Saathi - A Carpooling Solution

Final Report

Muneeb Shafique - ms06373
Fatima Alvi - fa06666
Syed Shayan Ahmed Qadri - sq06656
Sajeel Alam - sa06840

May 10, 2024

Supervisor: Dr. Qasim Pasta

Contents

1 Introduction

Saathi is a carpooling application designed to facilitate efficient and sustainable transportation amongst students and corporate employees. This eco-conscious app aims to reduce the environmental footprint of vehicles while providing a convenient mobility experience, specifically tailored to connect individuals from the same institution for shared commutes. This document outlines the comprehensive project plan, functional and non-functional requirements and system diagram for the development of this cutting-edge software solution.

1.1 Purpose

The primary purpose of Saathi is to:

- **Efficient Ride-Sharing**: Enable users to easily find and share rides with others who have similar commuting needs within their respective organizations or institutions.
- **Reduce Congestion**: Alleviate traffic congestion and the demand for parking spaces by encouraging ride-sharing.
- **Environmental Sustainability**: Promote a greener and more sustainable mode of transportation, reducing the carbon footprint associated with individual vehicle use.
- **Community Building**: Foster social connections among members of the organization or institution by encouraging collaborative transportation solutions.

1.2 Scope

The scope of Saathi encompasses the development of a user-friendly mobile application aimed at facilitating shared transportation for students and corporate employees. Key features include:

- **Authentication and Registration:** Develop secure user authentication and registration processes to safeguard user data and ensure a personalized experience.
- **Ride Scheduling:** Allow users to plan and book rides in advance, accommodating both regular commutes and one-time trips.
- **Communication Platform:** Enable riders and drivers to communicate seamlessly within the app for coordination and updates.
- **Route Matching:** implement a route matching algorithm that matches passengers to drivers that are travelling on the same route.
- **Scalability:** Design the application to handle potential growth by scaling infrastructure and resources as needed to accommodate a larger user base.

2 Project Plan

2.1 Project Objectives

- **Design and Development**
  - Create a user-friendly mobile application interface for two distinct user groups: drivers and passengers.

  - Implement a secure and efficient user authentication system.
  - Integrate a robust Maps and Navigation API for route planning and navigation.
  - Develop an intelligent matchmaking algorithm that pairs drivers with passengers based on routes, timing, and preferences.
  - Establish a notification system to alert users of ride status, updates, and promotions.

- **Testing And Quality Assurance**
  - Conduct testing within the development team to identify and fix technical issues.
  - Perform testing with a select user group from the university to gather feedback and ensure user satisfaction.

- **Deployment**
  - Launch the application for university-wide access with initial onboarding and informational sessions.
  - Monitor system performance and user adoption, adjusting technical support as needed.
  - Once testing is done with university-wide access we plan to move on to scaling our application to allow for more institutions to have their own instances of the app.

- **Scalability**
  - Ensure the system architecture is scalable and can handle increased user loads and expanded geographic service areas.
  - Adopt a modular architecture in the app development that allows for incremental improvements and easy updates to support new features and institutions without significant downtime or overhauls.

2.2 Timeline

The project tasks and timeline follows the structure below which is divided into two main categories being Kaavish I and Kaavish II.

- **Kaavish I**
  - September
    * Market research and user needs assessment.
    * Defining project scope and objectives.
    * Defining useful and necessary features.
    * Define use cases and user stories.
  - October
    * Designing a low fidelity design.
    * Defining the flow of the application and main users.
    * Developing a clickable prototype.
    * Review and iterate on the prototype based on user feedback.
  - November
    * Wireframing key screens and user flows.
    * Designing high-fidelity UI mockups.

    - Review and iterate on the High-fidelity mockups based on user feedback.
    - Planning the system architecture and selecting technology stack.
    - Setting up the development environment and beginning coding.
  - December
    - Development of core frontend features.
    - Frontend testing and preparation for integration with the backend.

- **Kaavish II**
  - January
    - Planning the backend architecture with scalability in focus.
    - Setting up databases, server, and implementing Continuous Integration and Continuous Deployment pipelines.
  - February
    - Development of the route matching algorithm.
    - Integration of the algorithm with the frontend.
  - March
    - Comprehensive testing of the entire application
    - Iterating based on testing feedback to refine the application.
    - Soft launch of the application to a limited university audience (Beta version).
    - Collection and analysis of user feedback.
    - Final adjustments and improvements based on feedback.
  - April
    - Official launch of the application within the university.
    - Planning and initial steps for scaling to other institutions.

2.3 Risks

Some risks that may arise while developing and deploying our carpooling solution are as follows:

- **Technical Challenges:** Unforeseen technical issues may arise during development, causing delays.
- **Data Security:** Data breaches or privacy concerns may arise.
- **User Adoption:** Low user adoption can impact the app's success.
- **Third-Party Dependency:** Reliance on external APIs or services may lead to service disruptions or changes that affect the app's functionality.
- **Scalability Challenges:** Inadequate server scalability may result in system downtime or performance issues as user numbers grow.
- **Bug and Issue Resolution:** Unexpected bugs and issues may arise post-launch, impacting user experience.
- **Competitive Pressure:** Competing apps may emerge with similar features, affecting user acquisition.
- **User Behavior and Feedback:** Users may not embrace carpooling or may provide negative feedback

2.4 Resources

- **Development: Frontend**
  - The mobile application's frontend will be developed using React Native, enabling cross-platform compatibility and a seamless user experience on both iOS and Android devices.
  - The web component will be crafted using React.js to ensure a dynamic and responsive user interface.
- **Development: Backend**
  - The backbone of our mobile and web platforms will be Node.js and Express, which will serve as our primary backend service along with MongoDB for the application's databases.
  - For deployment, we will look into cloud services to host the application and databases, with a scalable architecture to support growing user demand.
- **APIs**
  - Our routing and navigation features will be powered by APIs such as Google Maps or Open Street Map, offering reliable and accurate geographical data.
  - For communication, we'll integrate APIs that provide SMS and Email services, crucial for sending notifications and one-time passwords (OTPs) for authentication.
- **Collaboration Tools**
  - Version control will be managed through Git, with GitHub as our repository platform to streamline code sharing and review.
  - Project tracking and management will be conducted using Jira, enabling a transparent and efficient development process.
  - Our development pipeline will incorporate Docker for Continuous Integration/Continuous Deployment (CI/CD), ensuring that updates are smoothly rolled out.
  - Design and prototyping will be facilitated by Figma, which will be used to create both our wireframes and finalized design interfaces.

3 Literature Review

3.1 Recommendation System

Carpooling involves sharing a private vehicle with others who have similar destinations or trajectories [1]. In urban mobility, the role of a carpooling solution has become increasingly pivotal due to growing concerns over escalating transportation costs, traffic congestion, and environmental degradation due to single-occupancy vehicle commuting. The success of carpooling services heavily relies on the recommendation system, which is crucial in matching riders and drivers efficiently. In considering the primary attributes for generating user recommendations, we focus on two key aspects: similarity in trajectories and user profiles. For developing an effective driver-passenger matching recommendation system, assessing the extent of route similarity between a driver and a passenger and evaluating their compatibility based on preferences like gender, smoking or non-smoking preferences, desired travel times, etc., is essential.

There is extensive existing literature regarding recommendation systems for taxicab services [2][3][4]. Taxicab-based systems typically assume that the destination of passengers is unknown, thus focusing on predicting potential trajectories. This contrasts with carpooling scenarios where the destinations are predefined. Therefore, while taxi-based recommendation systems are geared towards trajectory prediction, our carpooling application, Saathi, focuses on matching based on known destinations and aligning user preferences. This distinction highlights the need for a specialized approach in carpooling recommendation systems.

[5] utilizes a recommendation framework capable of calculating an aggregate score by taking into account the average time delay, vehicle capacity, driving distance, and profit increment. However, it's important to note a limitation of this approach—it primarily searches for available vehicles within a specified radius around user requests rather than emphasizing the identification of similarities in trajectories. Similarly, other systems such as [6] within the literature make use of recommendation algorithms; however, they exist primarily to assess the similarity between driver-passenger routes without taking user profile similarity into account in their recommendation process. [7] introduces a vehicle sharing model with two matching layers, based on similar characteristics and personalised maximum waiting times. However this employs Support Vector Machines for predictions which are not ideal for recommendation systems due to sparse user-item data [8].

In our application, Saathi, we are implementing the system Micheal Oliveira da Cruz developed in his thesis "On the similarity of users in carpooling recommendation computational systems" along with the methodology presented in his paper, "Grouping Similar Trajectories for Carpooling Purposes." This system employs trajectory discretization, temporal filtering, and clustering based on the Optics algorithm. It also incorporates user profile data to enhance the method's robustness in identifying users with similar trajectories and profiles [9]. Our implementation is inspired from Michael Oliveira da Cruz's system to provide a comprehensive and data-driven solution for improving the carpooling experience.

Our project's primary objective is to create a scalable carpooling application. It is essential to efficiently handle and process large volumes of trajectory data to achieve scalability, especially when identifying trajectory similarity clusters. Trajectories are collections of 3-tuples comprising of longitude, latitude, and time data points. A single trajectory path can encompass many of these 3-tuples, making it a considerable volume of data to process, which can be computationally expensive. To address these challenges our methodology emphasizes key computational methods which are essential for our applications functionality. These papers use trajectory discretization, which simplifies this data by reducing redundancy. This simplification is done by creating points of interest along the trajectory, allowing us to decrease the amount of data to be processed while retaining the essential trajectory information. Temporal filtering further refines the data by ensuring that only trajectories with similar departure and destination times are processed [1]. Utilizing clustering algorithms also allows enhanced scalability to manage substantial amounts of data. Furthermore, these algorithms can effectively handle noisy data and missing values, allowing them to evolve over time [10].

Another reason to implement the recommendation system outlined by [1][9] was its close alignment with our requirements. We aim to develop a carpooling recommendation system that relies on user profiles, encompassing their preferences and ratings, and with trajectory data to facilitate driver-passenger matching. [9] identifies a gap in the field up to 2016, where the integration of user preferences and trajectory data for ride matching remained largely unexplored, and aims to address this gap with its proposed solution. This alignment further solidifies the paper's relevance to our application.

In the initial stages of our project, our methodology for path similarity incorporated the use of similarity algorithm and the OPTICS algorithm in alignment with the approach detailed in the relevant literature. These algorithms were selected for their potential to identify similarities in trajectories and facilitate efficient driver-passenger matching. However, upon further investigation, we encountered significant challenges with the practical application of these methodologies. The approach outlined required exhaustive comparison across all driver and passenger data points within the system, which proved to be inefficient. This inefficiency was particularly evident when trying to form proper clusters, a critical step for the effective operation of our recommendation system. This highlighted the need for a more robust solution.

In response to these challenges, our focus shifted towards exploring alternative technologies capable of creating geographical clusters that would enable a mapping of data points. Our search led us to the H3 library, utilized by Uber in their highly efficient ride-matching systems [11]. Uber's success in dynamically matching riders with passengers through geographical clustering offered a compelling precedent for adopting the H3 library in our system. Its ability to partition space into hexagons allows for the creation of geographical clusters that can significantly enhance the efficiency of our matching algorithm by limiting comparisons to data points within the same hexagons. By adopting this technology, we aim to overcome the limitations encountered within the initial methodologies, enabling us to process trajectory data more effectively.

