivers involved in reported incidents.
- As an Admin, I want to manage promotions, discounts, and referral programs.

3.2 Essential Requirements

User Registration and Authentication are essential for the creation of an account and for secure sign-in of the user. A registered user should be able to update/edit their profile and interests on their profile anytime they want, which in turn helps improve match quality. A registered user should be able to find optimal carpool matches based on routes, timing, and preferences so they can share rides conveniently. A user should be able to schedule carpool rides so they can plan their commute.

3.3 Desired & Optional Requirements

3.3.1 Desired Features

A user of the app would want to receive real-time notifications/updates about ride status, cancellations, or changes. A user of the driver app would want an optimized route that reduces detours and minimizes travel time for all the passengers and the driver. At the end of each ride, users would like to be able to rate and review carpool partners to foster a safe and reliable community.”

3.3.2 Optional Features

It'll be advantageous to integrate the app with Public Transit APIs**,** for the user to see public transit options alongside carpooling suggestions, or to find connecting travel options. We could introduce a Gamification aspect (Badges, Leaderboards) to encourage users to earn badges for frequent usage, timely arrivals, or eco-friendly rides.

3.4 Non-Functional Requirements

A good measure for the required performance of the platform could be grouping users in less than 5 seconds. The system must be scalable to handle up to 100,000 concurrent users during peak hours. User data must be secured by encrypting it at rest and in transit. To ensure reliability, 99.9% uptime is expected outside of scheduled maintenance. The applications should be easy to use, such that new users can successfully create an account and book a ride within five minutes.

4. Dependencies and Deliverables

Dependencies

The success of LessGo depends heavily on a reliable web and mobile application framework. The front-end will be developed with React so that students can easily register, log in, and browse or post rides in a user-friendly interface. The back-end will likely utilize Node.js with Express to manage account authentication, trip postings, and the core logic that matches riders to drivers based on their class schedules and overlapping routes.

A properly configured database system is also needed. PostgreSQL or MongoDB are good choices as they are essential for storing user information, trip schedules, route details, and completed transactions. A faulty database will immediately halt the system, since grouping and cost sharing cannot happen without this data.

For deployment, a cloud hosting environment like AWS can be used to maintain uptime, whereas Docker could be used to ensure consistent builds across production environments.

Another critical dependency is access to routing and mapping API's. Services such as Google Maps or Mapbox will be integrated to calculate trip distances, estimate detours, and display clear pick-up and drop-off routes to users. Without these API's, the system would be limited to static distance calculations and would not be able to adapt well to real-time scenarios.

Finally, the platform requires a secure payment processing service to enable cost-sharing between riders and drivers. Services such as Stripe or PayPal will handle fund transfers, enforce transaction security, and log payment histories.

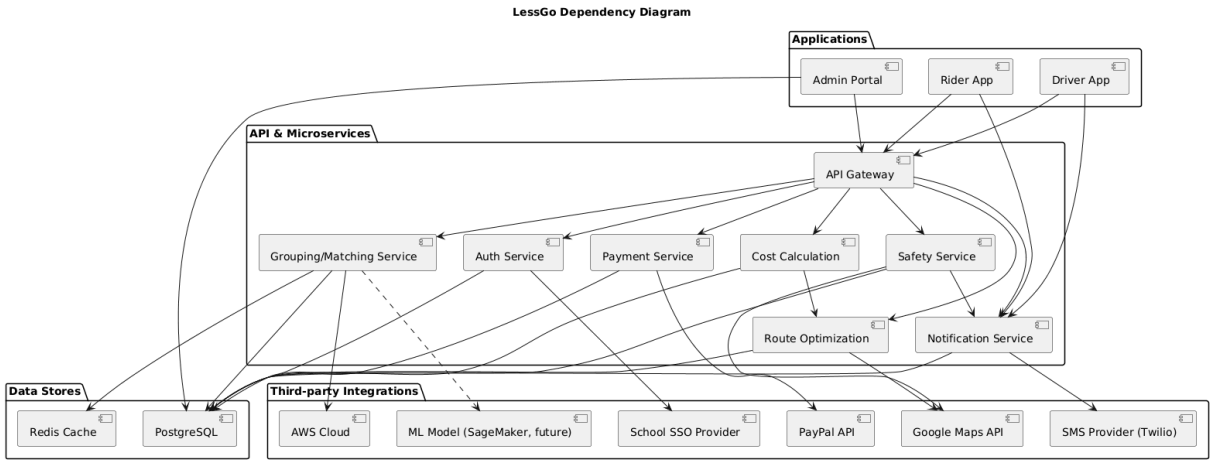


Figure 1. Dependency Diagram

Deliverables

The first deliverable is a functional web and mobile application that allows students to post their available trips or request rides based on class schedules and locations. The application will include features for route browsing, schedule alignment, and driver selection according to user preferences.

