Carpooling Application for UTAR Kampar Student

BY

Tan Jian Hua

A REPORT

SUBMITTED TO

Universiti Tunku Abdul Rahman

in partial fulfillment of the requirements

for the degree of

BACHELOR OF COMPUTER SCIENCE (HONOURS)

Faculty of Information and Communication Technology

(Kampar Campus)

Jun 2025

COPYRIGHT STATEMENT

ACKNOWLEDGEMENTS

I would like to express thanks and appreciation to my supervisor, Dr. Ng Hui Fuang and my moderator, Mr. Tan Chiang Kang who have given me a golden opportunity to involve in the web-based application field study. Besides that, they have given me a lot of guidance in order to complete this project. When I was facing problems in this project, the advice from them always assists me in overcoming the problems. Again, a million thanks to my supervisor and moderator.

Other than that, I would like to thank my project teammate, Tay Kai Sheng who has provided a lot of assistance to me when completing this project. Although both of us are having different project and task scope, he is still willing to support me when I faced difficulties in developing this project.

ABSTRACT

This project lies within the field of web-based application development, specifically targeting intelligent carpooling systems for university communities. It focuses on the design and implementation of a carpooling platform tailored for UTAR Kampar students, addressing the lack of a centralized, reliable, and affordable car-sharing solution. The primary objective is to provide a cost-effective alternative to commercial ride-hailing services like Grab by facilitating a student-exclusive platform to offer and request rides based on real-time and recurring schedules. Key features include user registration, ride listings, bookings, and ride management, with Google Maps API integration for geolocation and route assistance. Dialogflow is employed to deliver an AI-powered chatbot that helps users search for available rides through natural language interaction. A key novelty introduced in the second phase is timetable-based ride creation and search, which allows students to auto-generate recurring ride offers based on their weekly class schedules. This feature significantly reduces manual input, increases consistency in ride availability, and streamlines the carpooling experience by aligning with students' academic timetables. The project followed the Agile methodology throughout the Software Development Life Cycle (SDLC), incorporating iterative development, continuous feedback, and incremental improvements. The implemented prototype has been tested to enable smooth interaction between drivers and passengers, improve time efficiency in finding suitable rides, and foster a stronger community-based transportation culture. Conclusively, this system has shown promising results in reducing transportation friction among students and introduces a novel approach by integrating academic timetables with carpool scheduling—a unique feature not commonly found in existing ride-sharing platforms.

Area of Study (Minimum 1 and Maximum 2): Web application development

Keywords (Minimum 5 and Maximum 10): Carpooling, Web Application, Transportation, AI Chatbot, Management, Optical Character Recognition, Timetable

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS

s	seconds
%	percentage
>	bigger than
<	Smaller than

LIST OF ABBREVIATIONS

AI	Artificial Intelligent
AJAX	Asynchronous JavaScript and XML
API	Application Programming Interface
CDN	Content Delivery Network
CRUD	Create, Read, Update, Delete operations
CSS	Cascading Style Sheets
ERD	Entity Relationship Diagram
ETA	Estimated Time Arrival
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
ID	Identification
IDE	Integrated Development Environment
JSON	JavaScript Object Notation
MVC	Model View Controller
NLU	Natural Language Understanding
OCR	Optical Character Recognition
ORM	Object-Relational Mapper
PHP	Hypertext Preprocessor
SDLC	Software Development Life Cycle
SUS	System Usability Scale
SQL	Structured Query Language
UTAR	University Tunku Abdul Rahman
UI	User Interface

Chapter 1
Introduction

In this chapter, it provides an overview of the project by presenting the motivation and problems that led to the development of the system. It explains the scope and objectives of the project, followed by the potential impact and contributions that may bring to the target users. Moreover, it includes a brief explanation of the organization of the report.

1.1 Problem Statement and Motivation

Currently, UTAR Kampar students who are looking for carpooling rely on social media platforms such as RedNote and WhatsApp to find and communicate with the potential drivers or riders. Typically, students will post their academic timetables or a specific ride request and wait for others to reach them. While this approach may work for sometimes only, it is often inefficient and unorganized [1]. Drivers may be overwhelmed with multiple messages from different riders, making it difficult to manage their rides. On the other hand, riders may not always find a suitable driver due to timing mismatches or lack of clear information, especially during off-peak hours like midnight or early morning.

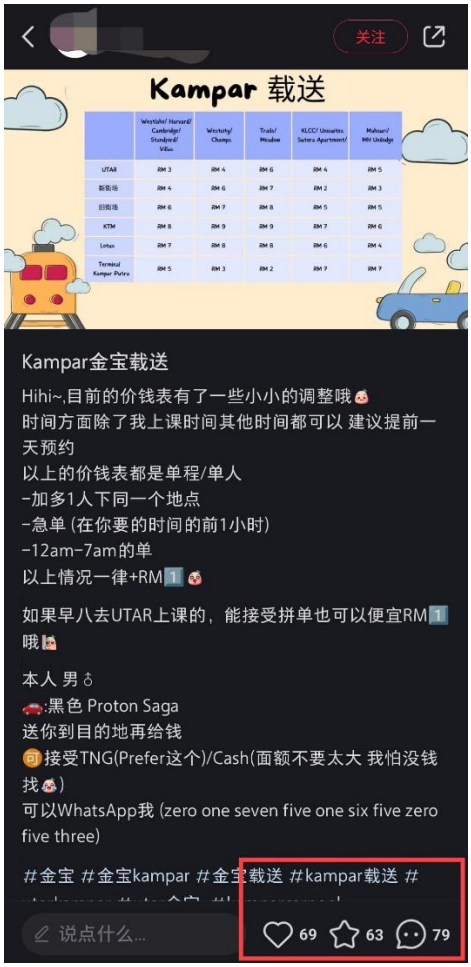


Figure 1.1 Carpool ride offer in RedNote

Moreover, paid transportation services like Grab are commonly used but often come with high charges [2] and are not always available when needed, especially during low-demand hours. Public transport such as buses also operates on fixed schedules, which does not cater to sudden changes like unexpected class cancellations.

The motivation behind this project comes from real observations on social media, where there are many students actively seeking carpooling opportunities by posting their schedules or urgent ride requests. These posts often receive multiple comments, showing clear demand and interest from the student community.

1.2 Objectives

The primary objective of this project is to develop a dedicated web-based carpooling platform for UTAR Kampar students to simplify the process of finding and offering shared rides. By centralizing carpool requests and offers into a single system, the platform aims to reduce

reliance on informal channels such as social media and provide a more organized, efficient, and user-friendly solution tailored to the student community.

Key goals of the project include:

- **Facilitating easy ride creation and booking** through an intuitive interface, allowing users to post ride offers, request rides, and manage their bookings with minimal effort.
- **Encouraging pre-planned ride arrangements** by using **schedule-based listings** instead of real-time matching, supporting students who plan their travel in advance based on class schedules.
- **Introducing timetable-based ride creation and searching**, a novel feature that allows students to automatically generate recurring rides and search for rides based on their weekly academic timetable, significantly reducing manual input and improving consistency.
- **Integrating an AI-powered chatbot** using Dialogflow to help users find suitable rides quickly through natural language input (e.g., departure, destination, date).
- **Ensuring simplicity and usability** for both drivers and riders, without the complexity of administrative functions, payments, or real-time GPS tracking, which are outside the current scope and may be considered in future mobile-based versions.
- **Focusing the system on the UTAR Kampar campus** while allowing for potential future expansion to nearby areas based on demand.

Through these objectives, the project seeks to enhance the overall carpooling experience by reducing communication gaps, improving the visibility of available rides, and providing a structured, convenient platform tailored to student needs.

1.3 Project Scope and Direction

This project aims to deliver a functional web-based carpooling platform specifically designed for UTAR Kampar students. The system provides a dedicated and organized environment where students can offer or request car rides, with a focus on planning rides based on academic schedules.

By the end of the project, a working prototype will be developed, allowing users to:

- Register and log in to the system
- Post ride offers and request rides
- Book available rides
- View and manage their carpooling activities

A key enhancement introduced in this phase is the **timetable-based ride creation and search feature**, which enables students to auto-generate recurring rides according to their weekly class schedules. This allows for more consistent ride availability and reduces the need for manual entry.

The system is developed using **PHP Laravel** for the backend, **MySQL** as the database, and **BladewindUI** with **Tailwind CSS** for the frontend interface. It also integrates the **Google Maps API** to support address autocomplete and location handling, and **Dialogflow** to power a chatbot that helps users search for rides through natural language queries.

The scope of the project is focused on essential ride-sharing functionalities for academic use and excludes features such as real-time GPS tracking, payment handling, and administrative tools, which may be considered in future mobile-based or expanded versions. The current system is scoped exclusively for the UTAR Kampar student community, but future directions may include expanding service coverage to nearby areas and adding features like ride statistics, email reminders, and driver/rider performance insights.

1.4 Contributions

This project provides a practical and impactful solution for UTAR Kampar students by addressing a common transportation challenge: the lack of affordable and reliable commuting options to and from campus [3]. With the rising cost of ride-hailing services like Grab and limited on-campus parking, many students struggle to find convenient transportation. By offering a dedicated web-based carpooling platform, this project helps students save time, reduce travel expenses, and simplify ride arrangements within a trusted academic community.

Beyond personal convenience, the platform also promotes broader environmental and social benefits [4]. By encouraging shared rides, it contributes to reduced traffic congestion and lower carbon emissions in the campus area. It also opens opportunities for students to earn a small side income by offering rides, fostering a mutually beneficial ecosystem among peers.

One of the key contributions of this project is the **timetable-based ride creation and search feature**, which allows students to generate and find recurring rides based on their academic schedules. This automation improves ride availability, minimizes manual effort, and aligns carpooling more closely with students' daily routines—making it a unique feature not typically found in general-purpose ride-sharing platforms.

Compared to informal carpool arrangements via social media or messaging apps, this platform centralizes all ride-sharing activities into a structured, easy-to-use system. It provides clear listings, a booking interface, ride management tools, and AI-driven ride search via chatbot, all tailored specifically to student needs. As the system matures, it has the potential to become a daily tool for the UTAR community and could be expanded to nearby regions or adapted for other institutions facing similar transportation issues.

1.5 Report Organization

This report is organized into 7 chapters: Chapter 1 *Introduction*, Chapter 2 *Literature Review*, Chapter 3 *System Methodology*, Chapter 4 *System Design*, Chapter 5 *System Implementation*, Chapter 6 *System Evaluation and Discussion*, and Chapter 7 *Conclusion*. Chapter 1 introduces the project background, problem statement, objectives, scope, and contributions. Chapter 2 reviews related works, existing systems, and technologies that inform the development of the proposed solution. Chapter 3 presents the methodology adopted, including the development approach, tools, and techniques. Chapter 4 discusses the system design, covering architecture, database design, and user interface planning. Chapter 5 describes the system implementation, detailing the development process and integration of core features. Chapter 6 evaluates the system through testing, performance analysis, and objective assessment, while also highlighting challenges encountered. Finally, Chapter 7 concludes the project by summarizing the findings, presenting recommendations for future improvement, and reflecting on the overall outcomes.

Chapter 2
Literature Review

2.1 Review of Technologies

2.1.1 Backend Framework

Laravel is a widely adopted open-source PHP framework designed for building robust and maintainable web applications [5]. It follows the **Model-View-Controller (MVC)** architectural pattern, which promotes clear separation of concerns, making code more organized and easier to manage. Laravel was selected as the backend framework for this project due to its comprehensive feature set, developer-friendly syntax, and strong community support.

Laravel offers built-in support for common tasks such as **routing, authentication, form validation, database migrations**, and **session management**, which significantly reduces the amount of boilerplate code required during development. These features proved especially useful in implementing core functionalities of the carpooling platform, such as **user registration and authentication**, **ride creation**, **booking management**, and **timetable-based recurring ride logic**.

The use of **Eloquent ORM (Object-Relational Mapping)** in Laravel allowed for seamless interaction with the MySQL database, enabling intuitive database operations through expressive PHP syntax. This made it easier to design and manage database relationships between entities such as users, rides, bookings, and ride schedules.

Furthermore, Laravel's **Artisan command-line interface** was used for generating boilerplate code, running migrations, and seeding the database with dummy data for testing purposes. This streamlined the development workflow and accelerated the implementation of features.

Laravel's **middleware system** also played an important role in securing routes and ensuring that only authenticated users could access sensitive features such as ride posting and booking. In addition, **Laravel's validation engine** was used extensively to ensure that user inputs, such as ride details and timetable entries, were consistent and error-free.

In this project, Laravel was chosen over other backend frameworks such as **Node.js with Express.js** or **Python with Django** due to its strong support for web-centric development, especially when working with relational databases and monolithic server-side applications [6].

Laravel's built-in tools such as **Artisan CLI**, **Blade templating engine**, **Eloquent ORM**, and **middleware** streamline the development of complex features like user authentication, ride listings, and timetable-based recurring ride logic — all of which are core to this carpooling system.

Overall, Laravel provided a solid foundation for developing a scalable and maintainable backend system, capable of handling the business logic and data management required for a student-focused carpooling platform.

2.1.2 Database System

MySQL is one of the most widely used open-source relational database management systems (RDBMS) and is a common choice for web-based applications. It is known for its reliability, performance, scalability, and compatibility with a wide range of programming languages and frameworks, including PHP and Laravel. In this project, MySQL was selected as the database system due to its strong support for structured data, ease of integration with Laravel's Eloquent ORM, and efficient handling of relational data models.

Laravel's Eloquent ORM abstracts database interactions into intuitive PHP syntax, allowing developers to define and manage relationships between tables such as users, rides, bookings, and timetable-based recurring ride patterns. This simplifies the process of querying, inserting, and updating data, thereby increasing developer productivity and reducing the risk of SQL-related errors.