3.2 System Architecture

In the dynamic landscape of modern software development, the architectural foundation of an application plays a pivotal role in determining its functionality, scalability, and long-term viability. This literature review explores and evaluates different architectural paradigms which we considered for our system, particularly focusing on Monolithic, Service Oriented Architecture (SOA) and Microservices Architecture, in the context of developing a carpooling application. Carpooling applications, which are inherently complex and demand high scalability, real-time processing, and robustness, require a carefully chosen architectural approach to meet these challenges effectively. This review seeks to dissect the nuances of these architectures, understanding their individual strengths, weaknesses, and applicability. In order to make a more informed decision relevant to our use case, we studied the architectural choices made by leading players in the carpooling and ride-sharing market, providing insights into industry practices and trends. Using our findings, the aim is to reach a justified architectural choice for our application which aligns with our technical requirements and business objectives.

3.2.1 Comparing Architecture Patterns

There are various software architecture patterns, each suitable for a particular use case however, as mentioned earlier, a few which stood out to us were the monolithic, service oriented and microservices architectures. Monolithic Architecture is essentially a traditional model of a software program, which is built as a unified unit that is self-contained and independent from other applications. It can be described as a singular, large computing network with one code base that couples all of the business concerns together [12]. On the other hand, a Service Oriented Architecture is a software architecture style that refers to an application composed of discrete and loosely coupled software agents which communicate with each other, where each agent or service performs a required function. Moreover, Microservices is a type of service-oriented software architecture that focuses on building a series of autonomous components or services that make up an app [13]. Each service can be developed, scaled and deployed independently and services communicate with each other using lightweight protocols. In order to select a suitable architecture we had to evaluate the pros and cons of each architecture against the requirements of our application.

The primary goal for our application, Saathi, is to achieve scalable growth. As we anticipate an increase in users and features, our system must adapt seamlessly. In monolithic architectures, scalability is inherently limited due to its unified structure, which complicates scaling of individual components or integrating new technologies [14]. Similarly, in service oriented architecture (SOA), scalability can be constrained when services heavily share resources and require extensive coordination [15]. In contrast, microservices architecture offers a more viable solution for scalability. Here, each service operates independently, allowing for horizontal scaling by adding more service instances. This architecture also facilitates the smooth integration of new services and technologies, making it well-suited for our evolving application needs.

In traditional monolithic architectures, a single central database is used for all application components. This approach, while simpler, may limit flexibility, especially when dealing with diverse data types. In contrast, service oriented architecture (SOA) and microservices architectures advocate for the use of individual databases for each service [16]. This design is particularly beneficial for our application, which handles varied data types like user data, geospatial information, and real-time data. By employing separate databases, we can optimize the management of each data type. This approach enhances our application's performance, ensures better scalability, and maintains data integrity more effectively, catering to the specific needs of each data category.

Our application will encompass various components such as real-time in-app chat, maps and navigation, and push notifications. In a monolithic architecture, modifying any of these features could pose significant challenges due to its inherent limited flexibility. Any change necessitates redeploying the entire application, which can be cumbersome and risk impacting other functionalities [17]. Conversely, service oriented architecture (SOA) and microservices offer greater agility. These architectures allow individual components or services to be updated independently, significantly reducing the complexity and risk associated with modifications. This flexibility is crucial for continuously evolving and improving specific features of our application without disruptions.

In terms of reliability, the monolithic architecture, with its single-container structure, presents a significant risk due to the single point of failure. Any issue in one part of the application can potentially bring down the entire system. Similarly, in service oriented architecture (SOA), which relies on an Enterprise Service Bus (ESB) for communication between services, there's also a centralized point of failure [18]. On the other hand, microservices architecture offers enhanced reliability. Since services in microservices are independent, a failure in one service does not necessarily lead to a system-wide failure, making it a more robust choice for our application.

Monolithic architectures can sometimes offer better performance compared to microservices due to their inherent structure. The close proximity of software components in monolithic systems, sharing the same codebase and memory, allows for faster communication between these components [13]. This can lead to more efficient processing of tasks. In contrast, microservices architectures, which involve numerous API calls across multiple, separate services, might initially face performance challenges due to network latency and the overhead of these calls. However, implementing load balancers can significantly mitigate these issues by efficiently managing the traffic and distribution of requests, thus enhancing the overall performance of a microservices-based system.

Table 1 illustrates this comparison between the three architectures and aims to provide insights into the strengths and weaknesses of each architecture, aiding in the selection of the most suitable approach.

Monolithic	SOA	Microservices
Difficult to scale	Limited Scalability	Easy to Scale
Single Database	Individual Databases	Individual Databases
Limited Flexibility	Flexible	Flexible
Single point of failure	Single point of failure (ESB)	Reliable
High Performance	Performance Challenges	Performance Challenges

Table 1: Comparison of System Architectures

Microservices architecture emerges as the optimal choice for our project due to its scalability and flexibility, aligning with our priorities. By decomposing the application into smaller, independently deployable services, we can efficiently scale individual components based on demand, ensuring optimal resource allocation. Additionally, the ability to have multiple databases, each tailored to specific data types and storage requirements, enables greater flexibility and adaptability. This approach allows us to address the diverse data needs of our system, ranging from user data to geo-spatial information and recommendation system data, without introducing dependencies or constraints across the application.

3.2.2 Architectures of Exiting Solutions

In order to make a more informed decision relevant to our use case, it was essential to study the architecture designs of existing ride-sharing/carpooling solutions. Uber being one of the most prominent ride-sharing application that we know of with 130 million monlthy users and operations in 72 countries started off with a monolithic architecture in 2009 but in order to improve its system's speed, performance and enable scalibility, they shifted to a microservices architecture in 2014. In fact, they went one step further and adopted a new architecture called Domain-Oriented System Architecture (DOMA) where related microservices are grouped together into collections called domains and these domains are classified into layers which determine the dependencies of the microservices in that domain which is essentially still a microservices based approach [19]. On the other hand, another ride-sharing application Careem with 48 million registered users, also started off with a monolithic database however with an increase in data as the number of users they faced issues such as down-times and thus decided to migrate to a microservices architecture leveraging AWS technologies [20]. Today Careem has a cloud-based microservices infrastructure which is supported by AWS and Google Cloud offering them a range of services such as compute, storage, security and networking capabilities [21]. Similarly, Lyft which is a ride-sharing service in USA and Canada also migrated from the traditional monolithic architecture to microservices in 2018 enabling them to roll out features faster since they could deploy services independently, and also use different programming languages according to their requirements [22]. Moreover, Bla Bla Car which is a French based application with 90 million users and unlike previously mentioned applications, solely offers carpooling services also went from a PHP monolithic architecture to a microservices-oriented architecture using Google Cloud in 2018 [23]. This migration allowed them to focus more on new features instead of managing their infrastructure, analyse data to recommend better rides and making their application more reliable thus offering a better carpooling experience for their users.

After careful consideration of the various pros and cons of different software architectures and reviewing the implementations in existing solutions, we have decided to opt for a microservices architecture for our application. This choice is driven by the scalability, flexibility, and reliability that microservices offer. We are aware that this approach introduces a certain level of complexity; however, we plan to leverage services offered by cloud infrastructures to manage this complexity effectively. Our team is prepared to learn and adapt as we progress, ensuring that our application remains scalable and robust in the face of evolving demands and growth.

4 SRS

4.1 Functional Requirements:

This section outlines the functional requirements of the ride-sharing application, detailing the capabilities provided to passengers, drivers, and general users.

4.1.1 Passenger Use Cases

- **Scheduling Rides:** Passengers can schedule rides by creating ride requests. These ride requests are visible to drivers and allow them to select passengers based on their preference. Ride requests can be placed in advance and can be scheduled as recurring or non-recurring. Passengers may also set preferences such as gender, smoking/non-smoking, etc.
- **Managing Scheduled Rides:** The passenger can view all the rides they have created. For each ride, they can view which drivers have accepted their ride and then choose to accept or reject them. Passengers have the option to cancel or update a ride before it begins. Cancellation of scheduled rides could result in reduced ratings.
- **Booking Rides:** Passengers can search and book rides by viewing available options after specifying commute details. To provide better ride recommendations, passengers can set preferences such as gender, smoking/non-smoking, etc. Passengers can send requests to join ride offers made by drivers based on cost, convenience or preferences.
- **Ride Information and Coordination:** Passengers may access and coordinate ride details, including viewing driver profiles and vehicle specs, tracking driver location, receiving updates and estimated arrival time. They can also communicate with drivers and fellow passengers.
- **Ride Notifications:** Passengers should receive notifications and reminders for upcoming rides. The app should provide timely notifications containing ride details such as pickup time. Additionally, the app should notify the passenger promptly in case of ride cancellations.
- **Feedback, Ratings, and Reporting:** Passengers can contribute feedback by rating drivers, providing detailed reviews, and reporting any issues or concerns during or after their rides.
- **Experience and Impact Tracking:** Saathi enables passengers to track their carpooling impact, sharing their experiences and assessing the environmental and cost-saving benefits.
- **Navigation and Commuting Enhancements:** Passengers can utilize tools like lost and found systems and FAQs for an improved commuting experience.

4.1.2 Driver Use Cases

- **Creating Ride Offer:** Drivers can offer rides by setting up ride requests. These ride requests allow the driver to decide on fares, accept and reject passenger responses, and schedule recurring and non-recurring rides efficiently. These ride offers require a driver to provide route details, seat availability and timings. Drivers may also set preferences such as gender, smoking/non-smoking, etc.
- **Managing Ride Offers:** The driver can view all the rides they have created. For each ride, they can view which passengers have accepted their ride and then choose to accept or reject them. The driver can also edit or delete their rides. The driver has the option to cancel a ride before it begins. However, this action may

result in a loss of their rating, as it may inconvenience passengers who have accepted the ride. The driver should use this option sparingly and consider passengers' needs.

- **Searching for Passenger Requests:** Drivers may search through existing requests made by passengers and select based on cost, timings and passenger route. If they find a suitable match, they can send them a request.
- **Driver and Ride Information:** Drivers can share their live location with confirmed passengers to coordinate with the passengers effectively.
- **Feedback and End Ride:** Drivers participate in the feedback loop, rating passengers and providing feedback, while also being able to mark no-shows. After dropping off all passengers, the ride end page is available for an hour unless the driver chooses to end the ride manually.
- **Sustainability and Engagement Tracking:** Drivers utilize tools to track their carpooling impact and engagement, encouraging sustainable driving practices.
- **Ride Information and Coordination:** Drivers can view passenger profiles, ratings and pickup/drop-off locations. They can also communicate with all confirmed passengers.
- **Driver Ride Management Notifications:** Drivers should receive timely notifications and reminders about upcoming rides, passenger cancellations, and any ride changes. These notifications enable the driver to effectively plan their commute, prepare for pickups, and stay informed about any ride alterations.
- **Support and Emergencies:** In case of emergencies or inquiries, drivers can access immediate support from the customer service system.

4.1.3 General User Use Cases

- **Account Management:** Users can handle account-related tasks including signing up, logging in and out, and updating profile information for continued access.
- **Navigation and Traffic Updates:** Users receive and provide real-time traffic information, helping to improve the ride planning process.
- **Ride History Review:** Users can review their past rides, keeping records and using the details for future ride planning.

4.1.4 Use Case Diagram

The use case diagram provides a visual representation of the functional interactions between the users (Passengers and Drivers) and Saathi. The diagram below outlines the key features available to users and shows how the system facilitates ride scheduling and management.