A robust back-end system will also be delivered, capable of managing the core grouping logic, handling schedule and route comparisons, and implementing the cost-sharing algorithms identified in the literature search. This back end will provide APIs for front-end interactions and will be built to ensure responsiveness and scalability as the platform grows.

The project will deliver a database schema and integration modules designed to manage user profiles, ride postings, payments, and historical ride data. Integration with mapping APIs will allow the system to calculate distances and optimize routes beyond simple address grouping, ensuring accuracy in both trip planning and cost allocation.

Another key deliverable is the cost-sharing and payment module, which will allow students to split ride costs fairly. This includes implementation of at least one algorithmic approach, such as the “quote never increases” model, so that students can see transparent pricing while ensuring drivers always receive a fair minimum payout.

5. Project Architecture

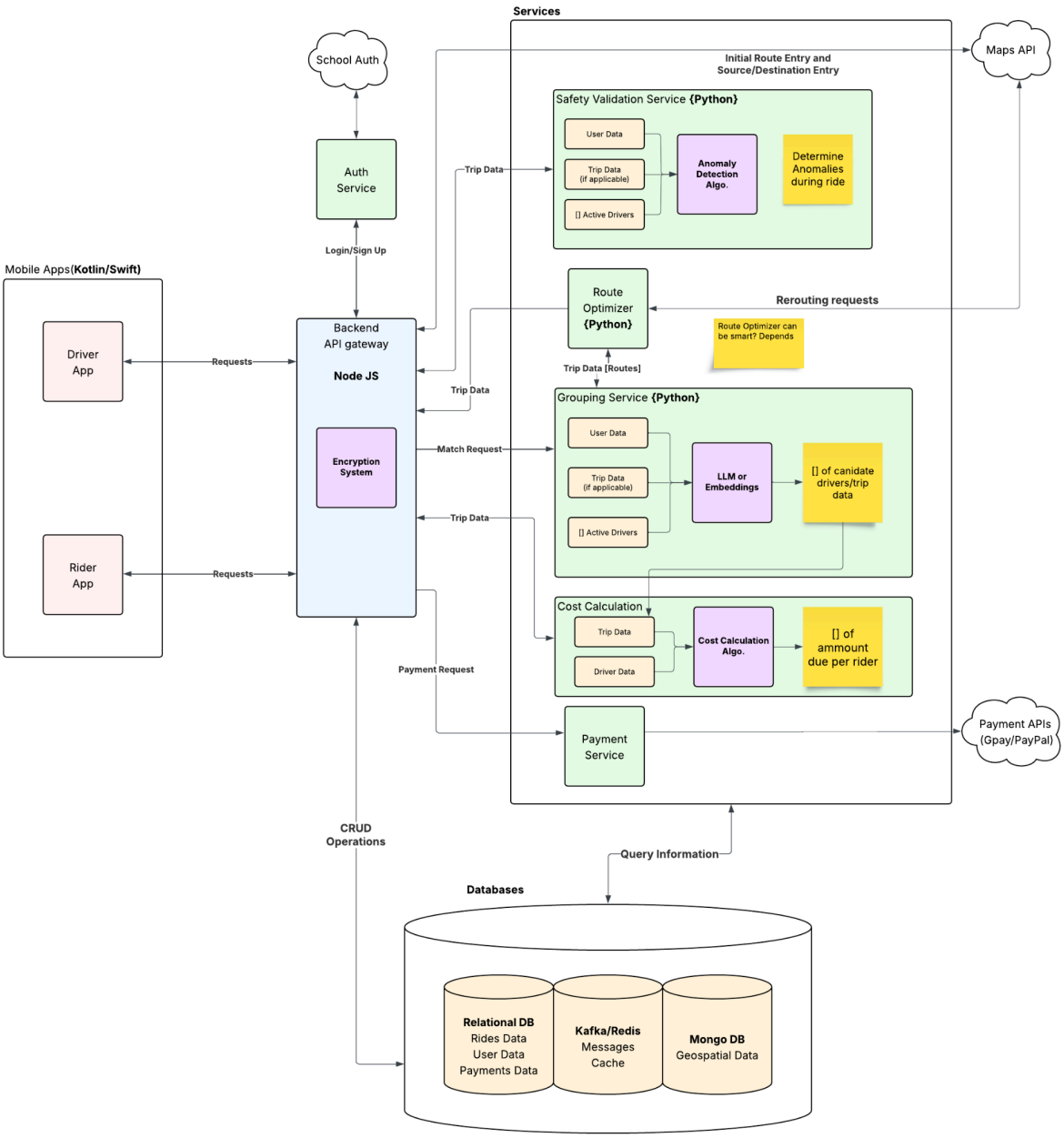


Figure 2. A System Architecture of our carpooling platform

Figure 2 shows a system architecture of the proposed system, called “LessGo”. We opted for a microservice architecture with each functional aspect of our project broken down into its own service. This allows us scalability and independence across features. Our services are described below.

- API Gateway:
    - Acts as a central entry point for both the rider and driver apps.