MySQL also provides robust support for **ACID (Atomicity, Consistency, Isolation, Durability)** properties, ensuring data integrity throughout ride creation, booking transactions, and user account operations. Its indexing and query optimization capabilities help improve the performance of complex search features, such as filtering rides by departure time, destination, and timetable alignment.

In the context of this carpooling platform, MySQL was used to store and manage all core data entities, including user profiles, ride offers, ride bookings, and recurring rides derived from academic timetables. These tables were connected through foreign keys, with proper indexing to support fast retrieval, especially for timetable-based ride search functions.

Compared to other database systems such as PostgreSQL or NoSQL alternatives like MongoDB, MySQL was chosen due to its maturity, simpler learning curve, and extensive

documentation [7]. While PostgreSQL offers more advanced features like full-text search and complex data types, these were not essential for the current scope of the project. Likewise, NoSQL systems were not considered necessary due to the structured and relational nature of the system's data.

2.1.3 Front-end Libraries

The front-end of this carpooling platform is developed using **Tailwind CSS**, a utility-first CSS framework, and **BladewindUI**, a Laravel-specific UI component library [8]. These technologies were chosen to enhance user experience, speed up UI development, and maintain design consistency across the web application.

Tailwind CSS offers a highly customizable, low-level utility-based approach to styling, enabling developers to design interfaces directly within the HTML structure. Unlike traditional CSS frameworks such as Bootstrap, which rely heavily on predefined components and class names, Tailwind allows for more flexibility and modular control over styling. This was particularly useful for building a modern, responsive interface tailored to the needs of UTAR students, such as ride listing pages, booking forms, and timetable-based ride management views.

In addition, Tailwind's responsiveness and mobile-first design philosophy ensure that the application is usable on various devices, which is crucial for students accessing the platform via laptops or smartphones.

BladewindUI complements Tailwind by providing ready-to-use Laravel Blade components that follow Tailwind's utility-first design approach. It simplifies the implementation of forms, alerts, modals, and other UI components, reducing the need to write repetitive HTML. This helped accelerate front-end development while maintaining a clean and consistent look across the application.

The combination of Tailwind CSS and BladewindUI was chosen over alternatives like Bootstrap, Material UI, or custom CSS because of their **seamless integration with Laravel**, lightweight nature, and developer efficiency. While Bootstrap offers faster prototyping, it imposes more rigid styles and often requires additional overrides, which can complicate customization. Tailwind, on the other hand, promotes a more maintainable and scalable design system suited for evolving projects.

2.1.4 AI Chatbot

As part of enhancing user experience and accessibility, an AI-powered chatbot was integrated into the carpooling platform using **Google Dialogflow**, a Natural Language Understanding (NLU) platform developed by Google [9]. Dialogflow enables applications to interpret user input in natural language and respond intelligently through conversational interfaces. In the context of this project, the chatbot assists users in searching for available rides by accepting inputs such as departure location, destination, and preferred date or time.

Dialogflow was chosen for its **ease of integration with web applications**, support for multiple languages, and its ability to deploy across various platforms (e.g., web, mobile, social media). Its **intent-based architecture** and built-in machine learning capabilities allow developers to define user intents (e.g., "Find a ride to Ipoh tomorrow") and entity extraction (e.g., recognizing locations and dates), which are critical for enabling intelligent, contextual ride search.

In recent years, the use of conversational interfaces in web-based systems has grown significantly due to their ability to improve user engagement and reduce friction in data input. Studies such as *"Chatbot Integration in Few Patterns"* highlight the role of chatbots in enhancing usability, automating responses, and simplifying user navigation in information systems. In particular, chatbots in educational or campus systems have been used to streamline services such as class schedules, FAQs, and transport arrangements, making them a suitable fit for the UTAR student community [10].

Dialogflow's webhook support also allows the chatbot to interact directly with the Laravel backend, enabling dynamic responses based on real-time data such as available rides, matched destinations, or filtered search results. This contributes to a more seamless and interactive user experience compared to traditional keyword-based search forms.

Compared to other chatbot frameworks like **Microsoft Bot Framework**, **Rasa**, or **IBM Watson**, Dialogflow was selected for its **low learning curve**, **integration with Google Cloud services**, and the availability of pre-built agents that accelerate development. While Rasa offers more customization and on-premises control, it requires significantly more setup and training data, which is beyond the scope of this academic project.

2.1.5 Maps and Location Services

Location-based services are essential in modern transportation and carpooling systems, providing users with accurate address input, route visualization, and spatial decision-making capabilities. This project integrates **Google Maps API** to handle various location-related functionalities, including address autocomplete, geolocation, estimated distance/time calculations, and interactive map displays [11].

Google Maps Platform offers a comprehensive suite of APIs—such as Places API, Directions API, and Geocoding API—that are crucial for building map-intensive web applications. In this project:

- **Google Places API** is used to suggest valid locations during ride creation to avoid typos or ambiguous entries.
- **Directions API** calculates and visualizes optimal routes between pickup and destination points.
- **Geocoding API** helps convert between human-readable addresses and geographic coordinates, improving the reliability of stored ride data.
- **Maps JavaScript API** embeds interactive maps on the ride listing and ride detail pages, providing a familiar and intuitive interface for users.

Interactive maps significantly improve user satisfaction in transportation applications, particularly when users are involved in planning their own routes. The use of Google Maps ensures consistency, accuracy, and trustworthiness in navigation, which are all critical for a student-based carpooling system where trust and clarity are vital.

Compared to open-source alternatives such as **OpenStreetMap (OSM)** or **Leaflet**, Google Maps was chosen for its superior data quality, global coverage, and integration with other Google services (e.g., Dialogflow and Firebase, if needed). While OSM offers better cost efficiency and privacy control, it lacks the ease-of-use and completeness required for rapid development in an academic setting.

Map APIs play a core role in route optimization and estimated fare calculation. Although this project does not implement real-time GPS tracking or live ETA updates, the pre-scheduled nature of rides still benefits significantly from reliable mapping features.

2.1.6 Optical Character Recognition

Optical Character Recognition (OCR) is a technology that enables the conversion of scanned or photographed documents into machine-readable text. In the context of this project, OCR is used to extract structured information from students' timetables, allowing for **automated ride creation** based on academic schedules.

Among available OCR tools, **PaddleOCR**—developed by Baidu as part of the PaddlePaddle deep learning framework—was selected due to its balance between **accuracy**, **speed**, and **support for multiple languages and document layouts** [12]. It provides ready-to-use pipelines for detection, recognition, and layout analysis, making it suitable for structured documents like university timetables.

Key Justifications for Choosing PaddleOCR:

- **Accuracy on Tabular Data**: PaddleOCR's layout analysis and support for multi-line and columnar text make it well-suited for parsing timetables with complex structures.
- **Speed & Lightweight Deployment**: Unlike larger models such as Tesseract with layout plugins or heavy cloud-based solutions, PaddleOCR provides efficient inference, especially when containerized using Docker.
- **Customizability**: It allows for easy fine-tuning and integration with post-processing logic for domain-specific tasks like identifying classroom codes, lecture durations, or recurrence patterns.

In this project, PaddleOCR plays a key role in enabling students to upload timetable screenshots, which are then parsed to extract subjects, locations, times, and days. These parsed values are converted into ride requests or ride offers with recurring patterns, reducing manual data entry.

Alternative Considerations:

Table 2.1 Comparison of OCR services

Technology	**Strengths**	**Weaknesses**
Tesseract OCR	Well-documented, open source	Poor with tables and layout-sensitive data

Technology	Strengths	Weaknesses
Google Cloud Vision OCR	High accuracy, scalable	Paid service, requires internet access, less control
PaddleOCR	Accurate, customizable, open-source, fast	Requires Python/Docker knowledge for deployment

Thus, PaddleOCR is aligned with the system's goal of offering a **low-cost, student-friendly** solution without relying on expensive third-party services or inaccurate text extraction.

2.1.7 Summary of The Technologies Review

This project integrates a variety of technologies, each selected based on a combination of technical suitability, development efficiency, and relevance in current academic and industry practices. For the **backend framework**, Laravel (PHP) was chosen due to its elegant MVC architecture, built-in features for routing, authentication, and security, as well as strong community support and documentation. Compared to alternatives such as Node.js with Express or Django (Python), Laravel provides faster scaffolding for web applications, making it particularly well-suited for academic and small-to-medium scale systems like this carpooling platform.

The **database system** employed is MySQL, a widely used relational database management system. MySQL supports structured data, enforces data integrity, and works seamlessly with Laravel's Eloquent ORM. While alternatives like PostgreSQL offer advanced features, MySQL's performance, simplicity, and widespread hosting support made it a practical choice for this project.

On the **frontend**, Tailwind CSS was adopted as the primary styling framework due to its utility-first approach that promotes consistency and flexibility without relying on rigid, predefined components like Bootstrap. This was paired with **BladewindUI**, a component library designed specifically for Laravel Blade. BladewindUI simplifies interface development by offering reusable UI components, which reduced development time and improved the visual quality of the system.

For the **AI chatbot**, Dialogflow was integrated to allow users to search for rides using natural language queries. Dialogflow stands out for its ease of integration, built-in NLP capabilities, and compatibility with webhook services, making it suitable for implementing a conversational interface in this academic setting. Other frameworks like Rasa or IBM Watson were considered but were more complex or resource-intensive for the project's scale.

The system also uses the **Google Maps Platform** to enhance location-based functionalities. APIs such as Autocomplete, Distance Matrix, and Directions provide accurate geolocation, estimated travel time, and map-based interaction. Google Maps was favored over alternatives like HERE Maps and OpenStreetMap due to its accuracy, extensive documentation, and feature richness.

An additional enhancement involves the use of **PaddleOCR** to extract structured data from student timetables. PaddleOCR offers high accuracy in recognizing both printed and tabular text, which is vital for parsing academic schedule formats. It was preferred over traditional tools like Tesseract OCR due to its multilingual support and better performance on complex layouts.

To manage the OCR system efficiently, **Docker** was utilized to containerize the PaddleOCR microservice. This allows the model to be hosted separately and accessed through FastAPI, reducing the need to reload the model for every request and improving response times. Compared to using virtual environments or local scripts, Docker ensures consistent deployment and easier maintenance.

In conclusion, the selected technologies form a robust stack that addresses the functional, usability, and performance requirements of the system. Each component was carefully chosen through comparative evaluation to balance development complexity, scalability, and user experience. This ensures that the final product is both technically sound and academically justifiable.

2.2 Review of Existing Systems

2.2.1 WeRide

WeRide [13] is a free carpooling platform, which aims for users in Malaysia and Singapore. It mainly serves as a digital meeting place for drivers and passengers who are interested in sharing a ride, especially for daily commutes and long-distance travel. The platform’s mission is to

reduce the number of single-occupancy vehicles on the road, thereby lowering traffic congestion, cutting travel costs, and promoting environmental sustainability through carpooling. It operates on a user-driven model where individuals can publish their own travel plans, either offering or requesting rides, and directly communicate with each other to coordinate the trip.

Figure 2.1 WeRide app logo

The platform supports both one-time and recurring trips, offering flexibility for users with different commuting patterns. It encourages users to create profiles that include basic details such as names, profile pictures, and links to their Facebook accounts to help increase trust among participants. Ride-related communication is facilitated via WhatsApp, making it easy for users to negotiate timing, pick-up points, and cost-sharing arrangements.

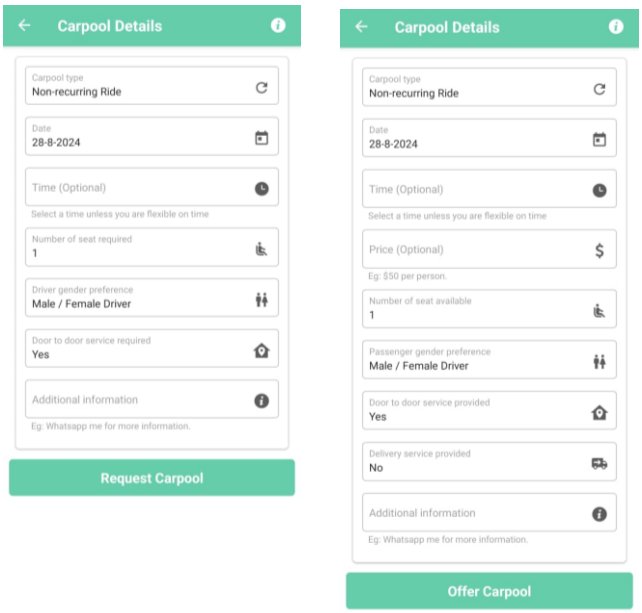


Figure 2.2 WeRide request and offer form

Additionally, WeRide provides extra features such as access to traffic cameras across major highways in Malaysia, which helps users better plan their journeys by checking real-time road

conditions. This extra layer of functionality gives WeRide an edge in terms of situational awareness and convenience for long-distance commuters.

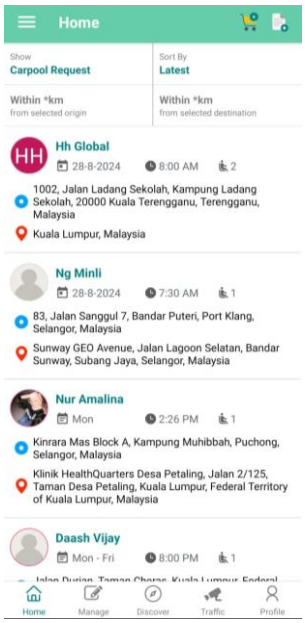


Figure 2.3 WeRide search interface

Despite these features, WeRide does not operate as a traditional ride-hailing service like Grab or Uber. It does not involve live driver availability matching, built-in payment systems, or real-time ride tracking. Instead, it places responsibility on users to initiate contact and organize their own rides, making it more of a self-service community platform than a full-fledged transportation service.