Figure 1: Use Case Diagram

4.2 Non functional Requirements

4.2.1 Performance Requirements:

- Manually time how long the app takes to load pages and respond to user inputs.
- The app should load any page within 2 seconds under normal conditions.
- Conduct informal load testing by having a group of users access the app simultaneously and monitor how well it responds.

4.2.2 Scalability Measurement:

- The system should support at least 100 concurrent users (within university) without performance degradation.
- The system should be designed to scale horizontally to handle an increase in load by adding more servers.
- The database should handle an increase in transactions and users, with the ability to scale to 1 million users.
- Use cloud services that provide auto-scaling capabilities based on system load.

4.2.3 Security Measurement:

- Perform basic security checks, such as ensuring passwords are hashed and personal data is encrypted.
- User Consent: Implement mechanisms to obtain explicit user consent before collecting, using, or sharing their data, especially for location tracking.
- Collect only the data that is absolutely necessary for the functioning of the app. Avoid collecting extraneous information that could compromise user privacy.
- While using third-party services or APIs for location tracking such as google maps, ensure they also comply with privacy standards and do not compromise user data.

4.2.4 Reliability Requirements:

- Track any crashes or bugs experienced during user testing sessions.
- Manually test the app's failover mechanisms by simulating failures (like shutting down a server).

4.2.5 Usability Measurement:

- Conduct user experience (UX) testing sessions (atleast 20) the student body and faculty/admin at Habib to gather direct feedback on the app's interface and workflow.
- Observe users interacting with the app and note any areas where they struggle or get confused. The app should be intuitive and require minimal training for new users. Achieve at least an 80% satisfaction rate on user experience surveys.
- More specifically, we will be measuring
  - Time to Complete Tasks: Measure the time it takes for new users to complete core tasks(ride booking, ride requesting etc) without assistance. Shorter times generally indicate more intuitive design.
  - Error Rate: Track the number of errors or missteps users make when interacting with the app for the first time. Fewer errors can imply a more intuitive interface.

- Success Rate: Determine the percentage of users who can successfully complete a task on their first attempt. A higher success rate can be indicative of an intuitive design.
- User Onboarding Experience: Evaluate the effectiveness of any onboarding process or tutorials. Users should be able to understand the app's main functions quickly.
- Qualitative Feedback: Collect qualitative feedback through interviews or open-ended survey questions to understand users' thoughts on the intuitiveness of the app.
- Clickstream Analysis: Analyze the paths users take within the app. Intuitive apps tend to have more direct and less convoluted user journeys.

4.2.6 Compliance Measurement:

- Review the app against any university's privacy and security requirements.
- Ensure the app meets any guidelines provided by the professors.
- Regularly review system processes and documentation to ensure they align with relevant standards and regulations.

4.2.7 Data Integrity Measurement:

- Implement input validation to prevent invalid data entry. Manually check that the data entered into the system is being saved and retrieved correctly.
- Use database constraints and transactions to maintain data consistency.

4.2.8 Backup and Recovery Measurement:

- Automatically back up data every 24 hours.
- Test the restoration process by using the backup data to rebuild the system in a test environment.

4.3 System Diagram

The system diagram presented here outlines the general structure of our application. At the heart of the platform are ten key elements that define its functionality: Authentication, User Interface, Passenger Interface, Driver Interface, Recommendation System, Maps and Navigation API, Rating Management, Ride Management, User Profile, and Live Updates Management. Each element plays a vital role in delivering a comprehensive service—from securing user data and personalizing experiences to ensuring smooth ride booking and up-to-date travel information. The diagram will guide you through the intricate relationships and processes that our application will harness to deliver a user-friendly and reliable carpooling service.

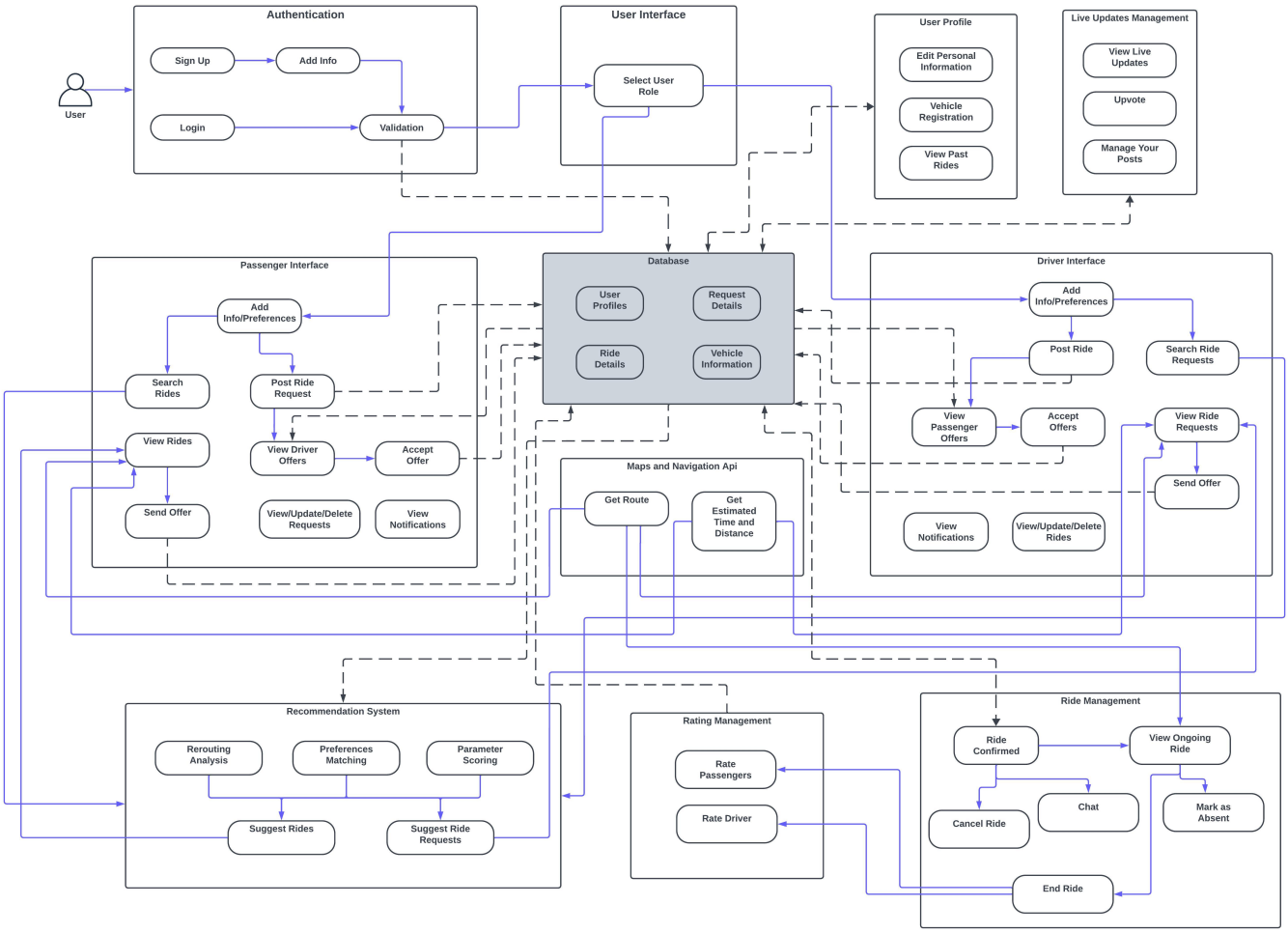

Figure 2: System Diagram

4.3.1 Description of Elements

In order to understand the intricacies of the diagram, it is essential to know what each components does and what purpose does it serve.

1. **Authentication**: The Authentication module is the entry point of our carpooling application, where users begin their journey. It is designed to provide a secure and straightforward mechanism for users to either sign up or log in. New users will register by providing their personal details, which the system will validate to maintain data integrity and security. Existing users will log in, and their credentials will be validated to ensure that unauthorized access is prevented.

2. **User Interface**: The User Interface acts as the central hub of interaction between the application and the user. After authentication, users select their role, either as a passenger or a driver—which tailors the subsequent options and functionalities available to them.

3. **Passenger Interface**: Through the Passenger Interface, individuals looking for a ride can interact with the application in various ways. They can add personal preferences, search for available rides and send offers, or post their own ride requests. When offers from drivers corresponding to the posted requests come in,

passengers have the option to review and accept them. This interface also allows passengers to manage their requests thus contributing to a hassle-free travel planning process.

4. **Driver Interface**: Parallel to the Passenger Interface is the Driver Interface, designed specifically for users who offer rides. Drivers add their information and preferences, search for ride requests from passengers and then choose to send offers to any suitable requests. They can also send post rides of their own and manage the responses they receive. This interface ensures drivers have control and visibility over their offered rides, facilitating an efficient connection with potential passengers.

5. **Recommendation System**: The Recommendation System employs advanced algorithms to enhance user experience by suggesting rides to passengers and ride requests to drivers based on their routes, preferences, and other parameters such as user ratings. This proactive feature aims to streamline the process of finding suitable matches for rides, thus optimizing the overall efficiency of the carpooling experience.

6. **Maps and Navigation API**: This integral component leverages geographical data to provide route information and estimated travel times. It's an indispensable tool for drivers to plan their routes and for passengers to know how long their trips will take. By integrating real-time navigation services, the application ensures that users are well-informed about their journey details and can make the best travel decisions.

7. **Ride Management**: At the heart of the operational aspect of the platform is the Ride Management module. Once a ride is confirmed, it facilitates communication between the passenger and driver via an in-built chat function. It also allows users to track the ride, cancel if necessary, and ensure that all parties are kept up-to-date with the status of the ride. This module is crucial for managing the dynamic aspects of each journey, from beginning to end.

8. **Rating Management**: Post-ride, the Rating Management feature allows both passengers and drivers to provide feedback on their experience. This not only fosters a community of trust and accountability but also assists in maintaining a high standard of service within the platform. The cumulative ratings serve as a reference for future users to make informed decisions when choosing rides or accepting ride requests.

9. **User Profile**: The User Profile section is a personalized space for users to manage their information, view their ride history, and keep track of their vehicle details if they are drivers.

10. **Live Updates Management**: This component ensures that users receive real-time traffic updates posted by other users, which can significantly impact ride planning. By providing up-to-date information, the application helps users avoid potential delays due to traffic, making for a smoother and more reliable carpooling experience.

A central database is implied in the system, which stores user profiles, ride details, vehicle information, and request details. The database interacts with almost every component, providing a persistent storage mechanism for the application's data.

4.3.2 Relationships and Data Flow

- **Authentication ↔ User Interface**: Users access the system through the User Interface, with the Authentication component managing access.
- **User Interface ↔ Passenger/Driver Interface**: Depending on the role selected, the User Interface directs the user to the respective interface.
- **Passenger/Driver Interface ↔ Recommendation System**: User actions from these interfaces feed into the Recommendation System to facilitate better matching of rides and requests.

- **Passenger/Driver Interface ↔ Maps and Navigation API**: Both interfaces use the Maps and Navigation API for displaying routes and estimating travel details.
- **Passenger/Driver Interface ↔ Ride Management**: The Passenger and Driver Interfaces use Ride Management to confirm, track, and manage their current ongoing ride.
- **Ride Management ↔ Rating Management**: After the ride has ended, Rating Management takes over to facilitate the feedback process.
- **Database ↔ All Elements**: The database acts as the central repository for data, interacting with every component to store and provide information as needed.

5 SDS

5.1 Data Model

This section provides the data model of our solution and how we plan to structure our data. We can see relationships between the tables along with the primary and foreign keys. We have created the data model using DB designer. The data model for Saathi is given below in figure 3.