    - Directs incoming requests to the relevant microservice and manages database operations.
- Authentication Service:
    - Handles Logins, Sign Ups, and verification with the school authentication system.
    - Ensures only verified members (e.g., students) access the platform, boosting trust and campus security.
    - Manage user session security and access control.
- Grouping Service:
    - Takes in a match request and uses information from users and active drivers with their respective trip data to make an optimized list of candidates.
    - Uses user (ex., based on class schedule), driver, and trip data to optimize candidate lists for carpooling, aiming for best-fit matches.
- Safety Service:
    - Monitors the ride activity to check if they deviate too much from the optimal route.
    - Sends silent alerts for the riders to respond to.
    - Sends a report to their emergency contacts if the rider does not respond within a set time.
- Route Optimization Service:
    - Takes in trip data and produces optimal routes using an algorithm that can be smart (A* algorithm) or non-smart (Dijkstra).
    - Minimizes total detours and travel time for all users in the carpool, which is crucial on crowded campuses.
- Cost Calculation Service:
    - Takes in the Trip data and info about the driver to calculate the respective cost of each rider in the trip.
    - Addresses the need for transparency and equity in fare splitting, which is lacking in most large platforms.
- Payment Service
    - Integrates with payment APIs like PayPal, Stripe, etc.
    - Securely handles fund transfers between riders and drivers.