WeRide offers several notable strengths that make it a useful carpooling platform for users across Malaysia and Singapore. One of its key advantages is its wide **geographical coverage**, which allows for cross-border carpooling opportunities, something that enhances its appeal to users traveling long distances. Additionally, the platform provides **user autonomy**, enabling users to make their own decisions when arranging rides, which promotes flexibility and aligns with individual preferences [14]. WeRide also incorporates additional features such as **integration with traffic cameras**, offering real-time traffic images to help users plan their routes more efficiently. To enhance trust and safety, the platform uses **user verification methods** by linking Facebook profiles and WhatsApp messaging, which adds a layer of identity assurance during ride coordination.

Despite its strengths, WeRide has several limitations that affect its effectiveness—especially in a student context. One major drawback is its **lack of structured scheduling features**; the

platform does not support functionalities that align with fixed academic schedules, making it less suitable for students with regular class timetables. Moreover, as the user base grows, WeRide lacks **robust ride management tools**, making it difficult for users, especially drivers, to efficiently handle multiple ride's offers or requests. Lastly, the platform does not incorporate **AI-driven assistance**, which could help users find optimal ride matches or streamline communication. This absence of intelligent matching or management may reduce the platform's convenience as the volume of users increases.

While WeRide serves a broad audience across Malaysia and Singapore, the proposed **UTAR Carpooling Web Application** is designed specifically to address the unique needs of UTAR Kampar students. One of the key differentiators is the **integration of academic schedules**, allowing users to post and search for rides based on their class timetables, leading to more convenient and relevant ride matches. The system also features an **AI-powered chatbot using Dialogflow**, which streamlines the process of finding and booking rides, thereby improving overall user experience. In addition, the platform includes **dedicated ride management tools** that enable users to effectively monitor their ride activities, such as viewing visual statistics for upcoming trips, booked rides, and estimated profits for drivers. Most importantly, the application is tailored for a **specific community**, fostering a more trusted environment where users share similar daily routines and transportation goals.

In summary, while WeRide offers a broad and flexible carpooling experience, the UTAR Carpooling Web Application is a more targeted solution, providing students with specialized features that enhance convenience, efficiency, and user satisfaction within an academic setting.

2.2.2 Grab Advance Booking

Grab [15] is one of Southeast Asia's leading ride-hailing platforms, offering a wide range of services from on-demand transport to food delivery and digital payments. Among its many transport-related features, the Advance Booking option stands out as a practical solution for users who need to plan their journeys ahead of time [16]. This feature allows passengers to schedule a ride for at least one hour to up to seven days in advance, ensuring peace of mind for time-sensitive trips such as airport transfers, early morning commutes, or urgent meetings.

Figure 2.4 Grab app logo

The process is simple and intuitive. Users select their pick-up and drop-off locations, set the desired date and time, and confirm the booking through the app. A few hours before the scheduled time, Grab assigns the driver to the trip and notifies the passenger once the ride has been successfully matched. This ensures a higher degree of certainty in securing transport during peak hours or off-peak periods where driver availability may be limited, such as late at night or early in the morning.

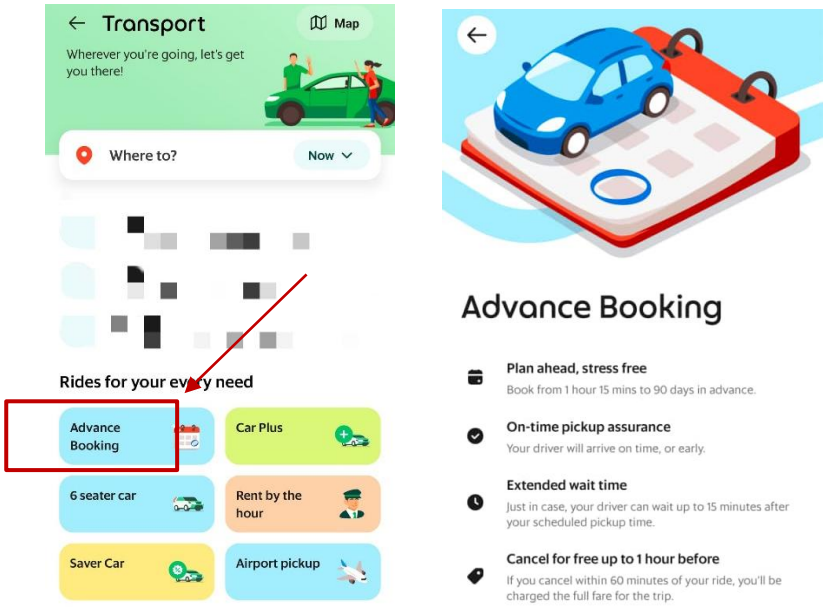


Figure 2.5 Grab advance booking feature

Advance Booking rides are subject to a priority allocation fee, which is added to the fare to compensate for the driver's commitment to accept the scheduled job. The service is currently available in selected cities and for specific ride types such as GrabCar, with availability depending on local driver supply and demand.

This feature particularly appeals to users who value predictability and time management, as it minimizes the uncertainty often associated with real-time ride-hailing. By offering a planned

alternative to last-minute bookings, Grab caters to a segment of the market that requires a more structured travel experience.

One of the key strengths of Grab's Advance Booking feature is its **convenience and flexibility** [17]. By allowing users to schedule rides ahead of time, it helps them avoid the stress of last-minute bookings—especially during **high-demand periods or odd hours** like midnight or early mornings, when driver availability may be low. This is particularly useful for passengers with fixed schedules, such as airport drop-offs or early lectures.

Another strength lies in its **automation and reliability**. The platform automatically matches riders with available drivers ahead of time, reducing the need for manual searching or constant communication. Grab's large network of drivers also increases the chances of a successful match, even in more remote areas.

In terms of user experience, the interface is **streamlined and easy to use**, offering quick scheduling within the familiar Grab app. Integration with Grab's **real-time GPS tracking**, in-app communication, and **fare estimation** adds value to the overall experience.

Despite its advantages, Grab's Advance Booking feature has several limitations. Firstly, it is **only available in selected cities and for specific ride types**, which limits accessibility for some users. In university towns or less populated areas, this feature may not be fully supported.

Secondly, **cost is a major drawback**. Grab applies an additional **priority allocation fee**, making scheduled rides more expensive than regular ones. For budget-conscious users such as university students, this can be a significant deterrent.

Another limitation is that **riders are not guaranteed a match**, even if they book in advance. The system only begins to assign a driver shortly before the scheduled ride, and there is a chance that no driver is found on time, especially during low-demand hours or in less-served areas.

Additionally, **Grab is a commercial service** with a focus on broader public transportation needs rather than community-driven or student-specific carpooling. It lacks the sense of **community, affordability, and student-based customization** that some users, especially students, might prefer.

Compared to Grab's Advance Booking feature, your proposed platform offers a **student-centric, community-based carpooling system** tailored specifically to the **daily commuting**

needs of university students. While Grab is designed for a broad user base with commercial pricing and operates on an on-demand model, our platform focuses on **pre-scheduled, affordable, and consistent ride sharing** among peers.

One major distinction is the inclusion of **recurring rides** on your platform. This feature allows users—both drivers and riders—to set up **routine trips** (e.g., daily rides to and from campus), eliminating the need to repeatedly search or post for new rides each day. This is especially useful in a university context, where schedules tend to be consistent week-to-week. In contrast, Grab does not support recurring advance bookings, requiring users to manually schedule each ride individually, which can be **inconvenient for regular commuters**.

Additionally, our platform supports **user-determined fare pricing**, which is typically lower and more flexible than Grab's commercial rates that include surge pricing and priority allocation fees. It also fosters a sense of **community and trust**, as students are more likely to interact and coordinate with fellow university members rather than with unknown drivers.

Our proposed solution also provides better **ride management tools**, helping drivers handle multiple ride requests more efficiently through a dedicated platform, something that is challenging when done manually via social media or through Grab's one-on-one system.

Overall, while Grab offers a professional, city-wide ride-hailing service, our proposed system delivers a **more relevant, cost-effective, and personalized** experience for students who need reliable, recurring transportation to and from campus.

2.2.3 BlaBlaCar

Figure 2.6 BlaBlaCar Logo

BlaBlaCar is one of the world's largest long-distance carpooling platforms, connecting drivers with empty seats to passengers looking to travel the same route [18]. Founded in France in 2006, the platform focuses on matching drivers and riders for intercity travel rather than short

urban commutes. BlaBlaCar operates by allowing drivers to post their planned trips along with departure time, destination, and available seats. Interested passengers can then search, view trip details, and book a ride directly through the app or website. The system also emphasizes community trust, offering verified profiles, ratings, and identity verification features to ensure a safe and reliable experience [19]. By facilitating cost-sharing between drivers and passengers, BlaBlaCar promotes affordable and environmentally sustainable travel options.

One of BlaBlaCar's major strengths is its focus on building trust within the community. The platform incorporates strong user verification processes, including phone number verification, government ID checks, and mandatory profile pictures, creating a sense of security for both drivers and riders [18]. In addition, the rating and review system allows users to share feedback about their experiences, which further encourages positive behavior. Another strength is the simplicity and clarity of trip postings: drivers provide detailed trip information upfront, making it easier for riders to find and book suitable options. The platform also handles payment transactions, ensuring transparency and reducing the chances of disputes. Furthermore, BlaBlaCar's emphasis on cost-sharing instead of profit-seeking aligns with legal requirements in many regions, helping it maintain operations without needing to comply with regulations meant for commercial taxi services.

Despite its strengths, BlaBlaCar does face some weaknesses. One key limitation is that it primarily focuses on long-distance, scheduled trips rather than real-time or last-minute rides [19]. This restricts its flexibility compared to on-demand ride-hailing services like Uber or Grab. Another weakness is that ride availability is highly dependent on user activity; in less populated areas or during off-peak times, finding a suitable ride can be challenging. Moreover, while BlaBlaCar has implemented various trust mechanisms, it still relies heavily on the honesty and behavior of its users, meaning that occasional safety or reliability issues may occur. Finally, since payments and bookings are handled online, users without access to digital payment methods may find the platform less convenient to use.

When comparing BlaBlaCar with the proposed community-based car-sharing platform, several similarities and differences emerge. Like BlaBlaCar, the proposed system focuses on connecting drivers and riders through pre-planned trip listings. Both platforms emphasize the importance of providing clear trip details upfront to allow users to make informed booking decisions.

However, there are key differences. While BlaBlaCar targets primarily long-distance intercity travel, the proposed platform is designed for **shorter, more local trips** — specifically focusing on university students commuting between their homes and the campus. Moreover, the proposed solution introduces **real-time dynamic searching, ride matching through AI chatbot**, and **integration with Google Maps APIs** to optimize routes, offering a more tech-driven and flexible system compared to BlaBlaCar's relatively manual matching process. Additionally, while BlaBlaCar is already optimized for mobile and desktop use, the current proposed solution is initially built focusing on desktop access first, with mobile responsiveness as a future enhancement.

Overall, the proposed system adopts some of the best practices from BlaBlaCar while tailoring its approach to suit the needs of a localized, student-focused audience with more AI integration and route optimization features.

2.2.4 Summary of The Existing Systems

Table 2.2: Comparison of previous work and proposed solution

Solutions/ Key Features and Differences	**WeRide**	**Grab Advance Booking**	**BlaBlaCar**	**Proposed System**
Target Audience	Malaysian and Singaporean	General Public	General Public	UTAR Kampar Students
Geographical Coverage	Malaysia and Singapore	Nationwide	Nationwide	Kampar
Platform Type	Mobile app	Mobile app	Web and Mobile app	Web application
Scheduling Type	Single and recurring rides	Single ride	Single ride	Single and recurring rides
AI Integration	-	Smart Matching	-	Chatbot
Ride Matching Method	Manual search	Auto-matching	Manual search	Manual search and chatbot
Ride Management Tools	Available	-	Available	Available
Communication Method	WhatsApp	In-app	Phone Number	WhatsApp

Verification/ Security	Linking Facebook and WhatsApp	Required verified driver	Government ID check and phone number verification	-
Cost Flexibility	Negotiable	Set by system	Set by driver only in acceptable range	Set by driver
Extra Features	Traffic cams	Estimated fare	Payment transactions	Academic timetable schedule

Chapter 3

System Methodology

This chapter outlines the methodological approach adopted for developing the Community-Based Car-Pooling Platform. It presents the selected software development methodology and its justification, along with project planning activities, tools, and technologies used throughout the lifecycle. The methodology includes the design of the system architecture, integration of components such as the AI chatbot, OCR module, and Google Maps services, as well as strategies for data management, API integration, and version control. By following a structured and iterative process, the development ensures the platform meets its functional requirements while delivering a reliable, efficient, and user-friendly experience tailored for university students seeking affordable and convenient car-pooling solutions.

3.1 Development Methodology

The development of the car-pooling system followed the Agile methodology, chosen for its adaptability, iterative progress, and focus on continuous improvement based on feedback. The project was divided into multiple sprints, each lasting approximately two weeks, with specific deliverables defined at the start of each sprint. Initial sprints concentrated on building the system's core backend using PHP Laravel, establishing the database schema in MySQL, and setting up essential user authentication functions. Subsequent sprints focused on frontend development using Laravel Blade with BladewindUI and Tailwind CSS to create a responsive and user-friendly interface. Integration tasks were carried out iteratively, such as connecting the AI chatbot developed in Dialogflow, embedding Google Maps API for location and route handling, and implementing OCR capabilities for timetable extraction. Each sprint concluded with testing and review sessions, allowing for adjustments to features, workflows, and user interface elements based on findings. Version control was maintained through GitHub to ensure smooth collaboration and tracking of changes, while Ngrok was used for temporary hosting to facilitate online testing of API endpoints and chatbot interactions. This iterative approach ensured that functional components of the system were delivered incrementally, with regular opportunities for refinement, ultimately leading to a robust and scalable car-pooling platform.