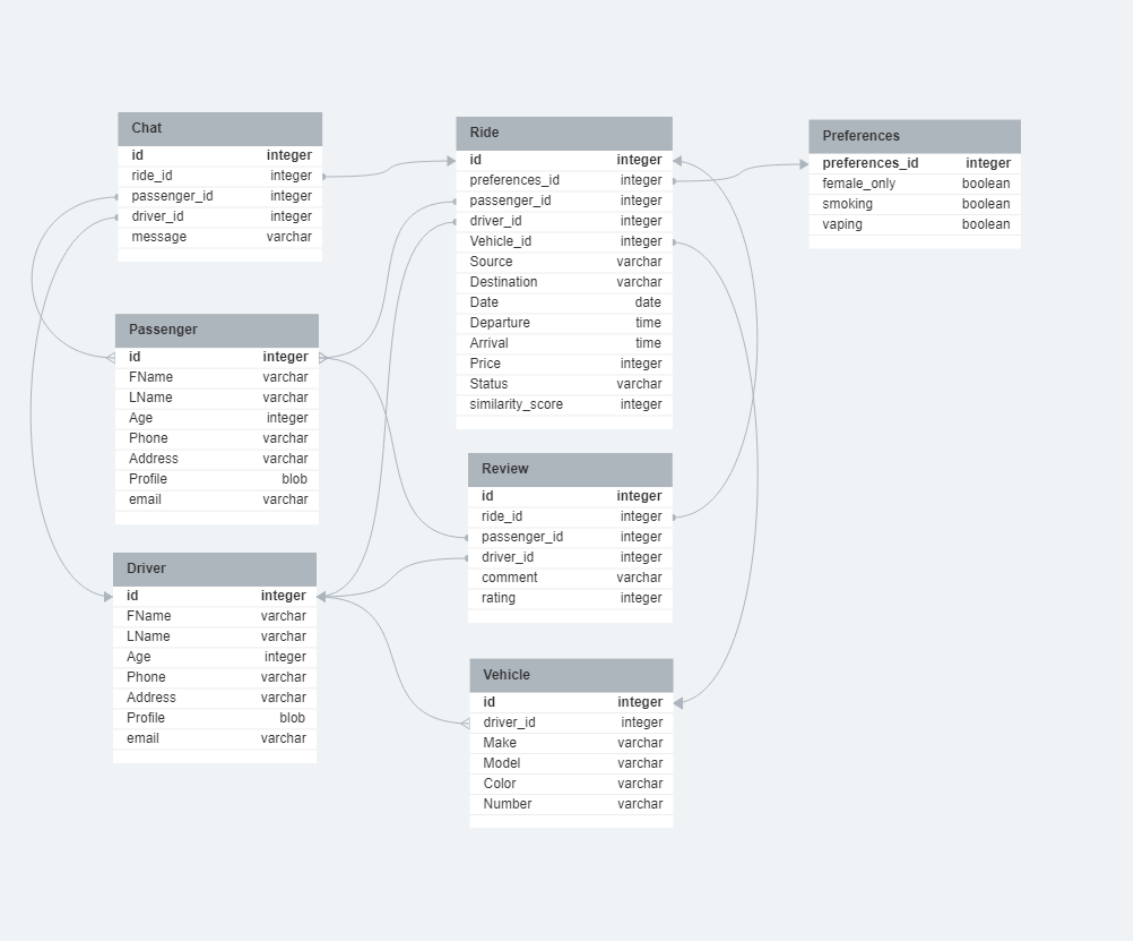

Figure 3: Data Model

- **Passenger/Driver:** The passenger and driver tables store important personal information about the individual.

- **Ride:** The Ride table stores data about a given ride, each ride is categorised by its own ride id as the primary key. Some more attributes of the Ride table include passenger and driver information along with source and destination points. It also contains a status attribute to decide if the ride is still in progress or not.
- **Chat:** Te Chat table stores information about a chat that is associated with a ride. The relation is one to one as only one chat is generated per ride that passengers will join as they join the ride.
- **Review:** Here reviews for each ride will be stored along with passenger comments and ratings provided by the passengers.
- **Vehicle:** The Vehicle table stores information about vehicles that are being registered by drivers. The table has a one to many relation as many cars can be registered by the same driver at a time. This contains important information about the cars to make sure passengers know what car to expect before they join a ride.
- **Preferences:** This table stores preferences for each ride through a preference id that is a foreign key in the ride table to set the preferences for each ride according to the ride details.

5.2 System Architecture

In this section, we present the microservices architecture that lays the foundation of our application. The architecture is designed to be scalable, reliable, and flexible, ensuring that the application can handle the dynamic demands of real-time carpooling services. It is composed of loosely coupled, independently deployable microservices, each responsible for executing a specific business capability.

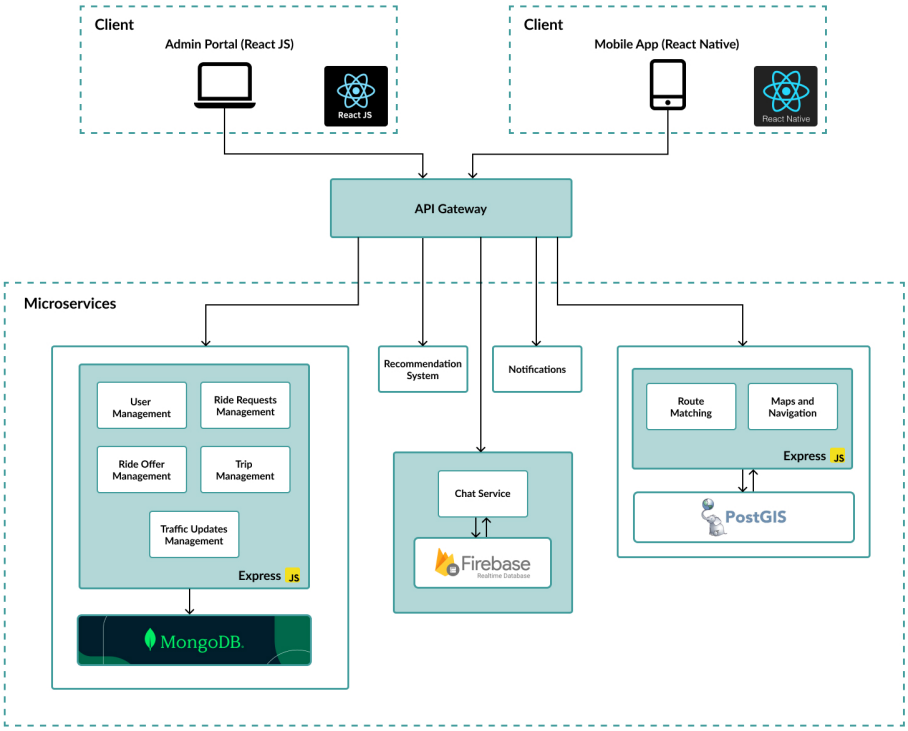

Figure 4: Architecture Diagram

The diagram visualizes the high-level structure of our application, illustrating the inter-service communication and data flow between the client applications, microservices, and the underlying databases. Our architecture is centered around an API Gateway, which acts as a single entry point for our React Native mobile application, effectively routing requests to the appropriate microservices.

5.2.1 Mobile Application - Frontend Technology (React Native)

The client-side of our application is developed using React Native, a popular cross-platform framework that enables us to build a native mobile application for both iOS and Android from a single codebase. Its modular and intuitive interface allows for building apps with a more agile and flexible approach. React Native also has a vast library of pre-built components available, which can accelerate development.

5.2.2 Admin Portal - Frontend Technology (React JS)

The admin portal is developed using ReactJS which was chosen for its component-based architecture. This modularity will allow various elements like charts, tables, forms, and navigation menus to be managed efficiently. Additionally, React's virtual DOM ensures efficient rendering of large datasets, by updating only the parts of the DOM that have changed thus enabling us to provide administrators with real-time insights and analytics. With ReactJS, we can build a responsive and scalable admin portal that efficiently caters to the data generated by our mobile application.

5.2.3 API Gateway

At the forefront of our backend architecture is the API Gateway, which serves as the orchestrator for all incoming API requests. It is the pivotal component that routes requests to the corresponding microservices, ensuring a seamless distribution of tasks and simplifying the client-side communication.

5.2.4 Communication Protocol: RESTful APIs

Inter-service communication within our architecture is predominantly handled via RESTful APIs, renowned for their simplicity and statelessness. These APIs are designed to be lightweight, enabling smooth and efficient communication between the various microservices that constitute the logic of our application.

5.2.5 Backend Framework: Express.js

Most of the microservices are built using Express.js, a lightweight and flexible Node.js web application framework. Express.js provides a robust set of features for web and mobile applications and is instrumental in the quick and easy setup of middleware necessary to respond to HTTP requests.

5.2.6 Databaes

Our microservices architecture is supported by three distinct databases, each chosen for their specific strengths and roles within the application.

- **MongoDB**: MongoDB is selected for storing majority of the data of our application as it offers scalable data storage through its sharding capabilities, which will be instrumental as our user base grows. This means that as the demand increases, we can maintain performance by distributing our data across multiple servers (horizontal scaling). The schema-less nature of MongoDB provides the flexibility needed to adapt our data model swiftly and efficiently as we iterate on our application, allowing us to incorporate new features without costly downtime or complex migrations. Its seamless integration with cloud platforms is another critical factor,

simplifying deployment and scaling in a cloud environment. This integration is not only about maintaining performance but also about leveraging the managed services that cloud providers offer, which can reduce the overhead of database management. Lastly, MongoDB's capacity for fast writes is essential for the real-time aspect of our application, ensuring that user interactions, such as ride bookings and updates, are processed quickly and reliably. This will contribute to a robust, responsive user experience, which is vital for the success of our application.

- **PostGIS**: PostGIS has been chosen as the geospatial database for our application due to its specialized capabilities in managing and querying geospatial data. As a dedicated geospatial database, PostGIS offers a comprehensive range of spatial functions that are essential for performing complex queries required by a location-based service like ours. These functions enable sophisticated spatial operations such as distance calculations, area measurements, and intersection checks, which are integral to route matching, optimizing routes, and providing real-time location updates. Moreover, PostGIS supports a variety of geometric and geographic data types, allowing us to accurately represent and manipulate a wide range of spatial data. This is particularly beneficial for storing detailed location data, which can include anything from simple point coordinates to complex polygons delineating service areas. The ability to handle such diverse data types with precision ensures that our application can provide accurate, context-rich information to our users, enhancing their experience and the overall functionality of our system.

- **Firebase Realtime Database**: Firebase Realtime Database is an ideal choice for the in-app chat feature in our application, primarily for its real-time synchronization, ensuring immediate and consistent message delivery across devices. Its offline capabilities are crucial for maintaining chat functionality even in fluctuating network conditions, guaranteeing message synchronization when connectivity resumes. Additionally, the serverless architecture of Firebase simplifies backend management, allowing us to focus on enhancing the chat interface without the complexities of server infrastructure. This choice streamlines development, ensures reliability, and optimizes operational efficiency, making Firebase perfectly suited for our application's dynamic chat requirements.

6 Design/Methodology

6.1 Mobile Application

6.1.1 Prototyping + Testing/Feedback

The work on the application began with the prototyping stage. During this phase, the primary goal was to ensure that all functional requirements are met and that the user experience flows seamlessly throughout the application. In our case, we initiated the prototyping stage by creating black and white screens using Figma. The decision to start with black and white screens was deliberate. By focusing solely on the functional aspects without the distraction of aesthetics, we could concentrate on the core functionalities of the application. This approach allowed us to prioritize usability and navigation, ensuring that users could accomplish their tasks efficiently without unnecessary friction.