6. Project Design

Design Overview

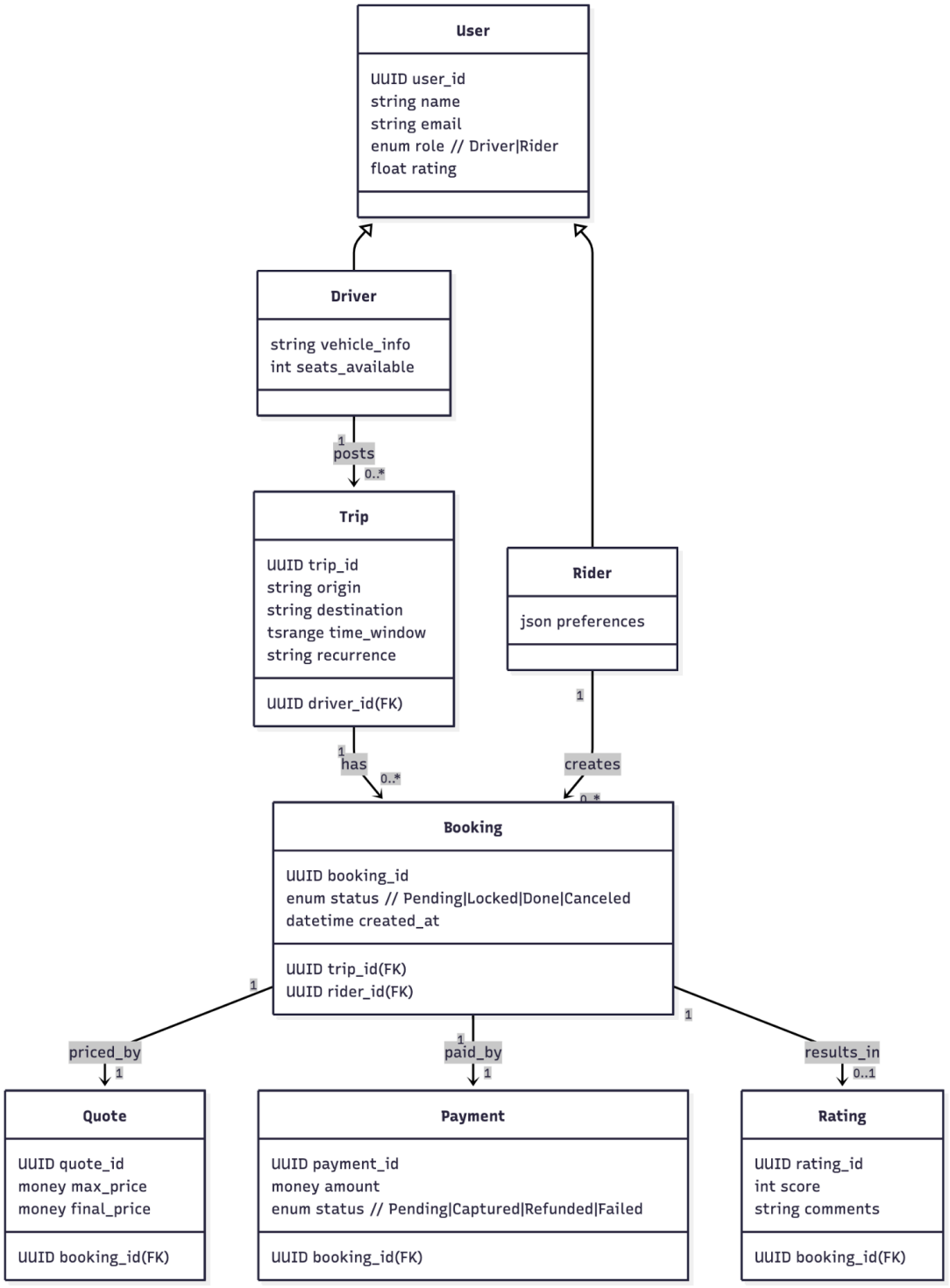
Our Smart Carpooling Matching Platform uses a client–server, layered design with two distinct clients (Rider App and Driver App) connecting through an API Gateway / BFF to a set of domain services (Matching, Pricing, Payments, Notifications). Route feasibility and stop ordering are delegated to a lightweight Optimizer Service (Python + OR-Tools), while the rest of the domain runs in Node.js/Express. Persistent data lives in PostgreSQL; Redis caches hot items such as open trips, seat counts, and quotes. External providers supply maps/ETAs (Google/Mapbox), payments (Stripe), and push/SMS.

- **Rider App (Swift/Kotlin)** lets students search by class time window and corridor, preview the route and ETA, and receive a max price quote that can only go down as seats fill.
- **Driver App (Swift/Kotlin)** lets students post recurring trips from their class schedules (e.g., "SF → SJSU, M/W/F 7:10 AM"), manage seats, and receive payouts.
- **Matching Service** filters candidates by time window and polyline overlap, then calls Optimizer for feasible stop ordering and detour metrics.
- **Pricing Service** implements the literature-backed Hu-style monotonic cost sharing with detour-aware allocation (cost-causer pays): driver minimum = miles × fair rate × (1+margin); riders pay base share + their own added miles/minutes + small delay surcharge they cause; quotes only reduce as seats fill.
- **Payment Service** holds/captures via Stripe; Notification Service pushes quote drops, lock events, and trip updates.
- **Data model** centers on Users, Trips (recurring), Bookings, Quotes, Payments, and Ratings.

This modular design keeps matching and pricing transparent and auditable, supports fast iteration on algorithms, and aligns with your separate Driver vs. Rider clients.

Design Artifacts

UML Class Diagram



Purpose: models entities and relationships for the domain layer.

Key Classes:

- User (abstract) → subclasses: Driver, Rider.
- Trip: recurring route posted by a driver.
- Booking: rider's reservation linked to a specific trip.
- Quote: stores both max and final prices for a booking.
- Payment: manages charge status for each booking.
- Rating: optional feedback entity tied to completed bookings.