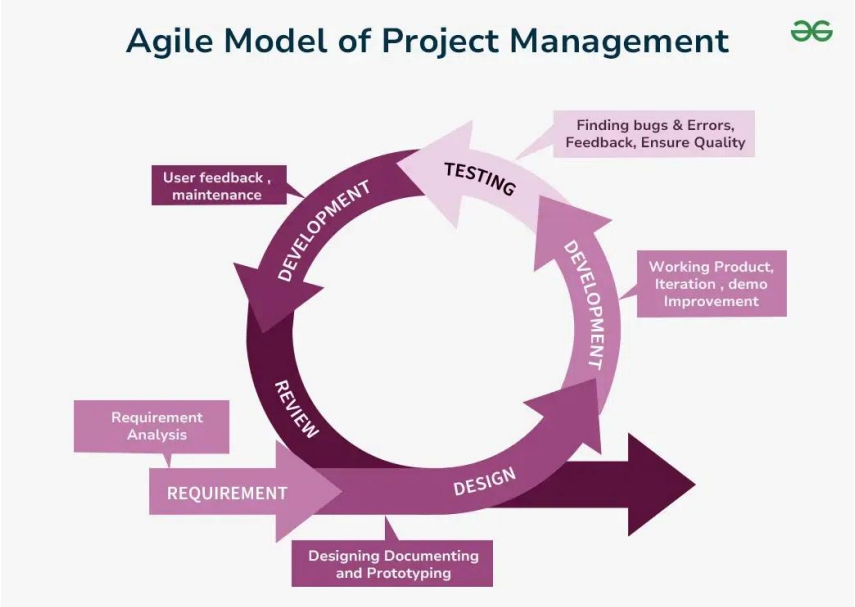


Figure 3.1 Agile Development Cycle

3.2 System Architecture

The system architecture for the proposed car-pooling platform follows a three-tier structure consisting of the presentation layer (web browser), application layer (server), and data layer (database).

The **presentation layer** is developed using Laravel Blade templates integrated with BladewindUI and Tailwind CSS, providing a responsive and user-friendly interface for both drivers and riders.

The application layer is built with PHP Laravel, which manages the core business logic, including ride creation, booking, trip matching, and communication features. This layer also integrates with external services such as Dialogflow for the AI chatbot and Google Maps API for location-based functionalities. In addition, it connects to a **custom-developed PaddleOCR API**, specifically designed to extract UTAR student timetable classes. This API is implemented in Python, where the PaddleOCR model is preloaded to perform text detection on uploaded timetables. The detected text is then processed to identify and categorize class information. The service is hosted using FastAPI, enabling Laravel to send timetable images and receive structured class data for timetable-based ride matching.

The **data layer** is managed by MySQL, which stores user profiles, ride listings, booking records, chat interactions, and system logs. The architecture ensures modularity and scalability, allowing each component to be updated or replaced without impacting other parts of the system. The design also incorporates secure API communication, version control through GitHub, and temporary online hosting via Ngrok for testing purposes during development.

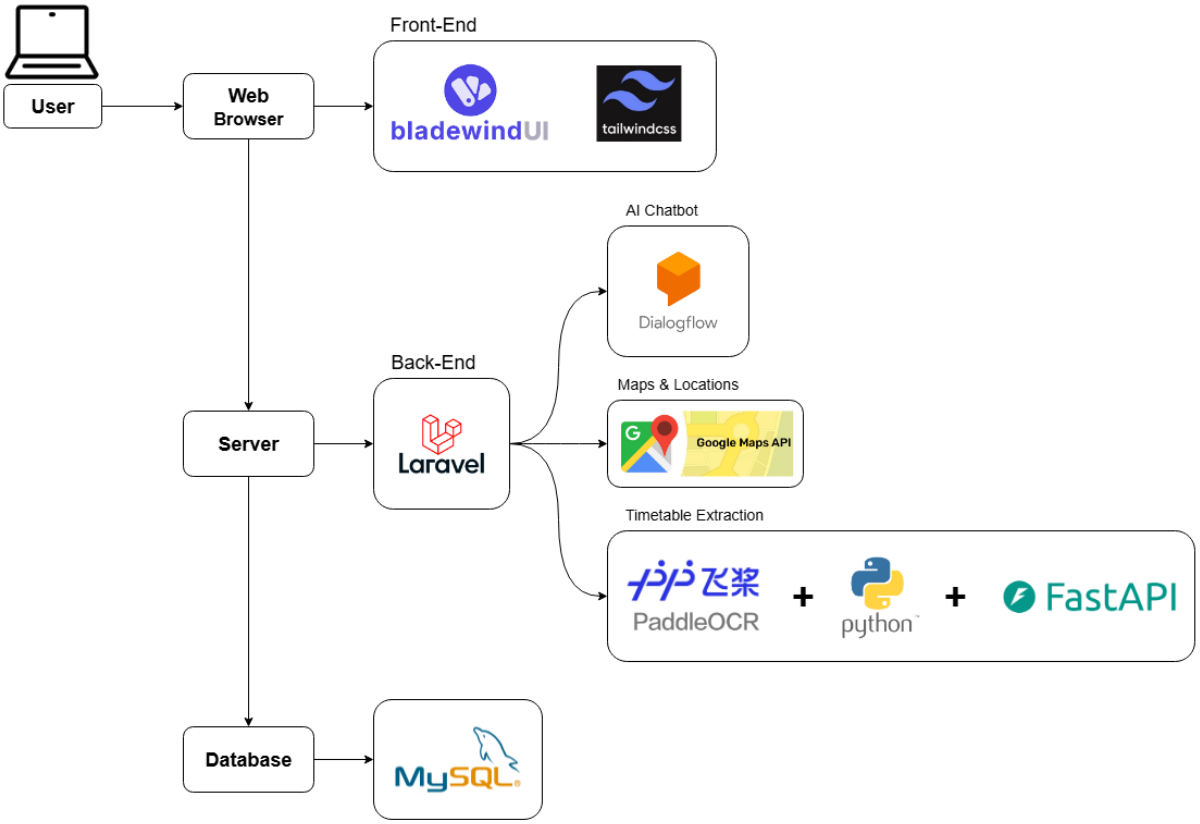


Figure 3.2 System Architecture Diagram

3.3 Software Design Pattern

This system follows the MVC (Model-View-Controller) architectural pattern provided by Laravel. MVC separates the application into three interconnected components, enabling modular development and easier maintenance.

Model: Represents data and business logic. For example, models such as User, Ride, and Booking interact with the MySQL database using Laravel's Eloquent ORM.

View: The UI of the application created using Blade templates and BladewindUI components. It displays data received from the controller.

Controller: Acts as the intermediary between model and view. It processes user requests, retrieves data using models, and returns the appropriate views.

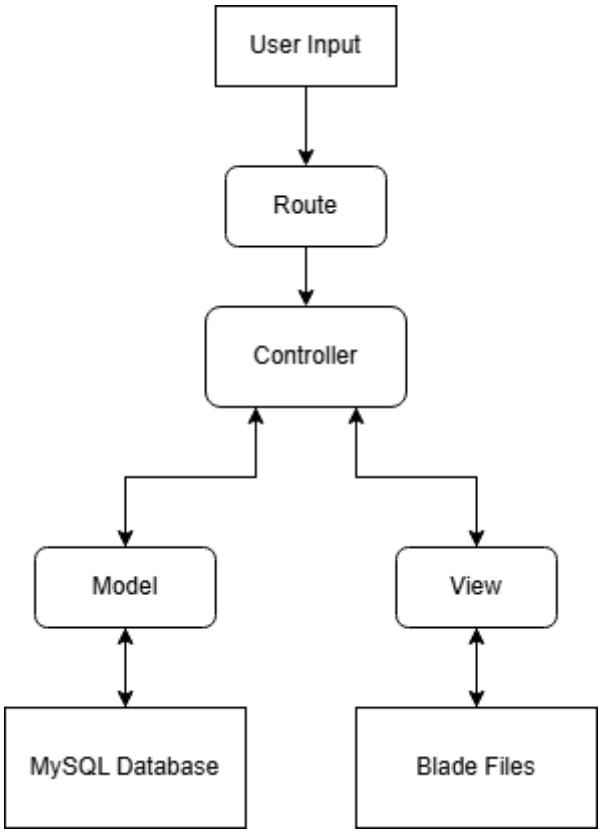


Figure 3.3 MVC Diagram

3.4 Use Case Diagram

The **Carpooling Web Application** is designed to allow users (both drivers and riders) to register, log in, search for available rides, create ride listings, book rides, and manage ride bookings. The following are the key use cases derived from the diagram:

Actors

- **User (Driver/Rider)**: A single actor representing both roles, as a user can act as a driver offering rides or as a rider searching for rides.

Use Cases

1. **Register Account**
    - Description: Allows a new user to create an account in the system by providing the required personal details.
    - Actor: User (Driver/Rider).

- Precondition: User must not already have an account.
- Outcome: Account is created and stored in the system database.

2. **Login Account**
   - Description: Allows a registered user to log into the system by providing valid credentials.
   - Actor: User (Driver/Rider).
   - Precondition: User must have an existing account.
   - Outcome: User gains access to their dashboard and available system features.
3. **Search Rides**
   - Description: Enables users to find available rides based on certain criteria.
   - Actor: User (Driver/Rider).
   - Variations:
     - **Address, Ride Type, Date, and Time-Based Search** *(include)*: Users can search using specific filters such as origin, destination, ride type, and date/time.
     - **AI Chatbot Search** *(extend)*: Users can interact with an integrated AI chatbot to search for rides in a conversational manner.
     - **Timetable-Based Search** *(extend)*: Users can view rides matched against submitted timetable.
4. **Create Ride**
   - Description: Allows a driver to list a new ride offer in the system.
   - Actor: User (Driver).
   - Variations:
     - **Ride Request / Offer** *(include)*: The ride creation process includes specifying whether the listing is a ride offer or a ride request.
     - **Recurring Rides** *(extend)*: Allows creation of rides that repeat on specific days or intervals.
     - **Timetable-Based Creation** *(extend)*: Drivers can create rides automatically by submitting a timetable.
5. **Book Ride**

- Description: Allows a rider to confirm and reserve a seat for a ride listed in the system.
- Actor: User (Rider).
- Precondition: Ride must be available with sufficient seats.
- Outcome: Booking is recorded in the system.

6. **Manage Rides / Booking**
   - Description: Enables users to update, cancel, or review rides they have created or booked.
   - Actor: User (Driver/Rider).
   - Outcome: Changes are saved to the database and reflected in ride availability.

Figure 3.4 Use Case Diagram

3.5 Activity Diagram

Ride Searching – Input Fields

The user visits the platform and enters search criteria (like departure, destination, date). The system queries the ride listings and shows matches. No login is required for this.

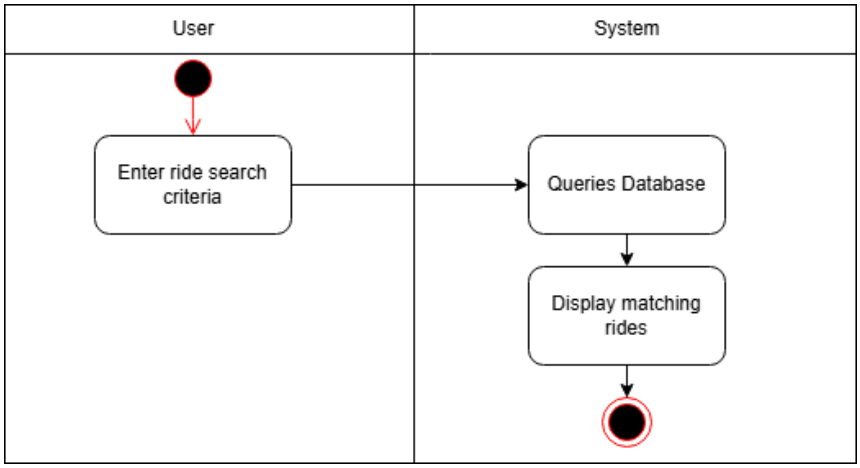


Figure 3.5 Ride searching activity diagram

Ride Searching – AI Chatbot

The chatbot lets users enter natural language queries. Dialogflow extracts the required parameters and sends them to the system backend. Matching rides are shown in a link.

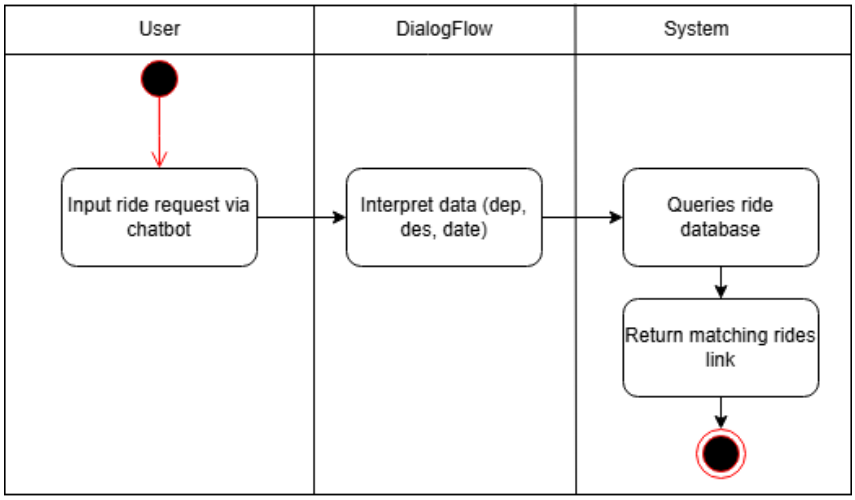


Figure 3.6 Chatbot ride recommendation activity diagram

Ride Searching – Timetable-based

This activity diagram represents the process of searching for rides using a UTAR timetable with the help of PaddleOCR. The process begins when the user uploads their UTAR timetable to the system. The system first validates the uploaded image for correct format and size, ensuring it meets the processing requirements. Once validated, the image is sent to PaddleOCR,

which extracts textual information from the timetable. The system then filters and classifies the detected text to identify relevant details such as class times, locations, and days. These details are grouped into class schedules, from which the system determines potential rides that match the user's timetable. The identified possible rides are stored, and finally, the system displays a filtered list of rides to the user based on the stored results, allowing them to select or explore available car-pooling options.