However, before diving into the prototyping stage, we made sure we had a clear understanding of the functional requirements of the application. This involved numerous brainstorming sessions, user interviews (both student & faculty), and analysis of competitor applications such as blablacar and carvan to identify key features. After the key features had been decided, we leveraged figma to create black and white wireframes that mapped out the entire user journey within the application. Each screen represented a specific function or interaction, allowing us

to visualize the flow of the application. During the prototyping stage, we adhered to an iterative design process to ensure that our carpooling application met the needs and expectations of its users. This iterative approach involved continuous refinement and improvement of our wireframes based on feedback from users(students & faculty). We conducted regular user testing sessions where we invited potential users, primarily students from our institution, to interact with the prototype. To simulate real-world usage scenarios, we ran the prototype on mobile devices and asked participants to use the app as they would in a typical carpooling scenario. Observing how users responded to each screen and interaction was instrumental in identifying usability issues and areas for improvement. Whenever participants encountered confusion or difficulty navigating the app, we made detailed notes to address these issues during subsequent iterations. Similarly, any missing functionalities or suggestions for enhancements were meticulously recorded for further consideration.

To ensure a diverse sample size and comprehensive feedback, we intentionally recruited participants from various academic majors and years of study. This approach allowed us to gather insights from a broad spectrum of users, ensuring that our carpooling application was usable and accessible to all members of our institution's community.By involving users early in the design process and incorporating their feedback into our iterative design approach, we were able to refine our prototypes iteratively, saving time and resources in the long run. This user-centric approach not only helped us address usability issues but also ensured that our carpooling application aligned closely with the needs and preferences of its intended users. To mitigate the risk of overlooking critical functionalities or encountering usability issues later in development thorough documentation of functional requirements was done.

Also, during this process, we also identified additional features that could enhance the user experience. However, we made strategic decisions to prioritize certain functionalities over others based on their importance to the core functionality of the app. For example, during the prototyping stage, we decided to prioritize the implementation of ride searching and posting functionalities, as well as the chat feature. We recognized these features as fundamental to the functioning of the app, directly impacting users' ability to find and coordinate rides effectively. In contrast, we deferred the implementation of the live tracking feature, considering it an added benefit rather than a core necessity. While live tracking could enhance the user experience by providing real-time updates on the location of rides, we determined that it could be introduced in subsequent iterations without compromising the core functionality of the app.

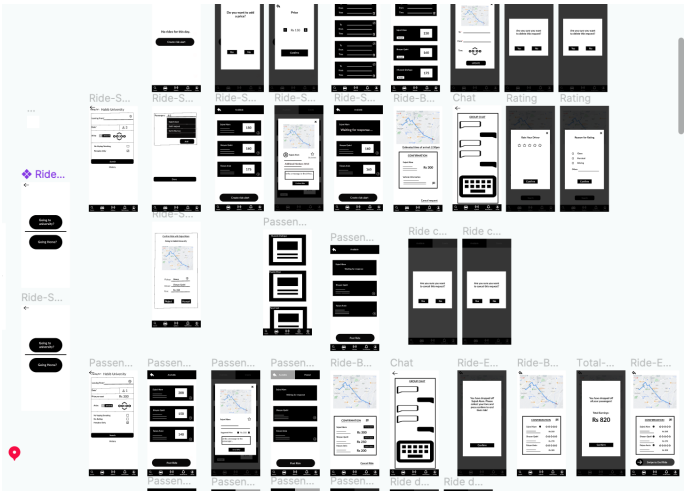

Figure 5: A snapshot of some of the wireframes

6.1.2 High Fidelity Designs

To transition from our prototyping phase to our high fidelity design phase for our carpooling app Saathi designed for university students, we used a rigorous approach centered around user feedback and regular improvements. Following the prototyping phase where we developed the basic flow and functionality of our app it was time to gain insights from a diverse group of users who experienced carpooling on a regular basis. The feedback was instrumental in providing insights to pain points, user preferences, and areas for enhancement.

We took these initial wireframes and used tools like Figma to shape them into high-fidelity designs that were enhanced with our vibrant theme colour #429e9d, clear user interfaces, and minimal visuals. High-fidelity prototypes replaced low-fidelity black-and-white designs, which improved the aesthetics and guaranteed a more user-friendly experience.

After the high-fidelity designs were finalised we conducted more testing in order to assess user comprehension and satisfaction. Users offered insightful feedback, emphasising those features that they understood and those that needed more explanation or required an easier approach. We iteratively improved the designs by adding changes to increase overall usability, streamline navigation, and improve clarity based on the input we received. We were able to refine the app's design through this iterative process, making sure it successfully met the needs of our target audience and resonated with features they required.

6.1.3 Deployment

Utilizing a microservices architecture, our mobile application required deploying several distinct services, each responsible for specific functions within our application. These services include user management, ride management, and the recommendation system. Each service is an independent Express.js application. Moreover, we wanted to build a continuous deployment pipeline in order to ensure that any changes made by our team were quickly reflected in the live deployment of our services. This meant that whenever an update was made, those changes seamlessly went live for our users. This approach aimed to minimize downtime and speed up the delivery of new features. The continuous deployment pipeline was essential because our project thrived on responsiveness to user needs. We wanted to adapt and evolve the application based on their feedback in real-time. By automating the deployment process, we could dedicate more time to improving the app and less time worrying about deployment logistics.

In order to achieve this and make the deployment process more streamline we leveraged Docker which is a containerization platform. Docker allowed us to encapsulate each individual service into a containerized image, containing all necessary dependencies for its execution. This approach ensured that each service could run reliably and consistently across different environments, regardless of the underlying infrastructure. Given the complexity of managing multiple services, we employed Docker Compose, a tool designed to simplify the orchestration and deployment of multi-container Docker applications. By crafting a unified Docker Compose file, we consolidated essential information for building images of each of our application's service in a single easy-to-read YAML file. This Docker Compose file specified the details of each service, including its build instructions, dependencies, networking configurations, and environment variables. Executing this single Docker Compose file initiated the creation of service images and the subsequent launch of their containers, orchestrating the complete backend infrastructure of our application. This initial setup was essential for building a Continuous Deployment pipeline after which the next step was to find a deployment platform capable of seamlessly translating this configuration into a live deployment environment. To make an informed decision, we conducted a comprehensive comparison of available options, considering factors such as support for Docker Compose, serverless capabilities, scalability, monitoring tools, pricing, and ease of integration.

Table 2 presents our findings, highlighting the strengths of each platform and ultimately leading to our selection of Koyeb as the preferred choice for deploying our backend services.

Feature	**Koyeb**	**Vercel**	**DigitalOcean**
Support for Docker Compose	Can interpret Docker Compose files directly	Limited support for Docker Compose	Limited support for Docker Compose
Serverless Deployment	Fully serverless deployment model	Limited serverless capabilities	Traditional VM-based deployment
Auto-Scaling	Automatic scaling based on workload	Automatic scaling for Next.js apps	Auto-scaling options available
Real-time Monitoring	Real-time monitoring of deployed services	Limited real-time monitoring capabilities	Monitoring tools available
Integrated Logging	Integrated logging and log management	Limited logging capabilities	Logging tools available
Pricing Model	Pay-as-you-go pricing model	Free tier available; usage-based pricing	Various pricing plans available
Platform Flexibility	Supports various programming languages and frameworks	Focuses primarily on JavaScript and Node.js	Supports multiple languages and frameworks
Integration Ecosystem	Integrates with a wide range of services and tools	Limited integration ecosystem	Extensive integration options available

Table 2: Comparison of Deployment Platforms

After our repository's integration with Koyeb, every commit made to our repository triggered Koyeb to interpret our Docker Compose file, facilitating the rebuilding of service images and the execution of their containers in real-time. This seamless integration empowered us to maintain a Continuous Deployment pipeline, ensuring rapid and efficient updates to our application's deployed backend services. Moreover, using Koyeb we assigned distinct ports to each service and monitored our application's logs in real-time, which enhanced the visibility and manageability of our deployed services. In summary, through the utilization of Docker and Koyeb, we successfully established a resilient and automated deployment infrastructure for our backend services, facilitating agile development and seamless updates.

The maps and navigation services utilized in our application are powered by Google Maps. To leverage these services, we established an application on the Google Cloud Platform. This platform served as the foundation for managing and monitoring the number of requests made to Google Maps services. By integrating with Google Cloud Platform, we ensured seamless access to the robust mapping functionalities provided by Google Maps. Moreover, the MongoDB database is hosted on Atlas, MongoDB's fully-managed cloud database service. Leveraging Atlas, our database infrastructure is already deployed in the cloud, eliminating the need for manual deployment. This cloud-based approach allowed us to focus on developing our application's features and functionalities while also offering scalability, reliability, and ease of management.

6.2 Admin Portal

6.2.1 Requirement

The admin portal stands as an integral part of our system, offering crucial functionalities for the effective management and optimization of our carpooling application. At its core, the portal serves as a central hub for administrators to access and analyze vital statistics essential for understanding the application's performance. By providing comprehensive insights into metrics such as total users, rides, and transactions, the admin portal empowers administrators to make informed decisions aimed at enhancing the application's efficiency and user experience. Moreover, the portal offers visibility into the ongoing activities within the application ecosystem. Administrators can access detailed information regarding each ride, including date, time, participants, and route details, enabling thorough monitoring and analysis of ride activity. Additionally, the admin portal offers comprehensive user management capabilities, allowing administrators to view user profiles, track their ride history, and efficiently manage user accounts. This ability to oversee and manage both user and ride data ensures a secure, reliable, and seamless experience for all application stakeholders.

6.2.2 Functionalities

The dashboard page of the admin portal serves as a comprehensive overview of the application's vital statistics, offering valuable insights into its performance and trends. At its core, the dashboard presents key metrics such as total transactions, total rides, and total users on the app, providing administrators with a clear understanding of the application's overall activity and growth. One of the standout features of the dashboard is its graphical representation of monthly transactions and rides over the past 12 months, enabling administrators to track trends and patterns over time. This historical data empowers administrators to identify growth opportunities, anticipate changes in demand, and make data-driven decisions to optimize resource allocation and operational efficiency. One of the significant benefits of the dashboard page is its focus on providing actionable insights through percentage increases compared to the previous month. This feature enables administrators to quickly assess performance trends and identify areas of improvement or concern. By highlighting month-to-month changes in key metrics, administrators can proactively address emerging issues, capitalize on growth opportunities, and steer the application towards continued success.