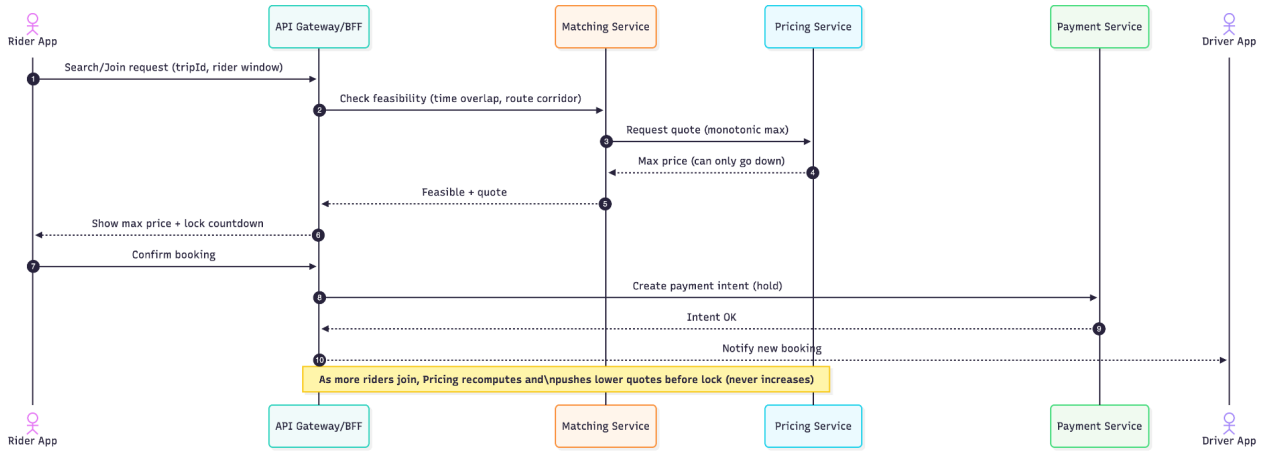
Relationships:

- Driver → posts → Trip (1..*)
- Trip → has → Booking (0..*)
- Rider → creates → Booking (0..*)
- Booking → has → Quote (1) and Payment (1)
- Booking → results_in → Rating (0..1)

Design Rationale:

- Keeps pricing, payment, and reputation logic modular.
- Ensures clear audit chain: Booking → Quote → Payment → Rating.

Sequence Diagram



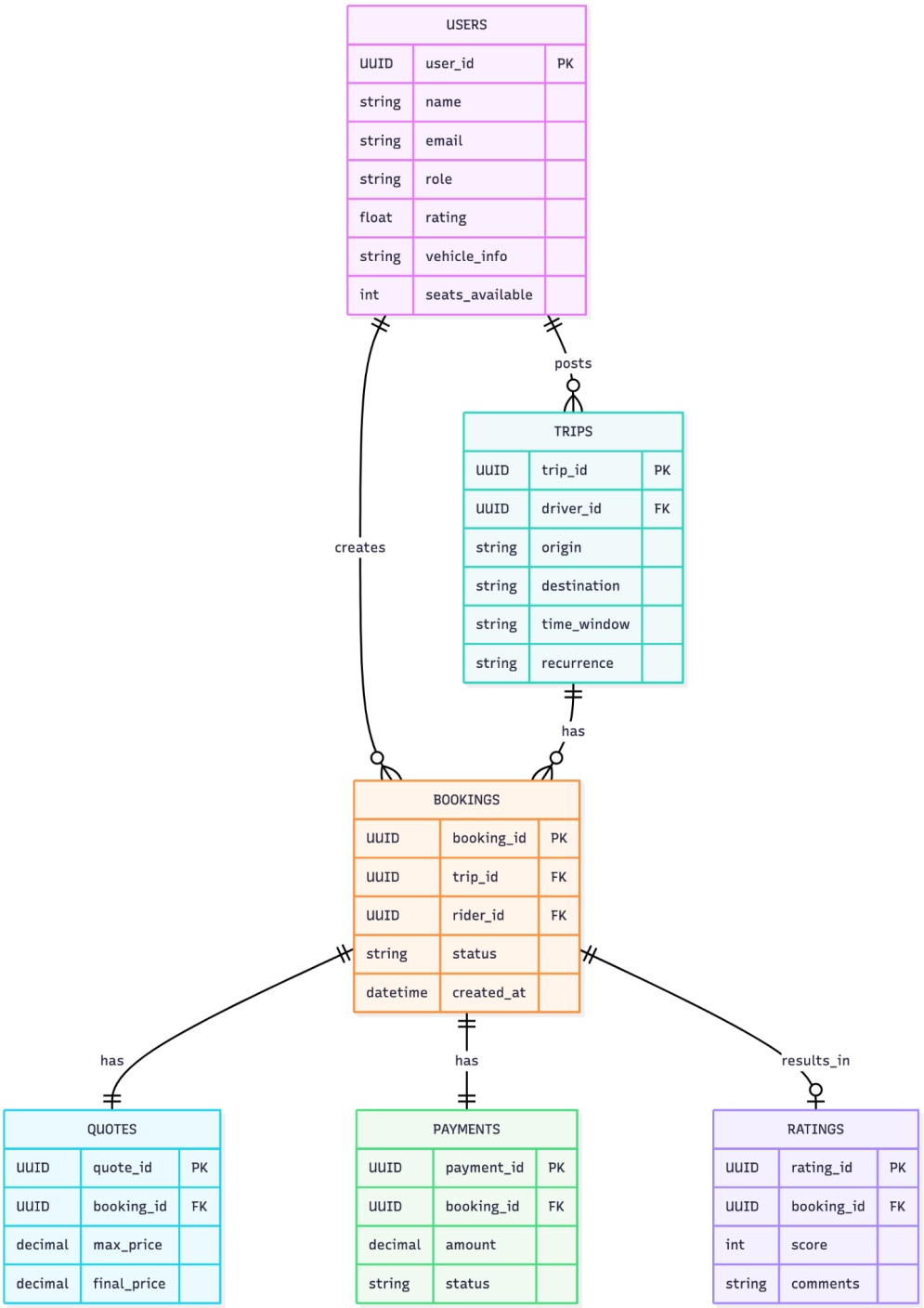
Scenario: Rider joins a posted trip.