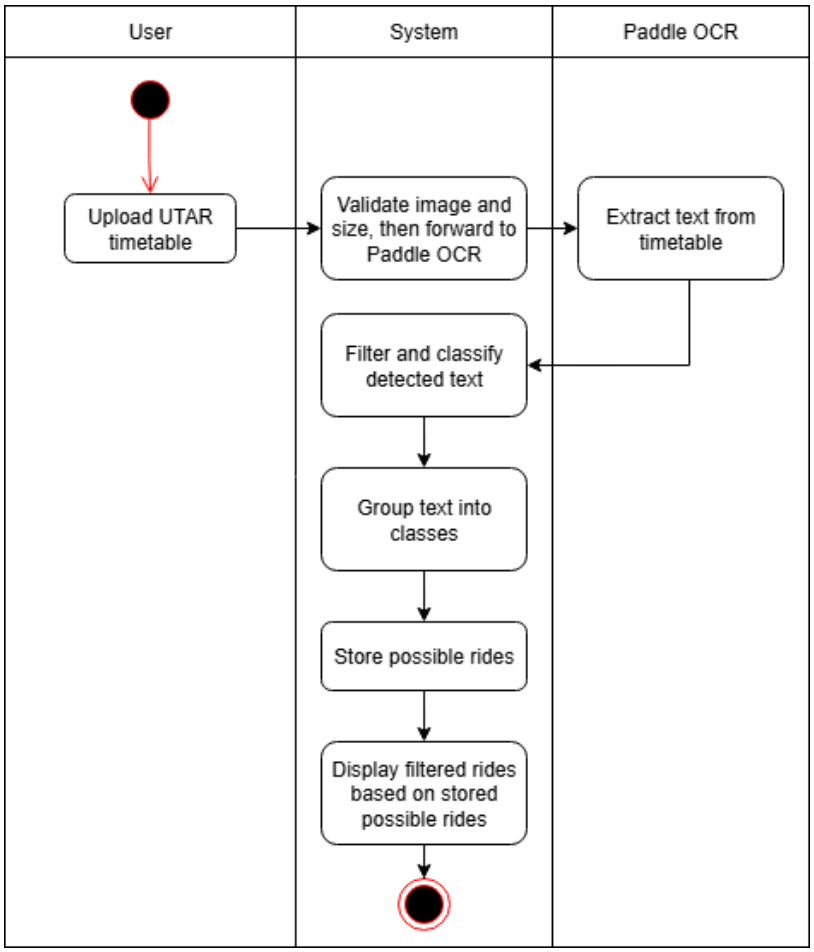


Figure 3.7 Timetable-based ride searching activity diagram

Ride Posting – Basic Input Form

Only logged-in users can post rides. After login, the user fills out a form. The system validates the input (e.g., via Google Maps API) and stores it. The user is redirected to their ride listings.

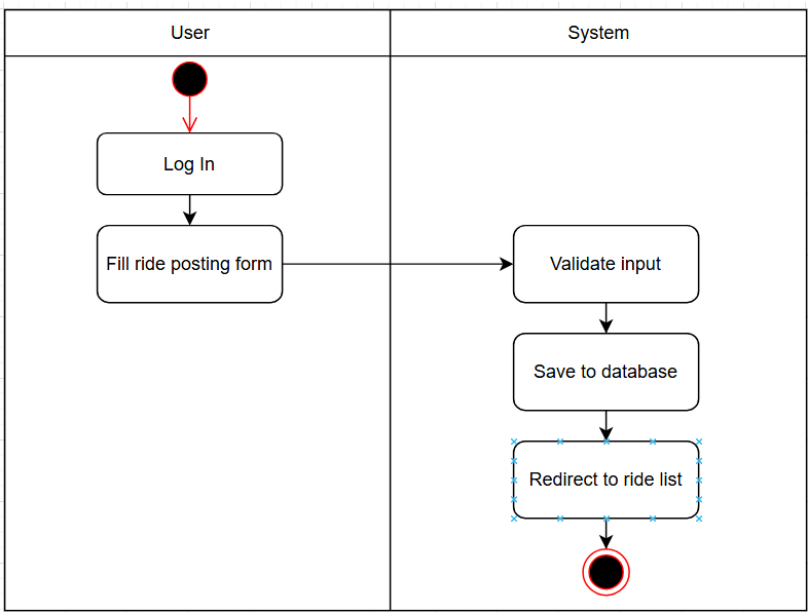


Figure 3.8 Ride posting activity diagram

Ride Posting– Timetable-based

The activity diagram illustrates the process of creating recurring rides from a UTAR timetable using the PaddleOCR text recognition system. The process begins when the user uploads an image of their UTAR timetable to the system. The system first validates the image format and size to ensure it meets the required specifications. Once validated, the timetable is forwarded to PaddleOCR, which extracts all the text from the image. The extracted text is then filtered to remove irrelevant details and classified into meaningful categories, such as class names, locations, and times. These classified elements are grouped into individual classes, forming the basis for ride creation. The system then displays the grouped ride details to the user for review, allowing them to provide any remaining input such as date and time, meeting point, or seat availability. Upon submission, the system automatically generates recurring ride entries based on the provided and extracted information, streamlining the car-pooling arrangement process.

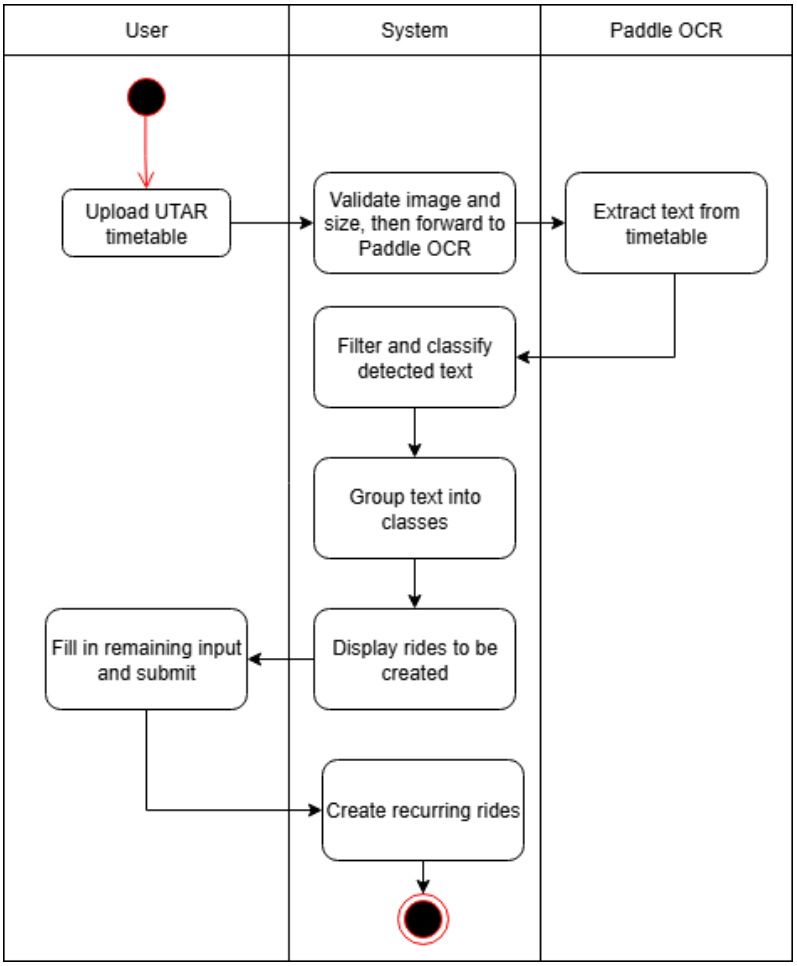


Figure 3.9 Timetable-based ride creation activity diagram

Ride Booking

Once logged in, the user can view ride details and book. If the user is the ride's creator or the ride is already accepted, they cannot book. Otherwise, the booking is marked as pending.

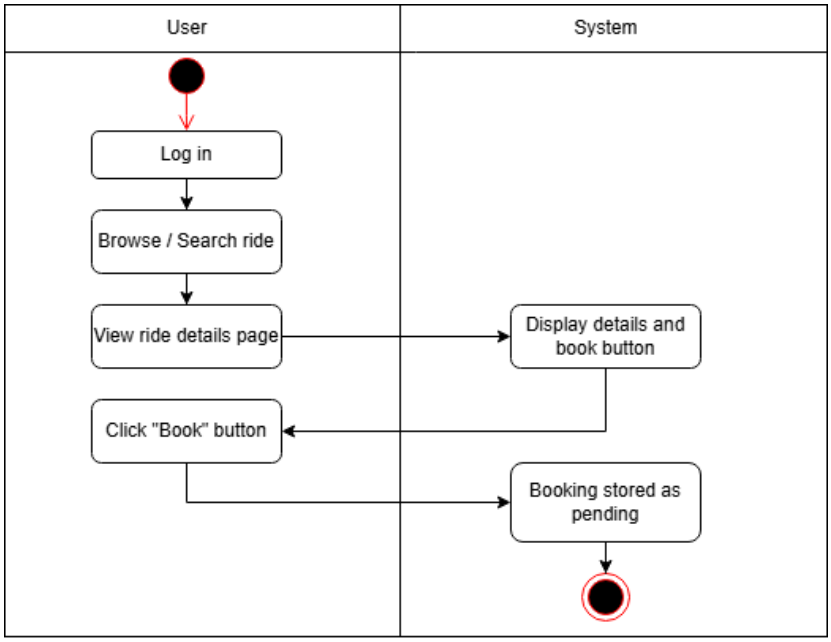


Figure 3.10 Ride booking activity diagram

3.6 System Flowchart

The system flowchart illustrates how users interact with the system from the landing page onward. It includes decision points such as login status and accessibility of user actions (search, book, post ride).

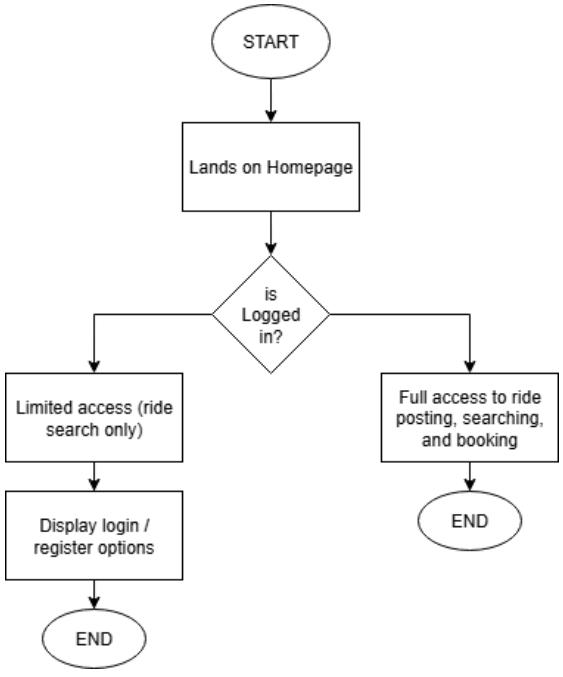


Figure 3.11 System Flowchart

Chapter 4
System Design

4.1 System Block Diagram

The system block diagram illustrates the main components of the carpooling web application and their interactions. At the top level, the User Interface is implemented using Laravel Blade templates styled with Tailwind CSS and BladewindUI, allowing users to interact with the system through a browser. User requests are processed by the Application Logic layer, built with PHP Laravel following the MVC architecture. This layer contains the controllers and service classes responsible for handling inputs, coordinating processes, and managing data flow.

The application communicates with two main forms of storage: a MySQL Database and Session Storage. The database stores persistent records such as user profiles, ride listings, and booking details. In contrast, session storage is used to temporarily hold conversational data from the Dialogflow chatbot, ensuring that chat context is maintained during a user session but not stored permanently. Moreover, session storage also used to temporarily hold the PaddleOCR API results for the timetable-based ride searching operation.

The Application Logic layer also integrates with several External APIs. The Google Maps API is used for location-based functionalities such as retrieving place details and generating routes. The Dialogflow API powers the AI chatbot, which provides conversational assistance to users and stores session context temporarily without committing data to the main database. The PaddleOCR API handles text recognition tasks, with its results passed back to the application in real time without database storage.

This structure ensures a clear separation of concerns: the user interface manages presentation, the application logic orchestrates processes, the database handles persistent storage, and external APIs deliver specialized services.