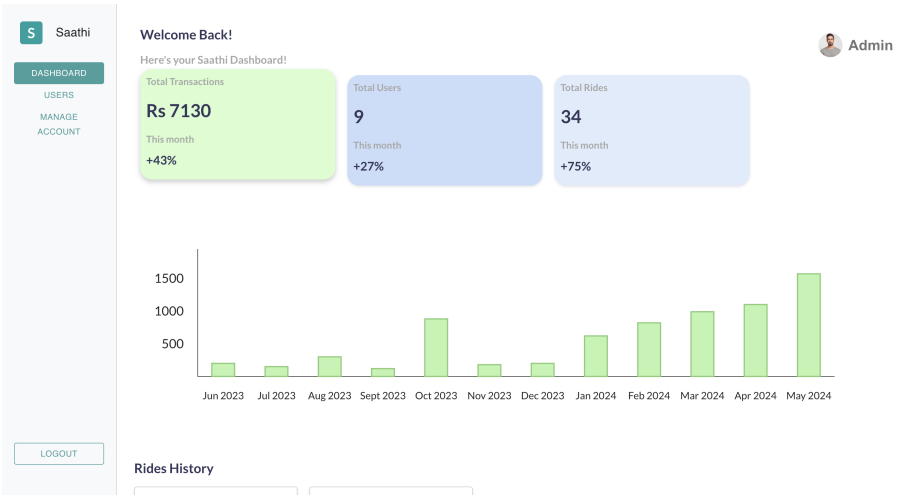

Figure 6: Admin Dashboard

In addition to providing insightful statistics, the admin dashboard offers a comprehensive list of all rides, sorted chronologically from most recent to oldest. Each ride entry includes essential details such as the driver's name, passenger names, source, destination, date, fare, and status. This organized presentation allows administrators to efficiently track ride activities, identify any anomalies or discrepancies, and address them promptly. Furthermore, the dashboard's search functionality enables administrators to quickly locate specific rides by filtering through driver and passenger names. This feature enhances usability and streamlines the process of retrieving relevant information, ultimately contributing to improved administrative efficiency and effective management of ride-related data.

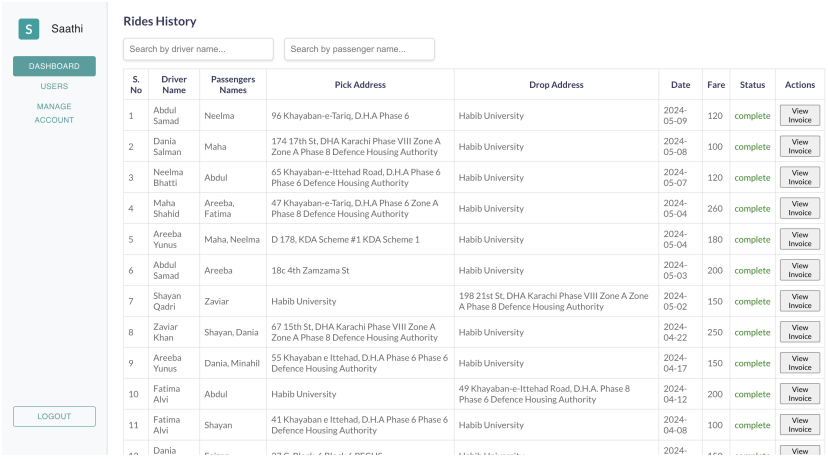

Figure 7: List Of Rides

On the Users page, administrators gain insight into all registered users on the application, alongside their ratings. They can efficiently search through the user list and access detailed profiles containing names, ratings, profile pictures, email addresses, contact numbers, and physical addresses. Additionally, administrators can delve deeper into individual user activities by examining the rides offered by each user. This includes comprehensive ride details such as passengers, source, destination, fare, date, and passenger ratings, enabling thorough monitoring of user interactions and ride engagements.

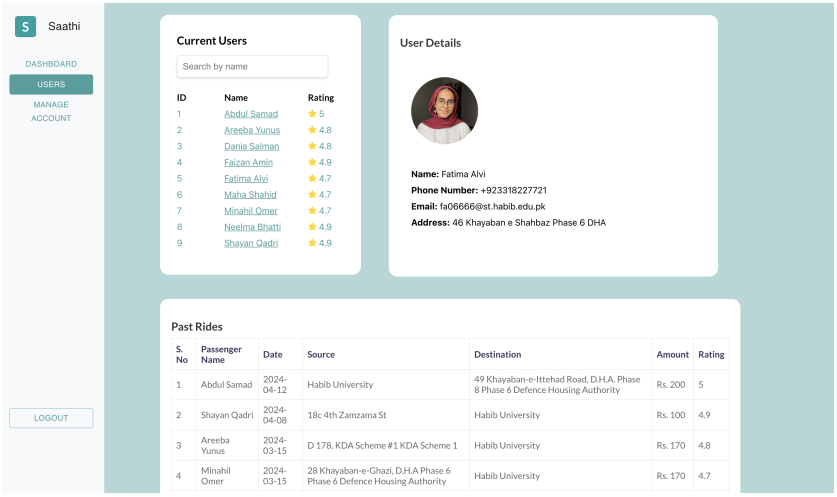

Figure 8: Users Page

The Manage Account page grants administrators the authority to delete user accounts by inputting the user's email address, along with their own email address and password for authentication. This feature ensures a secure and controlled process for managing user accounts, safeguarding the integrity of the platform and adhering to privacy regulations. By providing a straightforward mechanism for account deletion, administrators can swiftly address any compliance issues, security concerns, or user-related issues as they arise, thereby maintaining a safe and trustworthy environment for all platform users.

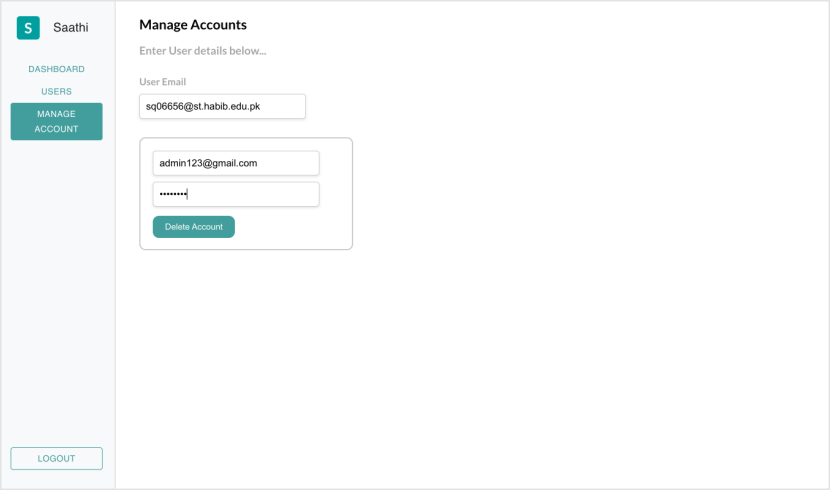

Figure 9: Manage Account

6.3 Recommendation System

6.3.1 Path Similarity

The core of Saathi's path similarity recommendation is based on the work of Micheal Oliveira da Cruz, particularly his methodology finding similarity between a rider and passenger. This approach allows us to address the unique requirements of carpooling, distinct from conventional ride-sharing services.

The initial step in our path similarity algorithm utilizes the Google Maps API to process user trajectories. This approach allows us to find the most efficient routes between the given origin and destination. By doing so, we can ensure that each path is represented using the smallest number of points necessary while still capturing all the information needed to represent the path accurately. Following this step, temporal filtering is applied to refine the selection of potential matches. This involves comparing the departure and arrival times of users to ensure that the matches are temporally compatible.

The next critical component is the application of the similarity algorithm, as derived from the work of Micheal Oliveira da Cruz. This algorithm allows us to identify potential carpooling matches by analyzing the spatial proximity between the trajectory of drivers and passengers. This drivers and passengers are grouped based on the alignment of their trajectory data points. Specifically, it evaluates whether the starting and ending point of a passengers route falls within a predefined distance from the points in the driver's trajectory. If a passenger's start and end points are found to be within a certain distance threshold from the driver's trajectory, it indicates a high potential for a successful carpool match. Conversely, passengers whose routes do not sufficiently overlap with that

of any driver's are not paired, ensuring that the system recommends only the most feasible and convenient matches.

In our similarity algorithm we have utilised the H3 Library to create a mapping technique which allows us to efficiently match drivers and passengers. The H3 library facilitates the partitioning of Earth into hexagonal cells at varying resolutions, creating a details grid overlay of the planet. We have used these hexagonal cells as clusters, where the longitude and latitude data points are mapped as indexes. The mapping process transforms raw geographical coordinates into a structured form of data, easily retrievable and comparable within our database system. These geographical clusters allow us to organise the spatial data and is the foundation for our path similarity algorithm.

Within our database, each ride registered-whether by a driver or passenger-is categorized and stored based on its H3 index. For drivers this involves mapping each point along their trajectory to corresponding hexagonal cells. Passengers, on the other hand, have only their origin and destination points stored. These mappings are stored in the database in the form of hash tables, taking advantage of the uniqueness of H3 indexes for effective data organization. The core of our matching processes lies in the utilisation of these Geo-spatial clusters. For drivers, to find potential passenger matches, the algorithm searches only those hexagons that intersect with the driver's trajectory. For passengers, matching involves searching the hexagons which correspond to both the starting and ending points of their journey. A successful match requires a driver to have data points in both these hexagons, ensuring both pick-up and drop-off points are within the driver's planned route.

Upon identifying all possible matches, the system then prioritised them based on the rerouting time for the driver, aiming to minimise any deviation from their original path. This approach ensures that the matches recommended to both drivers and passengers are not only practical but they also optimise the convenience and efficiency of the carpooling experience. This method streamlines the matching process, reduces computational overhead and enhances the overall user experience by providing timely, convenient and contextually relevant carpooling options.

6.3.2 User Similarity

Following the path similarity algorithm, the second step in Saathi's recommendation system focuses on user similarity. This aspect of the algorithm further refines the matches by assessing whether user characteristics and preferences are compatible. This is based on methodologies that leverage user profile data to enhance recommendation systems. Initially each user profile is transformed into a feature vector. These vectors consist of several attributes:

- **User Preferences:** Each user's specific preferences are encoded as binary values where '1' represents that a preference has been selected, and '0' denotes an unselected preference.
- **Frequency of Carpooling:** The frequency of carpooling represents how often a user participates in carpooling based on their historical data. Users who carpool frequently are more likely to continue doing so, making them ideal matches.
- **Feedback Scores:** High feedback scores given to users suggest that a user is reliable and preferable.

After creating these vectors, user similarity is found using cosine similarity to measure the compatibility between potential matches. Cosine similarity calculates the cosine of the angle between two vectors. It effectively quantifies how similar two users are based on their preferences and behaviors.

The resulting similarity scores are then weighted according to their importance. While path similarity carries the majority weight (80%), reflecting the critical importance of route compatibility, user similarity is weighted at 20%.

This balance ensures that while geographical feasibility is prioritized, the personal compatibility between users influences the final match as well.

The combined scores from path and user similarity analyses determine the final ordering of matches. This method allows Saathi to prioritize matches that not only minimize deviations from the intended routes but also enhance user satisfaction by pairing riders who are likely to enjoy shared rides based on similar preferences and positive historical interactions. This approach enhances the overall user experience by providing timely, convenient, and contextually relevant carpooling options.

7 Experiments/Results

7.1 Recommendation System

Evaluating clustering outcomes can be as challenging as the clustering process itself. Common methods for testing include internal and external evaluations. Internal evaluation allows us to summarize the clustering to a single quality score while external evaluation contrasts the clustering against an established ground truth. External evaluation, also known as manual evaluations is conducted by a human expert which assesses the clustering effectiveness in its practical application [24].

In internal evaluation of clustering, the approach focuses on scoring clusters based on the degree of similarity within a cluster compared to the similarity across different clusters. These methods aim to highlight clusters which are highly similar internally while being distinct from each other. This form of evaluation can allow us to differentiate the quality of clustering between different algorithms but it does not necessarily guarantee validity of the clustering results. While there are numerous methods for internal evaluations, such as Davies-Bouldin index and Dunn index, they all prove unsuitable in our context. Both methodologies calculate the ratio of intra-cluster distances to inter-cluster distances. However, given that our system utilizes pre-defined clusters that are directly adjacent to each other, the distances between any two clusters is consistently zero. This peculiarity renders traditional internal evaluation approaches which take intra-cluster distances into account as ineffective for our purposes.