Main Steps:

1. Rider App sends search/join request to API Gateway.
2. Grouping Service validates route/time overlap.
3. Grouping → Pricing Service requests monotonic quote.
4. Pricing returns the maximum fare (never increases).
5. Rider views the quote + countdown and confirms.
6. Payment Service creates a payment intent (funds held).
7. Driver App receives booking notification.
8. As new riders join, Pricing recomputes and sends lower quotes to all.

Outcome: predictable rider pricing, guaranteed driver minimum, transparent transaction trail.

Entity-Relationship (ER) Diagram



Purpose: defines logical schema for persistent data.

Core Tables:

- USERS(user_id PK, name, email, role, rating, vehicle_info, seats_available)
- TRIPS(trip_id PK, driver_id FK, origin, destination, time_window, recurrence)
- BOOKINGS(booking_id PK, trip_id FK, rider_id FK, status, created_at)
- QUOTES(quote_id PK, booking_id FK, max_price, final_price)
- PAYMENTS(payment_id PK, booking_id FK, amount, status)
- RATINGS(rating_id PK, booking_id FK, score, comments)

Relationships:

- One driver → many trips
- One trip → many bookings
- One booking → one quote, one payment, optional rating

Why relational:

- Guarantees referential integrity for pricing and payouts.
- Enables financial auditing and historical tracking.

Key Design Considerations

- Separate client apps → cleaner UX and role-specific logic.
- Monotonic pricing → transparent cost-sharing, rider trust.
- Detour-aware allocation → fairness: those adding distance pay more.
- Python optimizer → isolates heavy computation (detour routing).
- PostgreSQL + Redis → balance between ACID integrity and real-time caching.
- Service boundaries → modular growth (future RL-based seat optimization).

7. QA, Performance, Deployment Plan

Objective: Ensure the Smart Carpooling Matching Platform functions reliably across rider and driver apps, maintains cost transparency, and meets performance targets under real-world load.

Testing Methodology:

- **Unit Testing**
  - Performed on each service layer (Auth, Trip, Grouping, Pricing, Payment).
  - Tools: Jest (Node.js) for API routes; PyTest for the Optimizer (Python).
  - Goal: verify core logic such as quote calculation, detour-cost allocation, and seat-lock accuracy.

- **Integration Testing**
  - End-to-end trip flow: Driver posts trip → Rider joins → Quote generation → Payment intent → Lock.
  - Ensures all modules (PostgreSQL, Redis, Stripe, Maps API) interoperate correctly.

- **UI / UX Testing**
  - Performed using Cypress for both Rider and Driver web apps.
  - Focus: quote updates, live seat counts, and push notifications.
  - Acceptance criteria: quotes update within 3s of another rider joining; no stale UI state after payments.

- **Performance / Load Testing**
  - Tools: k6 and Postman Collections simulate concurrent bookings and quote recalculations.
  - Target Metrics:
    - Quote response time < 1.5 s (avg)
    - Group and pricing pipeline < 3 s end-to-end
    - 99.9 % uptime under normal usage (1000 users)

- **Regression & CI Tests**
  - Automated with GitHub Actions; runs on every commit.
  - Validates APIs, pricing accuracy, and route-group consistency.