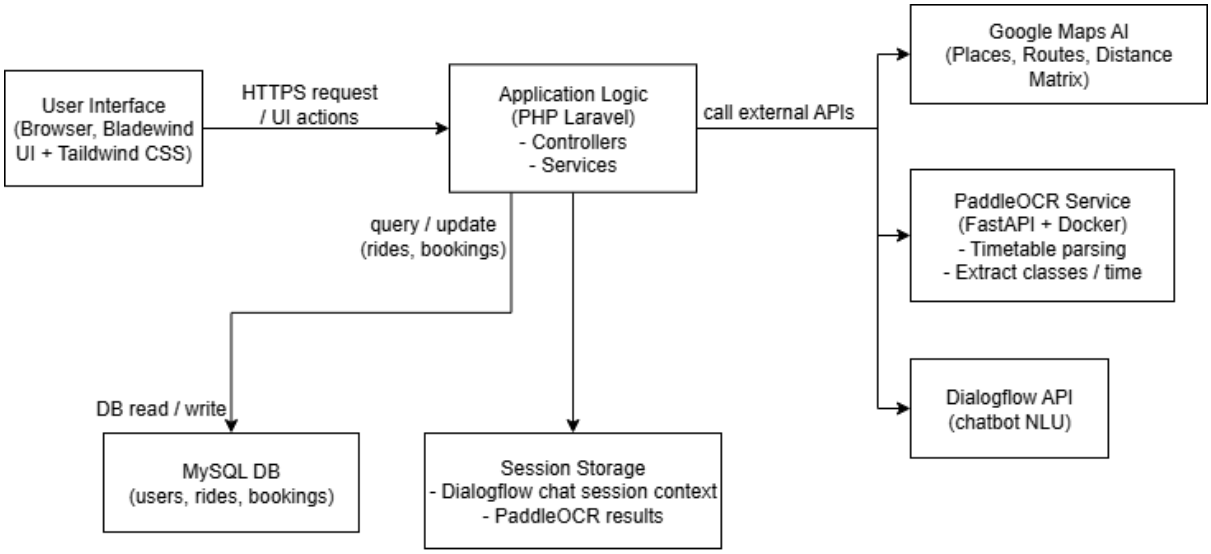


Figure 4.1 System Block Diagram

4.2 Deployment Diagram

The deployment diagram illustrates how the **carpooling web application** is hosted and how its components interact in the **production environment**.

1. **User Access (Browser)**
    - End-users interact with the system via a web browser.
    - The browser sends requests through the internet to the **Laravel Cloud hosting service**, which runs the production version of the carpooling web application.
2. **Laravel Cloud (Web Application Hosting)**
    - The Laravel-based application is deployed on **Laravel Cloud**, which handles incoming HTTP/HTTPS requests.
    - The web application contains all the logic for ride creation, booking, chatbot integration, and user management.
    - This environment ensures scalability and reliability for multiple concurrent users.
3. **Database Server (Azure Database for MySQL)**
    - The application connects to **Azure Database for MySQL**, which stores persistent data such as user profiles, ride offers/requests, bookings, and chatbot logs.

- This ensures data durability and centralized access for both the web application and APIs.

4. **OCR API (Azure Container Instance)**
    - The timetable-based ride creation feature relies on an OCR service deployed as a **Dockerized FastAPI application** inside an **Azure Container Instance**.
    - Users upload their timetable screenshots through the Laravel web application.
    - The request is forwarded to the OCR API, which processes the image, extracts timetable data, and returns structured results to the Laravel system.
5. **Communication Flow**
    - The browser communicates only with Laravel Cloud.
    - Laravel Cloud acts as the central hub, communicating with both the **MySQL database** and the **OCR API container**.
    - The OCR API communicates back with Laravel Cloud via REST API responses.
6. **Security & Deployment Considerations**
    - HTTPS is enforced between the browser and Laravel Cloud.
    - The database and container instance are secured with private credentials and access control, ensuring only the application can connect.
    - This architecture separates web application logic from specialized OCR processing, promoting modularity and scalability.

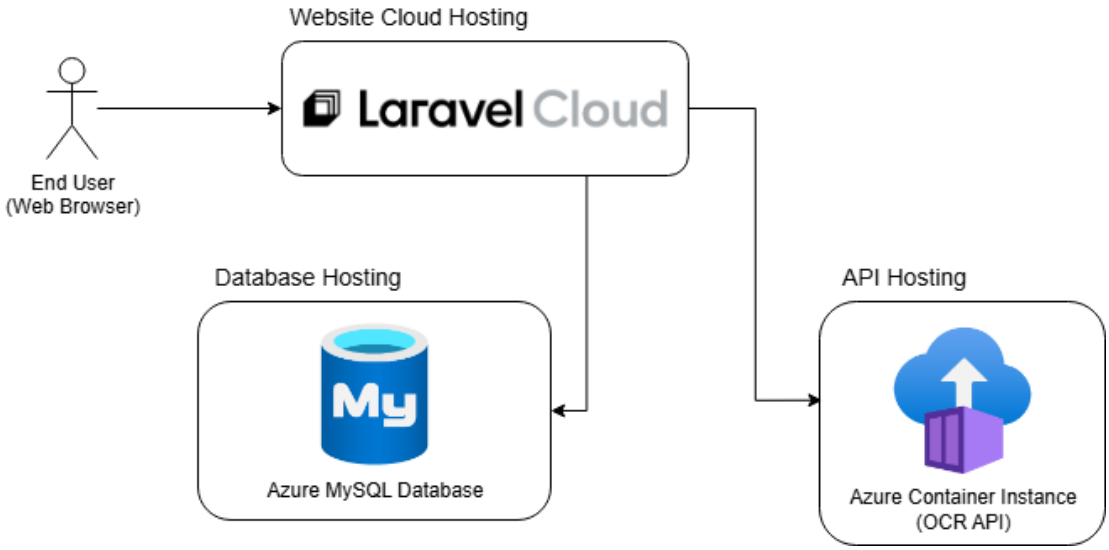


Figure 4.2 Deployment Diagram

4.3 System Components Specifications

4.3.1 Frontend

The frontend is designed to provide a seamless and interactive experience for users while maintaining consistency across the application's interface. Laravel Blade handles most of the rendering, while JavaScript is used for dynamic elements and real-time interactions.

Key Components:

- **Laravel Blade** – Server-side template engine for rendering views and layouts.
- **Tailwind CSS** – Utility-first CSS framework for responsive and consistent styling.
- **BladewindUI** – Pre-built Laravel-compatible UI components to accelerate development.
- **JavaScript** (AJAX & Chatbot)
    - Handles AJAX calls for actions such as form submissions without full page reloads.
    - Powers chatbot interaction:
        - Calls AI chatbot API.
        - Updates messages in real-time.
        - Stores chat history in session storage for persistence during the session.

4.3.2 Backend

- **Framework & Architecture:** Developed using **PHP Laravel** following the **Model-View-Controller (MVC)** pattern. Laravel's built-in routing, controllers, and middleware (including authentication) are used to manage request handling, enforce access control, and streamline the application workflow.
- **Business Logic:** Handles most of the core operations triggered by user actions. This includes **user authentication**, **form validation**, **ride creation**, **ride display**, **ride filtering**, and **ride updates**. All data-related operations follow Laravel's validation rules to ensure data integrity.
- **API Integration:** All third-party API calls (e.g., Google Maps API, PaddleOCR API) are initiated and processed in the backend, except for **Dialogflow API**, which is called directly from the frontend JavaScript for real-time chatbot interaction.
- **Security Measures:** User passwords are encrypted before storage in the database to maintain confidentiality. No other encryption is implemented for general data exchange.
- **File Handling:** Supports file uploads where applicable, processing them securely through Laravel's file handling mechanisms.

4.3.3 Database

Database Technology:

- **MySQL 8.0.30** is used as the relational database management system, offering reliable data storage, transactional integrity, and compatibility with Laravel's Eloquent ORM.

Key Tables:

- **users** – Stores user credentials and profile details.
- **rides** – Contains core ride information, linked to the ride creator and optionally to recurring or timetable rides.
- **recurring_rides** – Stores repeating ride patterns for automated ride creation.
- **timetable_rides** – Stores predefined ride schedules for easy reference.

- **offers** – Records offers made for specific rides.
- **bookings** – Tracks ride bookings, including sender and receiver details.

Table Relationships:

- **rides.user_id** → users.id (Ride creator)
- **rides.recurring_id** → recurring_rides.id (Optional recurring ride link)
- **rides.timetable_id** → timetable_rides.id (Optional timetable link)
- **offers.ride_id** → rides.id (Offer belongs to a ride)
- **bookings.ride_id** → rides.id (Booking for a specific ride)
- **bookings.sender_id / receiver_id** → users.id (Users involved in the booking)

Indexes & Constraints:

- Primary keys on all tables.
- Foreign key constraints enforced through Laravel migrations.
- Relational integrity maintained by Eloquent ORM.

Hosting & Backup:

- **Development:** Local MySQL instance.
- **Deployment:** Hosted on **Azure Database for MySQL Flexible Server**.
- Automated daily backups handled by Azure to ensure data reliability.

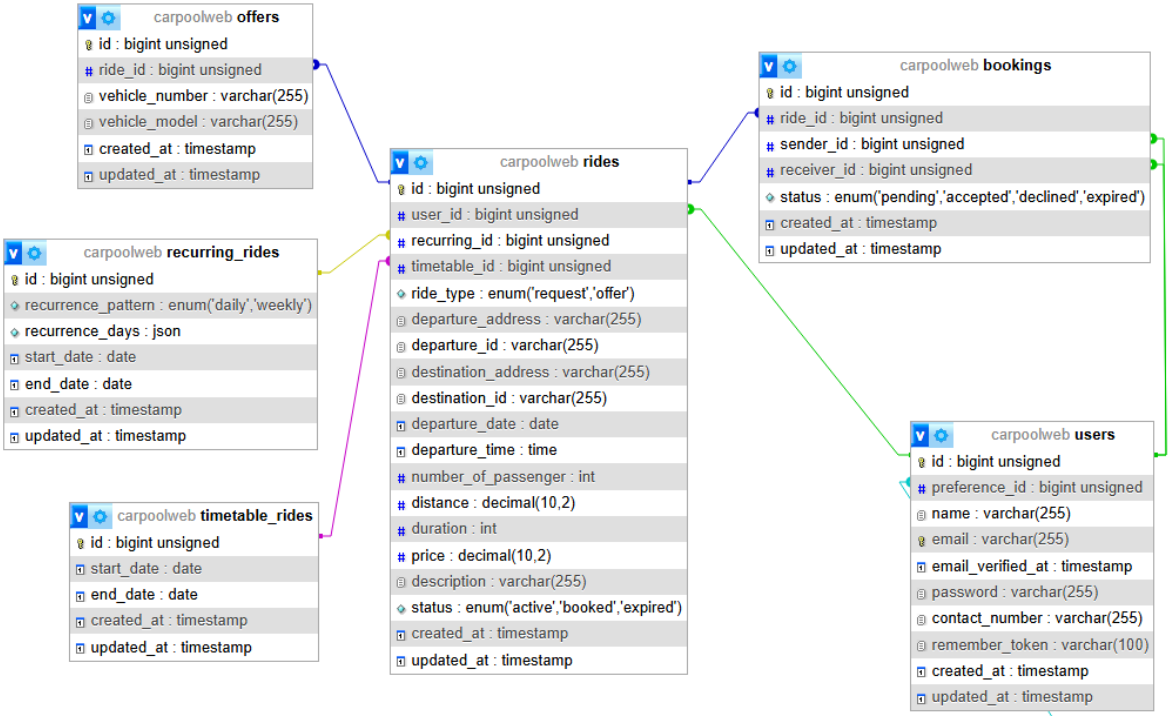


Figure 4.3 ERD diagram

4.3.4 External APIs

Google Maps API

- Purpose: Provides location-based services to enhance the carpooling experience.
- Functions Used:
    - Map Display – Embeds interactive maps in the application interface.
    - Route Display – Shows driving routes between departure and destination points.
    - Place ID Retrieval – Obtains unique identifiers for selected locations.
    - Distance & Duration Calculation – Retrieves travel distance and estimated time between two addresses.
    - Address Autocomplete – Suggests addresses dynamically as users' type.
- Integration: API keys are stored securely in Laravel's .env file, and calls are made from both frontend (JavaScript) and backend (PHP) where appropriate.
- Data Handling: Responses are processed in real-time without being stored permanently.

Dialogflow API

- Purpose: Acts as the natural language processing (NLP) engine for the chatbot feature.
- Functions Used:
    - Extracts departure location, destination location, and travel date from user messages.
    - Passes extracted data to Laravel backend for ride-matching operations.
- Integration: Invoked directly from the frontend JavaScript code.
- Data Handling: Responses are processed immediately and stored only in browser session storage for chat history.

PaddleOCR API

- Purpose: Automates extraction of timetable data from uploaded images to support timetable-based ride searching.
- Development Environment: Self-hosted locally via Docker container.
- Deployment Environment: Hosted on Azure Container Instances running FastAPI with a preloaded PaddleOCR model for faster inference.
- Data Handling: API responses are temporarily stored in session only for timetable ride searching.

Ngrok

- Purpose: Used exclusively during development to expose the local environment to the internet for API testing and integration.
- Scope: Temporary and not part of production deployment.

4.4 System Components Interaction Operations

4.4.1 Ride Posting – Basic Form Input

This process enables users to post either one-time or recurring rides through a structured form. The frontend leverages reusable Laravel Blade components for location inputs, enriched by Google Maps services for address autocompletion, distance calculation, and estimated travel

time. Data submission is handled asynchronously via AJAX, ensuring a smooth user experience.