In external evaluation, the effectiveness of clustering is assessed using data that was not involved in the clustering process such as predefined class labels or external benchmarks. This approach allows for the comparison of clustering outcomes against a set of know standards or classifications. It also providing a means to measure the accuracy and the relevance of the clustering in reference to some external frame of reference. Given the absence of labeled datasets specific to carpooling and the lack of comparable ride-hailing datasets within the context of Pakistan, external evaluation of our carpooling algorithm presents a unique challenge. To address this we believe it is necessary to generate our own dataset and manually label the data based on our understanding of carpooling dynamics.

We propose that one method of evaluation can be conducted based on the route distance and travel time between drivers and passengers within the same hexagons. This can be calculated precisely using the Google Maps API. By focusing on optimizing the distance and rerouting time across the various hexagons in our database we can demonstrate its effectiveness. Essentially, if our algorithm consistently results in shorter distances and a smaller rerouting time for the matches it suggests, it indicates that our approach to clustering within the carpooling context is successfully. This method not only offers a tangible measure of the algorithms performance but also aligns closely with the practical objective of carpooling-optimising routes for convenience and efficiency. Additionally, we have implemented a manual evaluation process within our system using the folium library to visually represent each

potential carpooling match, offering a visual validation of our quantitative findings.

Before evaluating the algorithm's performance, the first step involves optimising the resolution of hexagons. In this context, hexagon resolution refers to the spatial scale at which our path similarity algorithm divides the geographic area of Karachi. Within the H3 library we are able to choose different levels of resolution that determine the size of each hexagon. These can change from a few hundred meters to several kilometers in diameter. Higher resolutions mean smaller hexagons while lower feature larger hexagons. Optimizing the resolution with H3 allows us to accurately cluster drivers and passengers. It is important to achieve a balance between maximising the number of matches while minimising both rerouting time and distance. For instance, if the rerouting time is consistently less than a minute, there is a chance that very few riders are matching with each other since proximity is not guaranteed. One the other hand, if nearly every rider is matched but rerouting time averages over 30 minutes, the algorithm is prioritising matches at the cost of travel efficiency. In these scenarios, the algorithm optimizes only one factor, leading to sub-optimal results. Our objective is to avoid these extremes and find a balance that results in matches that are both sufficient in number and convenient, while minimising unnecessary travel time and distance.

To evaluate our carpooling similarity algorithm, we generated 400 random points across Karachi within the geographic boundaries of 24.75 to 25.05 degrees latitude and 66.9 to 67.15 degrees longitude. We created two sets of points: 200 heading towards Habib University and 200 departing from it. The points heading towards University were generated using a random seed of 42, while those departed were generated using a random seed of 2. We used two different random seeds so we could generate distinct set of coordinates for drivers and passengers to ensure consistent yet separate data for testing accurate pairing. These coordinates were populated into our database and we analyzed the resulting matches. This was done by measuring the distance and travel time from each driver to their corresponding passenger.

Testing was conducted across three different hexagon resolutions: 7, 8 and 9, as specified by the H3 library. According to the H3 Cell Statistics, higher resolutions correspond to smaller hexagons with shorter average edge lengths. The edge length of Resolutions 6, 7, 8, 9 and 10 were 3.72 km, 1.41 km. 0.53 km, 0.20 km and 0.08 km respectively. Requiring drivers to reroute by more than 3 km for pickup is unreasonable, and limiting matches within 100 meters would drastically reduce the number of matches. Given these considerations we chose resolutions 7, 8 and 9.

Table 3 provides the number of matches for different H3 resolutions, both to and from Habib University.

Resolution	**Number of Matches** (To Habib University)	**Number of Matches** (From Habib University)
7	624	650
8	144	217
9	8	34

Table 3: Number of Matches per Resolution

As there are 200 drivers and riders, the total number of pairs that could be made is $200 \times 200 = 40,000$. Analyzing the data presented in Table 3, we observe that resolution 7 has the highest number of matches, with 624 matches to Habib University and 650 matches from Habib University, totaling to 1,274 matches. Conversely, resolution 9 has the least number of matches, with only 8 matches to Habib University and 34 matches from Habib University, totaling to 42 matches.

Table 4 provides the average rerouting time for different H3 resolutions, both to and from Habib University.

Resolution	**Average Rerouting Time** (To Habib University)	**Average Rerouting Time** (From Habib University)
7	5.090 mins	7.312 mins
8	2.854 mins	3.521 mins
9	1.875 mins	1.676 mins

Table 4: Average Rerouting Time per Resolution

The data in Table 4 shows that resolution 7 has the longest average rerouting time, with an average of 5.090 minutes to Habib University and 7.312 minutes from Habib University. Conversely, resolution 9 shows the shortest average rerouting time, with an average of 1.875 minutes to Habib University and 1.676 minutes from Habib University.

Table 5 provides the average rerouting distance for different H3 resolutions, both to and from Habib University.

Resolution	**Average Rerouting Distance** (To Habib University)	**Average Rerouting Distance** (From Habib University)
7	1.924 Km	2.506 Km
8	0.968 Km	1.004 Km
9	0.399 Km	0.4608 Km

Table 5: Average Rerouting Distance per Resolution

Examining Table 5, we can observe differences in the average rerouting distances across different resolutions. Resolution 7 shows (like both tables above) the longest average rerouting distance, with distances of 1.924 kilometers to Habib University and 2.506 kilometers from Habib University. On the other hand, resolution 9 shows the shortest average rerouting distance, with distances of 0.399 kilometers to Habib University and 0.4608 kilometers from Habib University.

To provide a more detailed analysis of these variations, we have presented the data in a box plot. This allows us to highlight the central tendencies and variability of distance and time within each resolution. It also illustrates the presence of outliers. This allows us to evaluate each resolution's practicality for daily commuting.

At resolution 9, the interquartile range (IQR) for rerouting distance is between 0.001 km and 0.6 km for going to Habib as shown in Fig. 12 and 0.2 km to 0.7 km for going from Habib as shown in Fig. 10. The IQR for rerouting time is between 1.0 minute and 1.5 minutes for going to Habib as shown in Fig. 13 and 1.0 to 2.0 minutes for going from Habib as shown in Fig. 11. Although minimizing rerouting time is ideal for efficiency, excessively restricting matches dramatically reduces the number of feasible matches. For instance out of 40,000 potential matches, only 42 matches were identified. This scarcity underscores the limitations of overly stringent matching criteria.

At resolution 7, as shown in the figures below, the interquartile range is much broader and it contains distances and times which are generally considered acceptable for carpooling however they are pushing the upper limit of what drivers might tolerate. Notably there are a few outliers where rerouting time has reached 70.0 minutes and rerouting distance is almost 34.0 kms. Such extreme cases are rare however they highlight potential inefficiencies and risk of user dissatisfaction as lengthy detours are generally undesirable. Despite there being 1200 matches at this resolution, the presence of such outliers undermines the perceived adaptability of carpooling arrangements.

Resolution 8 presents a more balanced scenario. The IQR for both time and distances fit within an acceptable

range for everyday carpooling. The IQR for time is between 2.0 and 4.0 minutes to Habib as shown in Fig. 13 and between 1.0 and 6.0 minutes from Habib Fig. 11. The IQR for distance is between 1.0 and 1.2 kms to Habib as shown in Fig. 12 and between 0.5 and 1.5 kms from Habib Fig. 10. Outliers in this resolution also remain within a reasonable limit, suggesting that it offers a viable compromise between the number of matches and the extent of rerouting required. In conclusion, resolution 8 emerges as the most practical choice.