- **Security & Data Validation**
  - JWT-based authentication verified against tampering.
  - SQL injection, rate-limit, and input-validation tests were performed on the API Gateway.
  - Stripe sandbox used for secure payment simulation.

Testing Outcomes (Planned / Current):

- Core services (Grouping, Pricing) achieve functional correctness within test environments.
- UI integration and push-notification reliability are at 95 % success in preliminary trials.
- Beta testing with simulated SJSU user data scheduled post-deployment.

Algorithm Verification and Performance Evaluation

1. **Grouping Algorithm – Validates time-window and route-overlap logic.**
    - Dataset: 500 simulated riders, 150 driver routes (SJSU corridor).
    - Success metric: ≥ 90 % feasible matches in < 2 s.

2. **Cost-Sharing Algorithm – Confirms quotes never increase.**
    - Test: sequential rider joins on the same trip.
    - Outcome: final fare decreases 8 – 15 % as seats fill; driver payout stable within ± 3 %.

3. **Optimizer (Python + OR-Tools) – Evaluates stop-order generation and detour minimization.**
    - Average computational time < 1 s for ≤ 5 stops; scales linearly up to 20 stops.
    - Improvement: 12 % shorter detour distance vs. naïve sequential order.

4. **Redis + PostgreSQL Hybrid Cache – Assessed for query latency.**
    - Quote lookups improved ~ 40 % over DB-only implementation.

Deployment Plan

Environment:

- **Cloud Platform:** AWS (ECS / Elastic Beanstalk)
- **Containerization:** Docker for Node.js services, Python Optimizer, and Redis.
- **Database:** AWS RDS (PostgreSQL) + Elastic Cache (Redis).
- **Static Assets:** S3 + CloudFront for React builds.
- **Monitoring:** CloudWatch / Datadog with alerting (quote latency > 2 s, CPU > 80 %).
- **CI/CD:** GitHub Actions pipeline triggers on merge to main; auto-builds containers, runs full test suite, deploys via Terraform.

Deployment Stages:

1. **Development** – Local Docker Compose (Node.js + Python + Postgres).
2. **Staging** – Hosted on AWS ECS; integrated with sandbox Stripe + Maps API.
3. **Production** – Full SSL deployment with custom SJSU domain and monitored endpoints.

4. **Beta Rollout** – Invite-only SJSU users (10 drivers / 50 riders).
5. **Full Launch** – Post-beta tuning, reliability certification, and scalability testing.

Rollback Strategy:

- Versioned container images enable one-click rollback via ECS.
- Database migrations run with Liquibase in reversible mode.
- Blue/green deployment keeps the prior environment live until the new one passes health checks.

8. Implementation Plan and Progress

Implementation Plan

1. Environment Setup
    - Team Members should have Node.js, Python, DBs, XCode, and any Cloud Service set up.
    - Orchestration Tools should also be set up for the microservices
2. Acquire Development Tools
    - Backend: Node.js, npm/Yarn, Express, Swift, Kotlin
    - Apps: Swift/Kotlin/React Native (depends on which app)
    - Database: PostgreSQL or MongoDB
    - Models: PyTorch/scikit-learn, Jupyter Notebook/Lab
3. Develop algorithms/models
    - For any ML models, find any public datasets and relevant models
        i. Locate, clean, and examine any public datasets.
        ii. Train & Test Model.
    - Theorize and refine cost calculation algorithms
    - Theorize Privacy and Safety Service
4. Sample Programs / Example Runs
    - Any new algorithms / AI Models need to be tested

    - Each service should be run/experimented on separately
    - API Gateway should be tested (Routes correctly to dummy service)
    - DBs should be tested with inserts and queries to validate schemas

5. Prototype Development
    - Auth service & DB Schema finalized
    - Gateway finalized
    - Models & Algorithms finalized
    - Microservices finalized
    - UI/Front-end finalized
    - Testing: Scaling and Load

Progress Tracking

In Progress

- 1) Environment Setup
- 2) Acquire Development Tools
- 3) Develop algorithms/models