Interaction Flow:

1. **User Input**
    - **One-Time Ride**: User enters departure and destination addresses, date, time, number of passengers, price, description, and ride type (request/offer). If the ride is an offer, vehicle number and vehicle model are also required.
    - **Recurring Ride**: User enters departure and destination addresses, time, number of passengers, price, recurrence pattern (daily/weekly), recurrence days (if weekly), start and end dates, description, and ride type (request/offer).
2. **Frontend Processing**
    - Address fields use a custom Blade component integrated with Google Maps Places Autocomplete.
    - Once both addresses are selected, Google Maps Distance Matrix Service calculates travel distance and duration.
    - Distance and duration are stored in hidden form fields for backend use.
    - Simple HTML-based validation is applied before submission.
3. **Form Submission**
    - Data is sent via AJAX request to the appropriate Laravel route, mapped to the corresponding controller method.
4. **Backend Processing**
    - Laravel validates all form data.
    - Determines ride type (one-time or recurring) and ride category (request or offer).
    - Creates records in the appropriate database tables (rides, offers, recurring_rides).
5. **Post-Save Response**

- On success: Backend responds with a success message, which is displayed using SweetAlert2, followed by a redirect to the ride list page.
- On validation errors: Returns error messages for each invalid field, which are displayed in line with red borders and messages.
- On system errors: SweetAlert2 displays a relevant error notification.

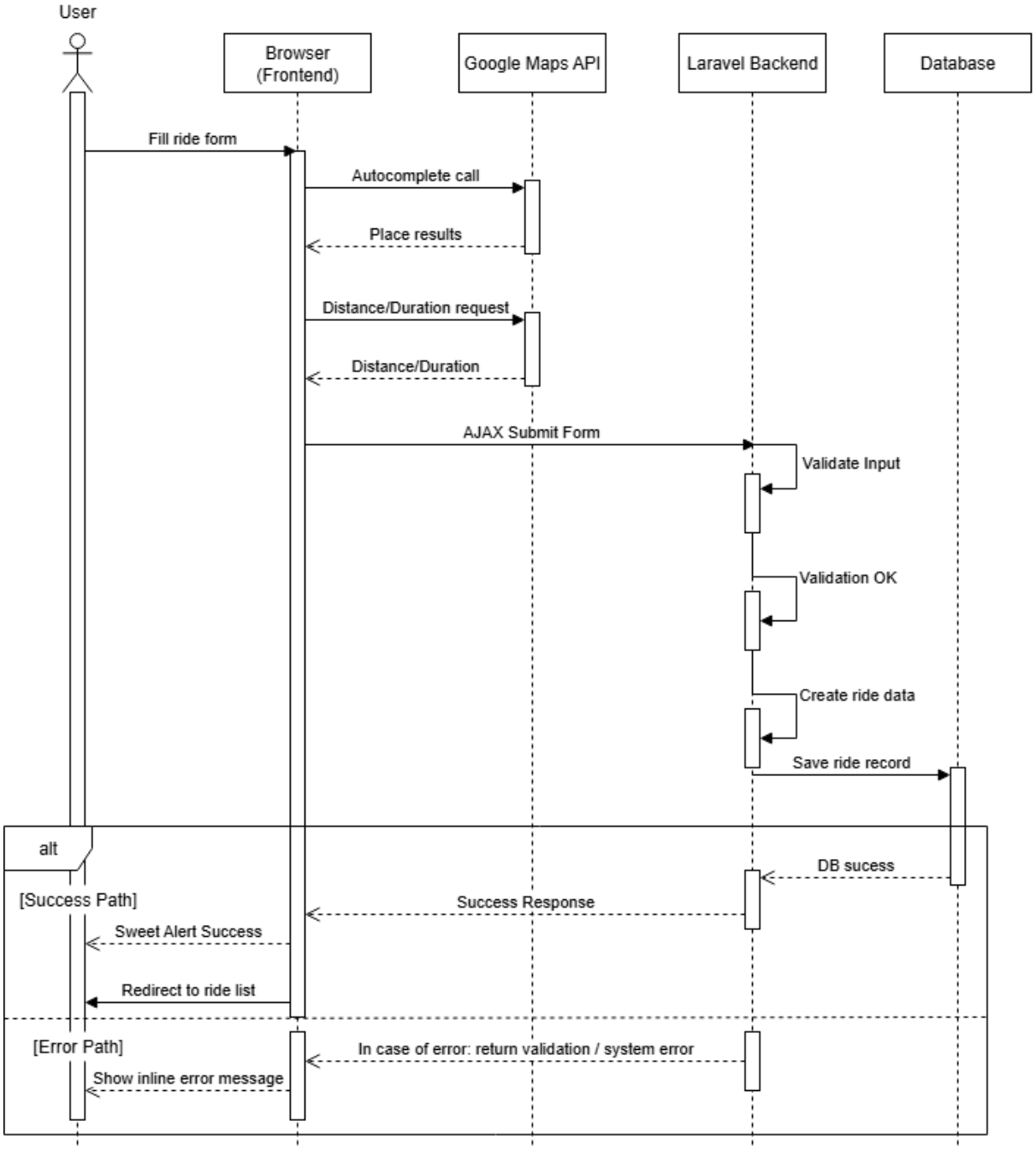


Figure 4.4 Ride posting – basic form input Sequence Diagram

4.4.2 Paddle OCR API

This self-developed PaddleOCR API is designed to automatically extract structured class schedule data from a submitted timetable image. Its purpose is to simplify the ride creation and searching process in the carpooling platform by eliminating the need for manual entry.

Interaction Flow:

1. **User Upload**
    - User submits an image of their timetable through the Laravel application.
    - Laravel validates and prepares the image file.
2. **Laravel → PaddleOCR API (Request)**
    - Laravel sends an **HTTP POST request** with the timetable image to the PaddleOCR API endpoint.
3. **PaddleOCR API Processing**
    - The API validates the uploaded file (e.g., size/format).
    - Crops the timetable region using **OpenCV** (cv2).
    - Uses the **pre-loaded PaddleOCR model** (loaded once when container started) to recognize texts from the cropped image.
    - Classifies recognized texts into:
        - **Day**
        - **Start Time / End Time**
        - **Classroom Code**
    - Aligns times by comparing box centers with timetable headers (column matching).
    - Maps classroom codes to correct day and time slot.
4. **PaddleOCR API → Laravel (Response)**
    - API sends back the extracted timetable in **JSON format**, containing for each class:
        - Day

- Start Time
- End Time
- Location (classroom code).

5. **Laravel post-processing**
   - Laravel receives the JSON response.
   - Converts each class entry into a **ride entity** (with recurring ride pattern logic).
   - Stores rides in the database.
6. **Completion**
   - User receives confirmation in the Laravel app (e.g., rides created successfully from timetable).

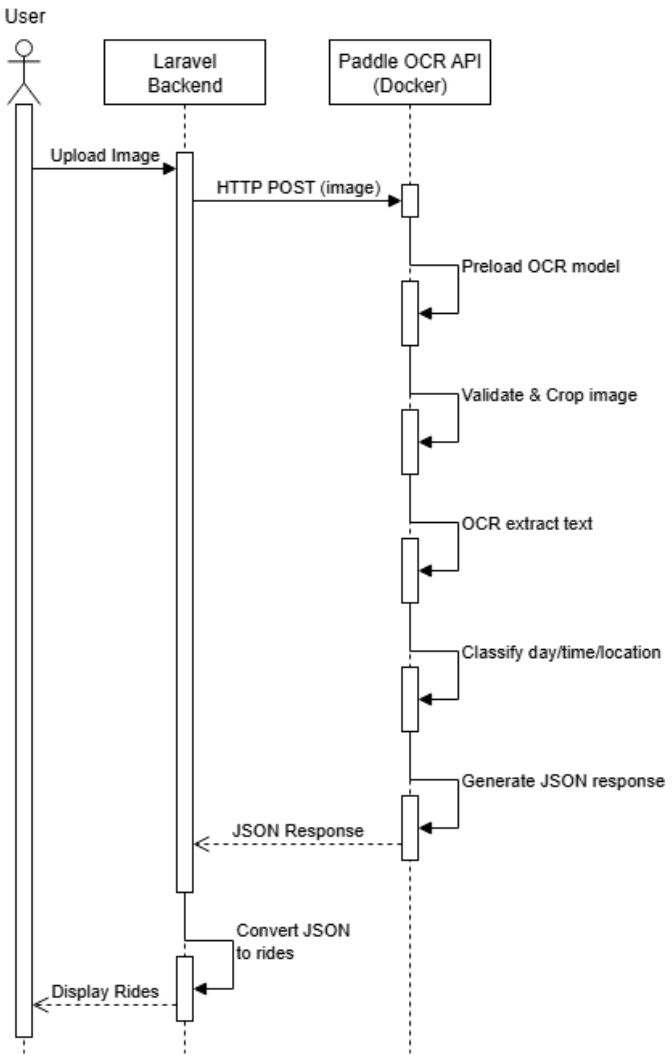


Figure 4.5 Paddle OCR API sequence diagram

4.4.3 Ride Posting – Timetable-based

In the timetable-based ride posting feature, the user uploads a timetable image to automatically generate potential recurring rides based on their class schedule. The image is processed by a self-developed PaddleOCR API (built with Python and FastAPI, with a custom-trained model for UTAR student timetables), which extracts the class details such as day, start time, end time, and location. The Laravel backend processes this extracted data to generate ride suggestions, following a set of rules:

- Create rides 30 minutes before a class starts and right after a class ends.
- Skip rides where the gap between two classes is less than or equal to 1 hour.

After reviewing the suggested rides, the user completes the remaining form fields, including home address (with Google Maps autocomplete), price, number of passengers, and vehicle details if offering a ride. Upon submission, the Laravel backend calculates distances and durations using the Google Maps DistanceMatrix API, validates all inputs, and stores the recurring rides in the database.

Interaction Flow:

1. **Upload Timetable**
    - The user uploads their timetable image via AJAX.
    - Laravel validates the file and sends it to the PaddleOCR API.
    - PaddleOCR returns JSON containing extracted class details.
    - Laravel processes the JSON and generates suggested rides based on the timetable rules.
    - Laravel returns the ride suggestions and displays the remaining input form to the user.
2. **Complete Ride Details**
    - The user fills in remaining fields (home address, start/end date, number of passengers, price, vehicle details, and description).

- Google Maps Autocomplete assists in entering the home address.

3. **Submit Form**
    - The completed form is submitted via AJAX to the Laravel backend.
    - Laravel validates inputs, calculates distance and duration via Google Maps DistanceMatrix API, and stores the rides in the database.
4. **Feedback to User**
    - On success → SweetAlert2 success message → redirect to ride list.
    - On validation or system error → display inline messages or SweetAlert2 error.

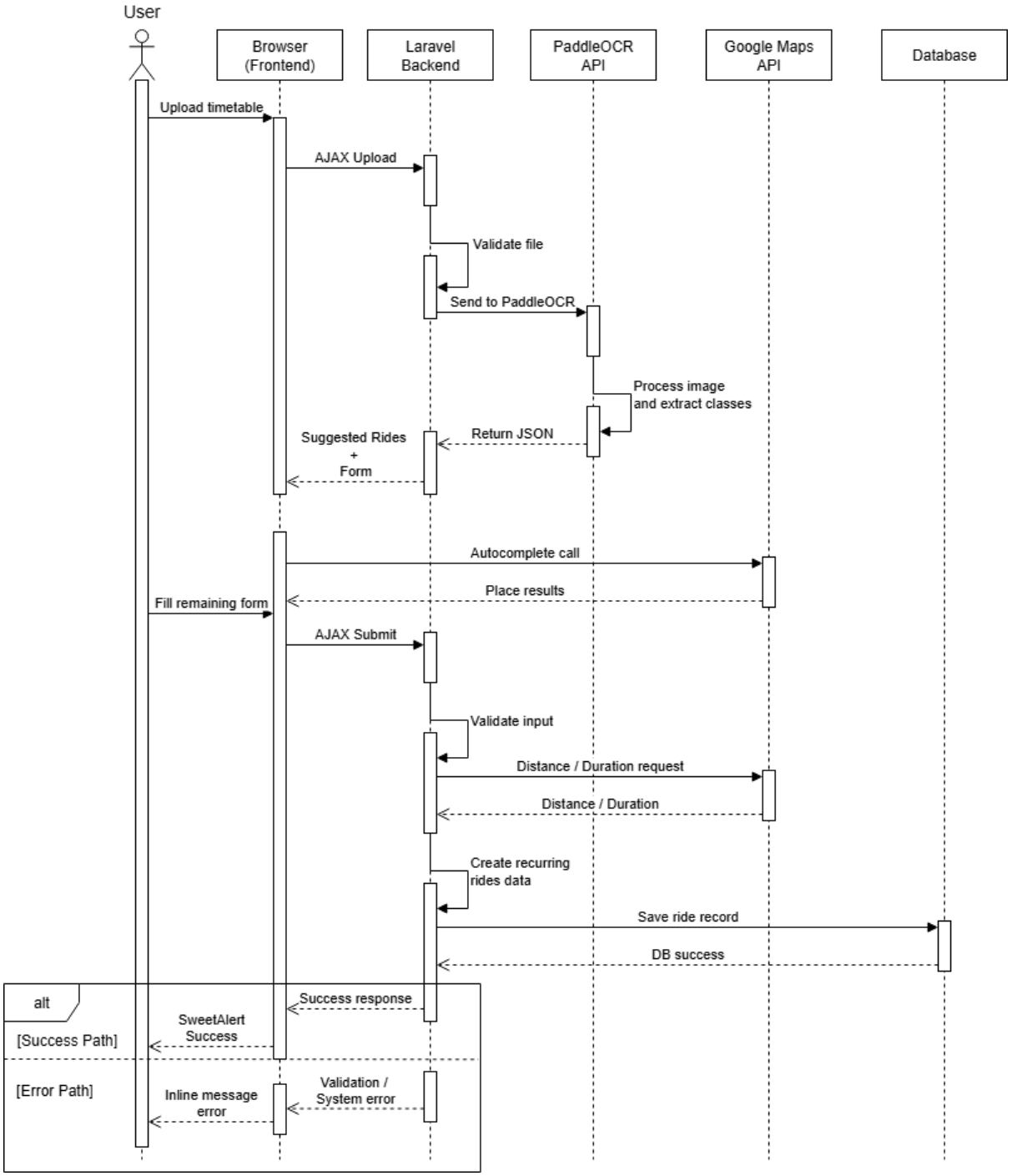


Figure 4.6 Ride Posting – Timetable-based Sequence Diagram

4.4.4 Ride Searching – Input fields

This feature allows users to search for rides based on specific criteria, including departure address, destination address, date, and ride type (offer/request). The departure and destination address fields use the same reusable Google Maps autocomplete Blade component used in the

ride posting forms, ensuring consistent UI and geocoding functionality. No distance or duration calculation is performed during the search stage.