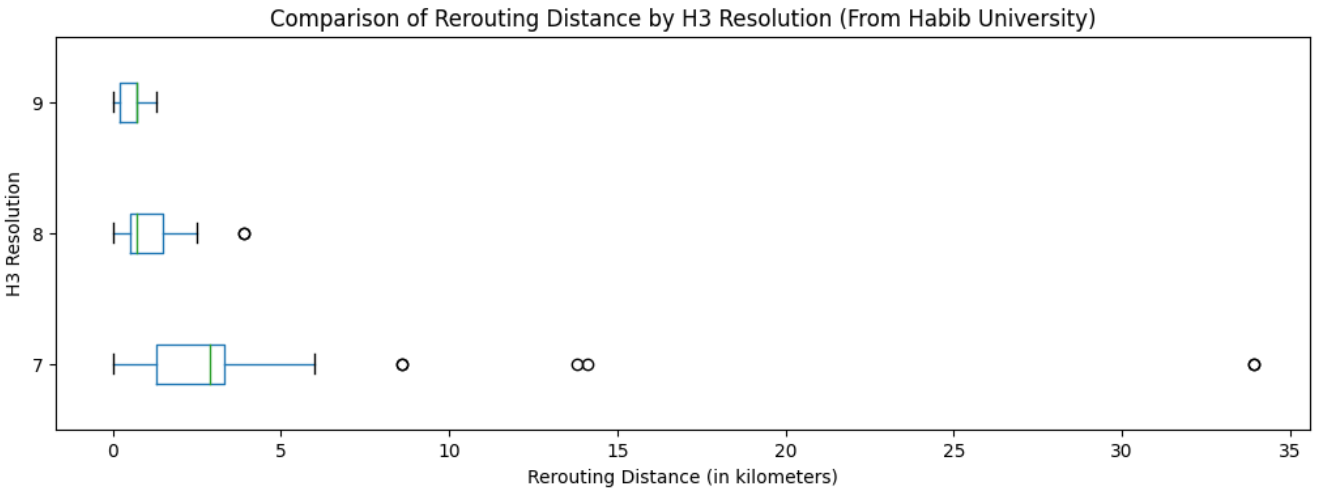

Figure 10

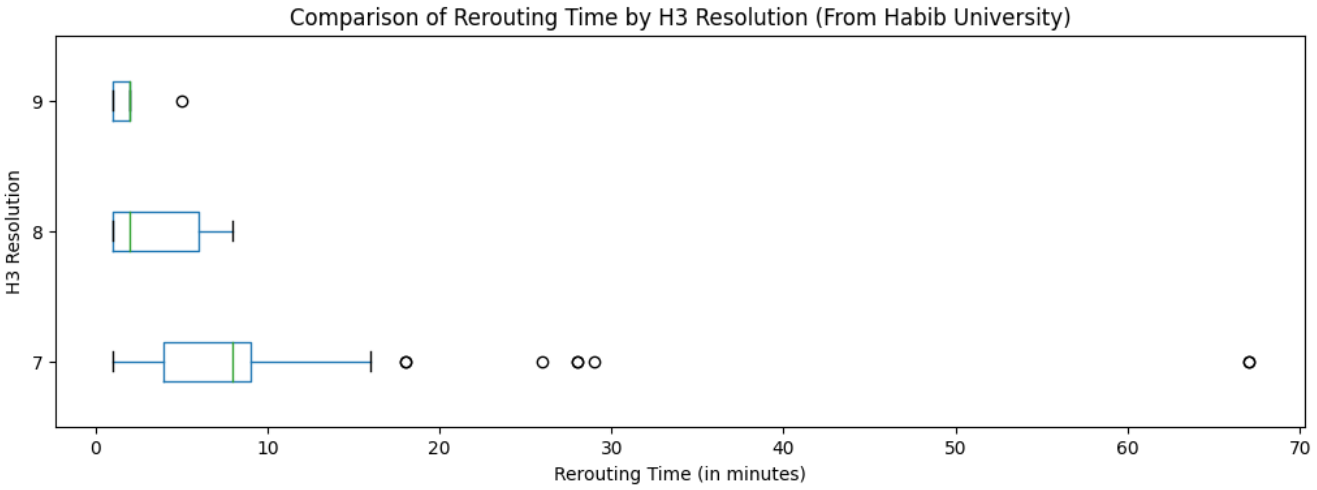

Figure 11

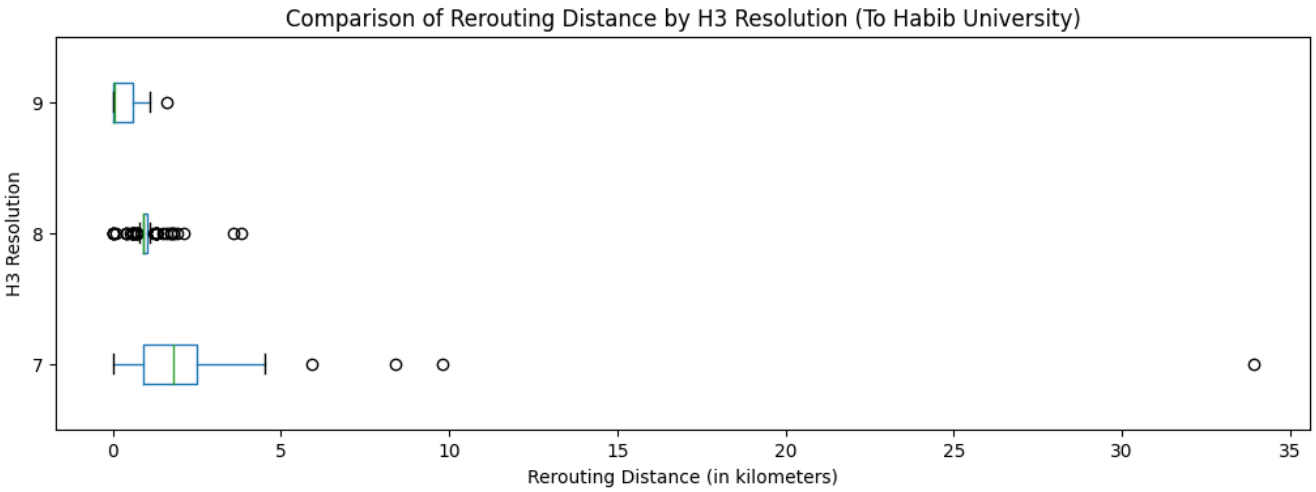

Figure 12

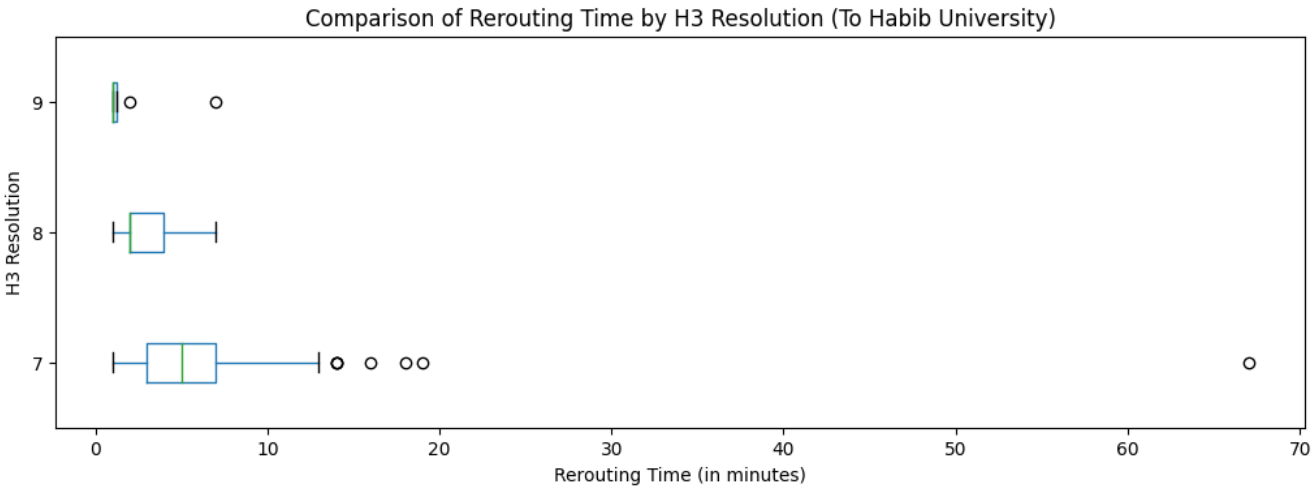

Figure 13

In addition to quantitative analysis we also conducted manual evaluation as a form of external validation to complement our computational findings. We used the Folium library to visualise every single match. This was used to assess the accuracy according to real-world carpooling logistics. Every match was tagged as accurate or not based on practical commuting scenarios. Given the sheer volume of potential pairs, manually reviewing all 40,000 combinations was impractical. Instead, our approach focused on examining riders in adjacent hexagons around each driver's location to identify additional plausible matches. Specifically, using resolution 7 for our analysis, we achieved an accuracy of 100% for the matches evaluated. It is crucial to understand that this assessment is subjective and based on our judgement and may not capture all nuances of predefined carpooling dynamics. This method of manual evaluation ensured that the matches were not only statistically valid but also practically feasible.

References

[1] M. O. Cruz, H. Macedo, and A. Guimaraes, "Grouping similar trajectories for carpooling purposes," in *2015 Brazilian Conference on Intelligent Systems (BRACIS)*, 2015. DOI: 10.1109/bracis.2015.36.

[2] P. Bellavista, P. Chen, H. Lv, S. Gao, Q. Niu, and S. Xia, "A real-time taxicab recommendation system using big trajectories data," *Wireless Communications and Mobile Computing*, vol. 2017, p. 5 414 930, Jul. 25, 2017. DOI: 10.1155/2017/5414930. [Online]. Available: https://www.hindawi.com/journals/wcmc/2017/5414930.

[3] R. B. Jadhao and J. M. Patil, "Recommendation system for carpooling and regular taxicab services," in *2017 International Conference on Inventive Systems and Control (ICISC)*, 2017, pp. 1–8. DOI: 10.1109/ICISC.2017.8068628.

[4] D. Zhang, T. He, Y. Liu, and J. A. Stankovic, "Callcab: A unified recommendation system for carpooling and regular taxicab services," in *2013 IEEE International Conference on Big Data*, 2013, pp. 439–447. DOI: 10.1109/BigData.2013.6691605.

[5] H. Qadir, O. Khalid, M. U. S. Khan, A. U. R. Khan, and R. Nawaz, "An optimal ride sharing recommendation framework for carpooling services," *IEEE Access*, vol. 6, pp. 62 296–62 313, 2018. DOI: 10.1109/ACCESS.2018.2876595.

[6] D. Lee and S. H. L. Liang, "Crowd-sourced carpool recommendation based on simple and efficient trajectory grouping," in *Proceedings of the 4th ACM SIGSPATIAL International Workshop on Computational Transportation Science*, ser. CTS '11, Chicago, Illinois: Association for Computing Machinery, 2011, pp. 12–17, ISBN: 9781450310345. DOI: 10.1145/2068984.2068987. [Online]. Available: https://doi.org/10.1145/2068984.2068987.

[7] G. Yatnalkar, H. Narman, and H. Malik, "An enhanced ride sharing model based on human characteristics and machine learning recommender system," *Procedia Computer Science*, vol. 170, pp. 626–633, Jan. 2020. DOI: 10.1016/j.procs.2020.03.135.

[8] Z. Xia, Y. Dong, and G. Xing, "Support vector machines for collaborative filtering," in *Proceedings of the 44th Annual Southeast Regional Conference*, ser. ACM-SE 44, Melbourne, Florida: Association for Computing Machinery, 2006, pp. 169–174, ISBN: 1595933158. DOI: 10.1145/1185448.1185487. [Online]. Available: https://doi.org/10.1145/1185448.1185487.

[9] M. O. d. Cruz, "On the similarity of users in carpooling recommendation computational systems," English, Graduate Program in Computer Science (PROCC), Master's thesis, Federal University of Sergipe, São Cristóvão, Feb. 2016, p. 83.

[10] K. Rajah. "Clustering based algorithms in recommendation system." (Feb. 2023), [Online]. Available: https://medium.com/@Karthickk_Rajah/clustering-based-algorithms-in-recommendation-system-205fcb15bc9b.

[11] *H3: Uber's Hexagonal Hierarchical Spatial Index*, https://h3geo.org/, Accessed: 2023-04-10.

[12] Atlassian, *Microservices vs Monolith*, Accessed: 2023-11-27, 2023. [Online]. Available: https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith.

[13] RubyGarage, *Best architecture for an MVP: Monolith, SOA, Microservices, or Serverless?* Accessed: 2023-11-27, 2023. [Online]. Available: https://rubygarage.org/blog/monolith-soa-microservices-serverless.

[14] Codvo, *Top 5 Challenges of Monolithic Architecture*, Accessed: 2023-11-27, 2023. [Online]. Available: https://www.codvo.ai/post/top-5-challenges-of-monolithic-architecture.

[15] A. W. Services, *What is Service-Oriented Architecture?* Accessed: 2023-11-27, 2023. [Online]. Available: https://aws.amazon.com/what-is/service-oriented-architecture/.

[16] CircleCI, *Polyglot vs Multi-Model Databases*, Accessed: 2023-11-27, 2023. [Online]. Available: https://circleci.com/blog/polyglot-vs-multi-model-databases/.

[17] N. Salaheddin and N. Ahmed, "Microservices vs. monolithic architectures [the differential structure between two architectures]," *MINAR International Journal of Applied Sciences and Technology*, vol. 4, pp. 484–490, Oct. 2022. DOI: 10.47832/2717-8234.12.47.

[18] DesignGurus, *Monolithic, Service-oriented, Microservice Architecture*, Accessed: 2023-11-27, 2023. [Online]. Available: https://www.designgurus.io/blog/Monolithic-Service-Oriented-Microservice-Architecture.

[19] Emizentech, *Uber Tech Stack: Software Architecture*, Accessed: 2023-11-27, 2023. [Online]. Available: https://www.emizentech.com/blog/uber-tech-stack-software-architecture.html.

[20] A. W. Services, *Careem Dynamodb Case Study*, Accessed: 2023-11-27, 2023. [Online]. Available: https://aws.amazon.com/solutions/case-studies/careem-dynamodb-case-study/.

[21] Appscrip, *Careem Tech Stack and Infrastructure*, Accessed: 2023-11-27, 2023. [Online]. Available: https://appscrip.com/blog/careem-tech-stack-and-infrastructure/.

[22] L. Engineering, *Scaling Productivity on Microservices at Lyft - Part 1*, Accessed: 2023-11-27, 2023. [Online]. Available: https://eng.lyft.com/scaling-productivity-on-microservices-at-lyft-part-1-a2f5d9a77813.

[23] G. Cloud, *BlaBlaCar Customer Story*, Accessed: 2023-11-27, 2023. [Online]. Available: https://cloud.google.com/customers/blablacar.

[24] R. Feldman and J. Sanger, *The Text Mining Handbook: Advanced Approaches in Analyzing Unstructured Data*. Cambridge Univ. Press, 2007, ISBN: 978-0521836579.