Future Items

- 4) Sample Programs / Example Runs
- 5) Prototype Development

9. Project Schedule

Gantt Charts

Task Name	Subtask	Owner	Week 1 Aug 25	Week 2 Sep 01	Week 3 Sep 08	Week 4 Sep 15	Week 5 Sep 22	Week 6 Sep 29	Week 7 Oct 06	Week 8 Oct 13	Week 9 Oct 20	Week 10 Oct 27	Week 11 Nov 03	Week 12 Nov 10	Week 13 Nov 17	Week 14 Nov 24	Week 15 Dec 01
Research Survey	Peer review - cost calculation methods	Shyam Kannan															
Research Survey	Peer review - safety/privacy algorithms	Spencer Davis															
Research Survey	Peer review - mapping & routing algorithms	Sri Ram Mannam															
Research Survey	Peer review - mapping & routing algorithms	Johnny To															
Research Survey	Analysis of existing solutions (Uber/Lyft/others)	All															
Design Deep Dive	Notification Service Design Spec	Sri Ram Mannam															
Design Deep Dive	UI Components Planning	Shyam Kannan															
Design Deep Dive	Privacy & Security Architecture	Spencer Davis															
Design Deep Dive	Matching & Routing Algo DSR	Johnny To															
Requirements	Draft User Stories	Sri Ram Mannam															
Requirements	Stakeholder Interviews	Spencer Davis															
Design	UI Mockup Concepts	Sri Ram Mannam															
Design	System Architecture Sketch	Johnny To															
API Planning	Draft Endpoints	Shyam Kannan															
Data Model	Domain Entities Plan	Spencer Davis															
Tech Evaluation	Choose Backend Stack	Johnny To															
Prototype	Mockup Rider UI	Sri Ram Mannam															
Prototype	Mockup Driver UI	Shyam Kannan															
Prototype	Integration Demo Prep	Spencer Davis															
Review	Team Feedback Session	All															
Technical Exploration	Prototype Google OAuth API	Shyam Kannan															
Technical Exploration	Test Firebase Notifications	Sri Ram Mannam															
Tech Review	Investigate Python async/await	Johnny To															
Tech Review	Try React Native maps	Spencer Davis															
Synthesis	Team shares findings + quick demos	All															
Setup & Environments	Create GitHub repos & folder structure	Johnny To															
Setup & Environments	Configure CI/CD and linting	Shyam Kannan															
Basic App	Initial React Native App Skeleton	Sri Ram Mannam															
Basic App	Initial Python Backend Skeleton	Spencer Davis															
Basic App	Push a working Hello World for both apps	All															
Basic App	Document setup in README	Shyam Kannan															
Basic App	Environment validation & install scripts	Johnny To															
Cross-platform Investigation	Explore reusing business logic between Kotlin/Swift	Spencer Davis															
Cross-platform Investigation	Prototype code conversion from Kotlin to Swift	Sri Ram Mannam															
Cross-platform Investigation	Find common libraries usable in both apps	Shyam Kannan															

Figure 3. Gantt Chart for 1st semester's schedule

Task Name	Subtask	Owner	Week 1 Feb 02	Week 2 Feb 09	Week 3 Feb 16	Week 4 Feb 23	Week 5 Mar 02	Week 6 Mar 09	Week 7 Mar 16	Week 8 Mar 23	Week 9 Mar 30	Week 10 Apr 06	Week 11 Apr 13	Week 12 Apr 20	Week 13 Apr 27	Week 14 May 04	Week 15 May 11
Infrastructure	Repo/Infra Setup	Sri Ram															
Backend	DB Schema & API Gateway	Spencer															
Auth Service	Campus SSO Integration	Shyam															
Notifications	Push/SMS Service	Johnny															
Cost Calculation	Price Logic & API	Sri Ram															
Payments	Gateway Setup	Spencer															
Matching Service	ML-based Matching	Johnny															
Safety Service	Deviation & SOS	Shyam															
Route Optimization	A*/Dijkstra	Spencer															
Rider App	Registration/Login	Sri Ram															
Rider App	Ride Book/Negotiation	Shyam															
Rider App	Live Trip & Status	Johnny															
Rider App	Receipt & Rating	Spencer															
Driver App	Registration/Login	Sri Ram															
Driver App	Trip Setup & Details	Shyam															
Driver App	Pricing & Publish	Johnny															
Driver App	Match/Accept Riders	Spencer															
Driver App	Navigation & Earnings	Sri Ram															
Admin Portal	Login & Map	Shyam															
Admin Portal	User Management	Johnny															
Admin Portal	Dashboard & Insights	Spencer															
Integration	End-to-end Demo	Sri Ram															
Security Testing	Data Privacy	Shyam															
Stress Testing	Peak Load	Johnny															
Gamification	Leaderboards/Streaks	Spencer															

Figure 4. Gantt Chart for 2nd semester's schedule