Interaction Flow

1. **User Input** – The user enters the departure address, destination address, date, and ride type.
2. **AJAX Form Submission** – When the search form is submitted, JavaScript sends the input values via AJAX to the backend without reloading the page.
3. **Routing** – Laravel routes the AJAX request to the appropriate controller method based on the route definition.
4. **Backend Filtering** – The controller uses Laravel Eloquent queries to filter rides according to the input criteria. Recurring rides are grouped and labeled as such.
5. **Sorting** – Results are sorted by date, ensuring the soonest rides appear first.
6. **Response Data** – The backend returns the filtered results in JSON format.
7. **Dynamic Rendering** – JavaScript processes the returned JSON and updates the rides list dynamically in the DOM.
8. **No Match Case** – If no rides match the filters, the frontend displays the message *"There are no rides available"* with a button for the user to create a new ride.

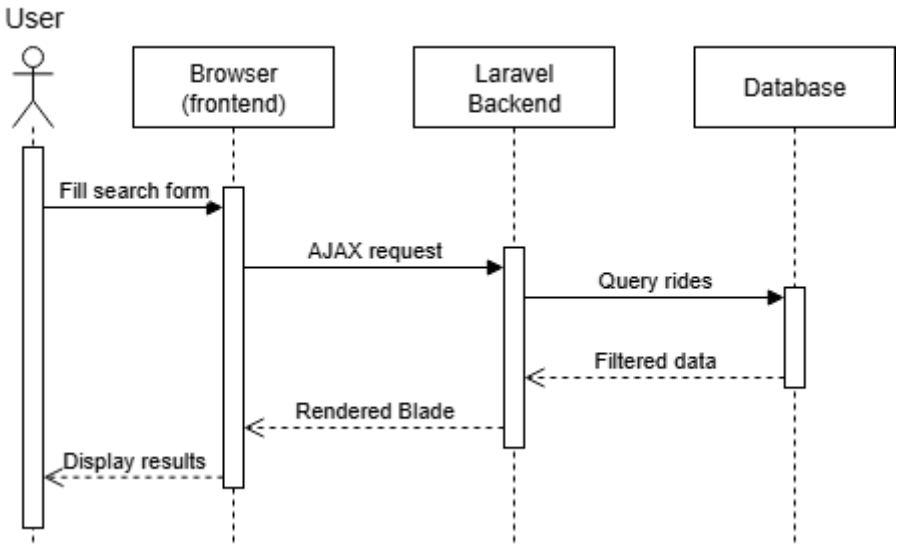


Figure 4.7 Ride Searching – Input fields Sequence Diagram

4.4.5 Ride Searching – Timetable-based

This feature allows users to search for available rides based on their personal timetable by leveraging the PaddleOCR API class extraction system. Instead of manually entering trip details, users can upload their timetable image, and the system automatically identifies potential rides that match their class schedule.

Interaction Flow

1. File Upload:
    - The user initiates the search by clicking a button to reveal the file upload input, then uploads their timetable image via AJAX.
2. Backend Processing (Extraction):
    - Laravel validates the uploaded file, then sends it to the self-hosted PaddleOCR API for timetable class extraction.
    - PaddleOCR returns a JSON containing class details (day, start time, end time, location).
    - Laravel applies the timetable ride extraction logic (same rules as timetable-based ride posting) to generate possible ride data from these classes.
    - The generated ride data is stored in the session for temporary use.
3. Redirection to Search Results:
    - If extraction succeeds, the backend responds to AJAX with a redirect route that includes a GET parameter (user_timetable=true).
    - The frontend redirects the user to this route.
4. Matching Against Database:
    - The controller detects the user_timetable=true parameter and retrieves the stored rides from the session.
    - Laravel uses Eloquent queries to match these session rides against existing rides in the database based on class time and location.

- Matching results are grouped (recurring rides grouped together) and sorted by date before being displayed.

5. Failure Handling:

- If OCR extraction fails, the user is notified via SweetAlert2 and remains on the upload page.
- If no matching rides are found, the search results page shows “No rides available” with an option to create a new ride.

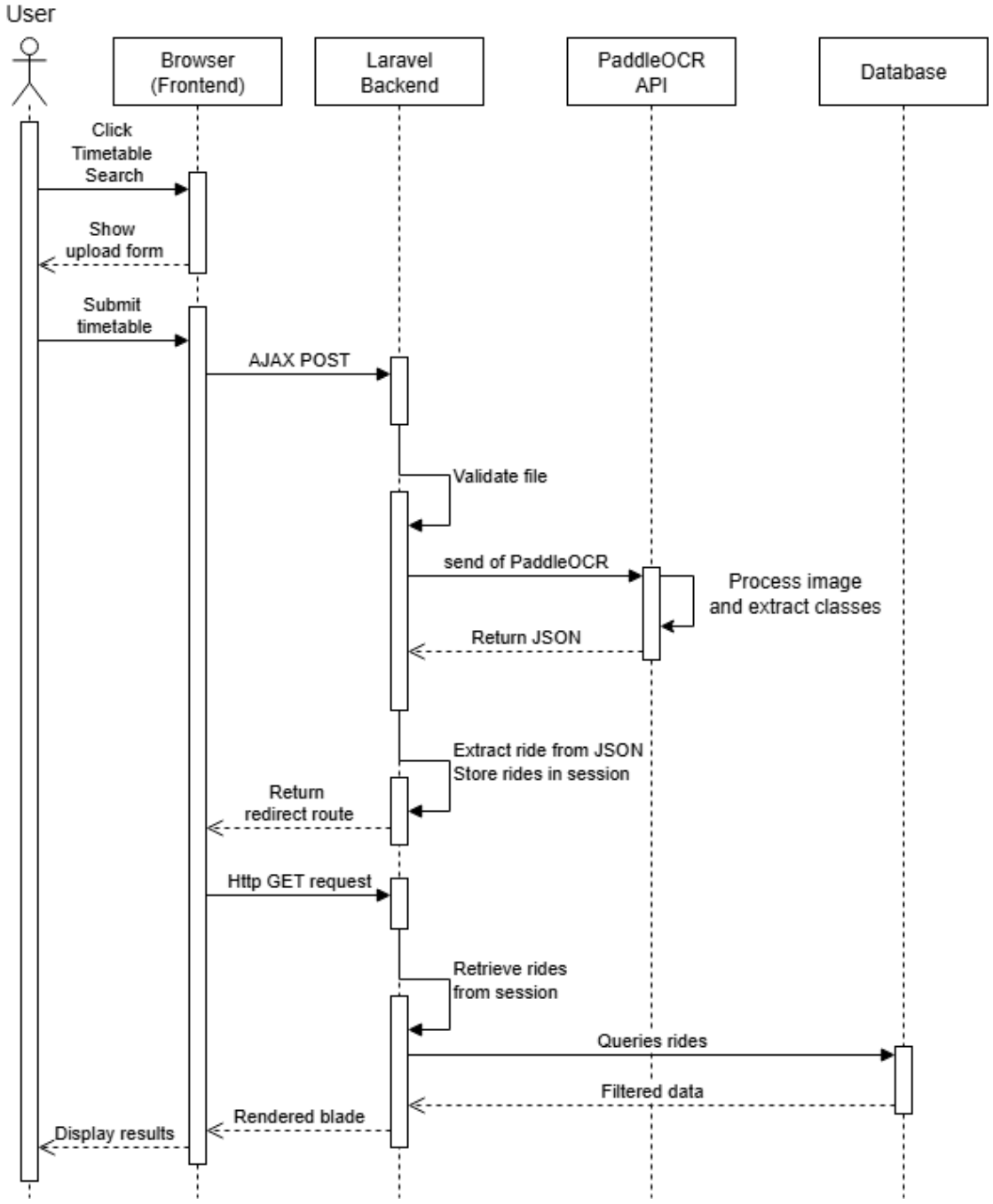


Figure 4.8 Ride searching timetable-based Sequence Diagram

4.4.6 Ride Searching – AI Chatbot

This feature allows users to search for rides by conversing in natural language through a chatbot interface, which interprets the user's message, extracts key ride details, and returns matching ride information directly within the chat.

Interaction Flow

1. User clicks the chatbot icon/button to open the chat interface.
2. User enters a query containing departure, destination, and date.
3. JavaScript sends the query via AJAX to a Laravel route.
4. Laravel retrieves a Dialogflow access token and sends the query to Dialogflow.
5. Dialogflow processes the query, extracts departure, destination, and date, and returns them to Laravel.
6. Laravel searches the database for rides matching the extracted data.
7. Laravel returns the first matching result to JavaScript via AJAX.
8. JavaScript updates the chat interface to show the ride result.
9. If required parameters are missing, the chatbot prompts the user for them.
10. If no matching ride exists, the chatbot displays "No rides found."

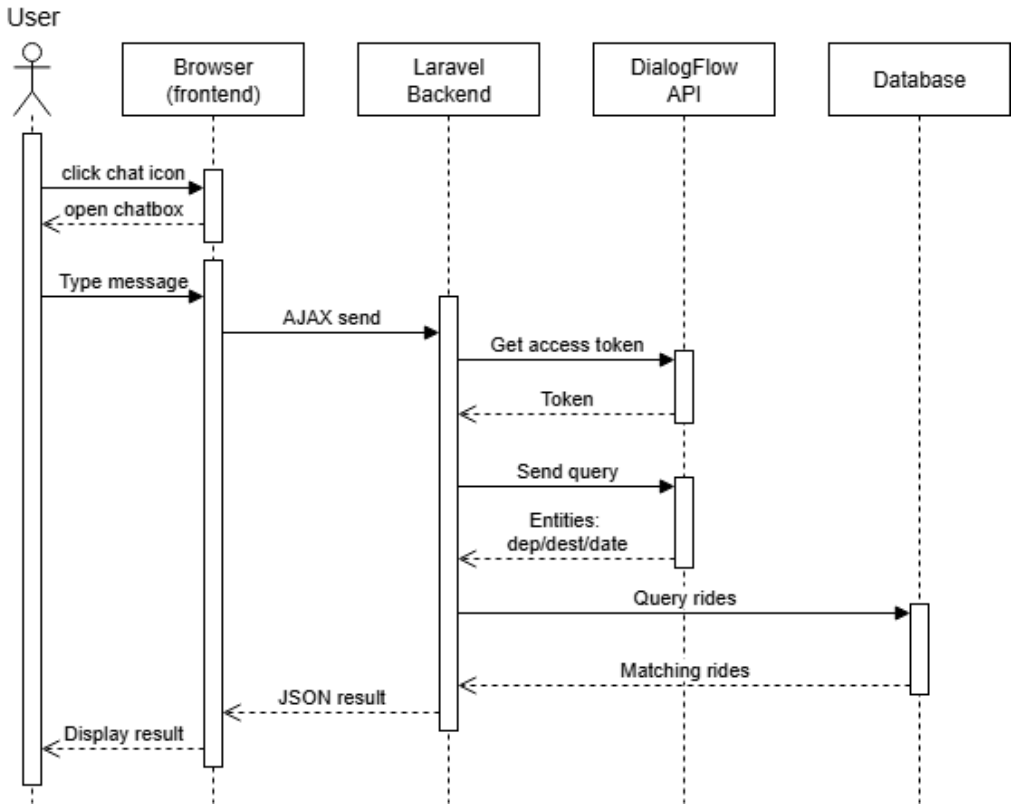


Figure 4.9 Ride searching – AI chatbot sequence diagram

4.4.7 Ride Booking

In the Ride Booking operation, a user selects a ride from the ride list and initiates booking from the ride details page via AJAX. Laravel validates the booking data (ride ID or recurring ride ID, and receiver ID), stores it in the bookings table, and updates the ride status to "pending." The system then returns a SweetAlert2 success message, or an error message if validation fails.

Interaction Flow:

1. **User Action**
   - The user navigates to the **ride details page** from the ride list.
   - The user clicks the **"Book Ride"** button to reserve the ride.
2. **Frontend Processing**
   - The booking request is sent via **AJAX** without reloading the page.
   - The request includes the ride_id (or recurring_id) and the receiver_id (driver's user ID), which are passed in the background and not entered manually by the user.

3. **Backend Processing – Laravel**
    - Laravel **validates** the request, ensuring that at least one of ride_id or recurring_id exists and is valid.
    - If validation passes:
        - The booking is stored in the **bookings table**.
        - The ride's status is updated to **"pending"**.
    - If validation fails, Laravel prepares an error message.
4. **Database Interaction**
    - **Insert** booking data into the bookings table.
    - **Update** ride status in the rides table.
5. **Response Handling**
    - If successful:
        - Laravel sends a success response back to AJAX.
        - SweetAlert2 shows a confirmation message to the user.
    - If unsuccessful:
        - Laravel sends an error response back to AJAX.
        - SweetAlert2 displays the error to the user.

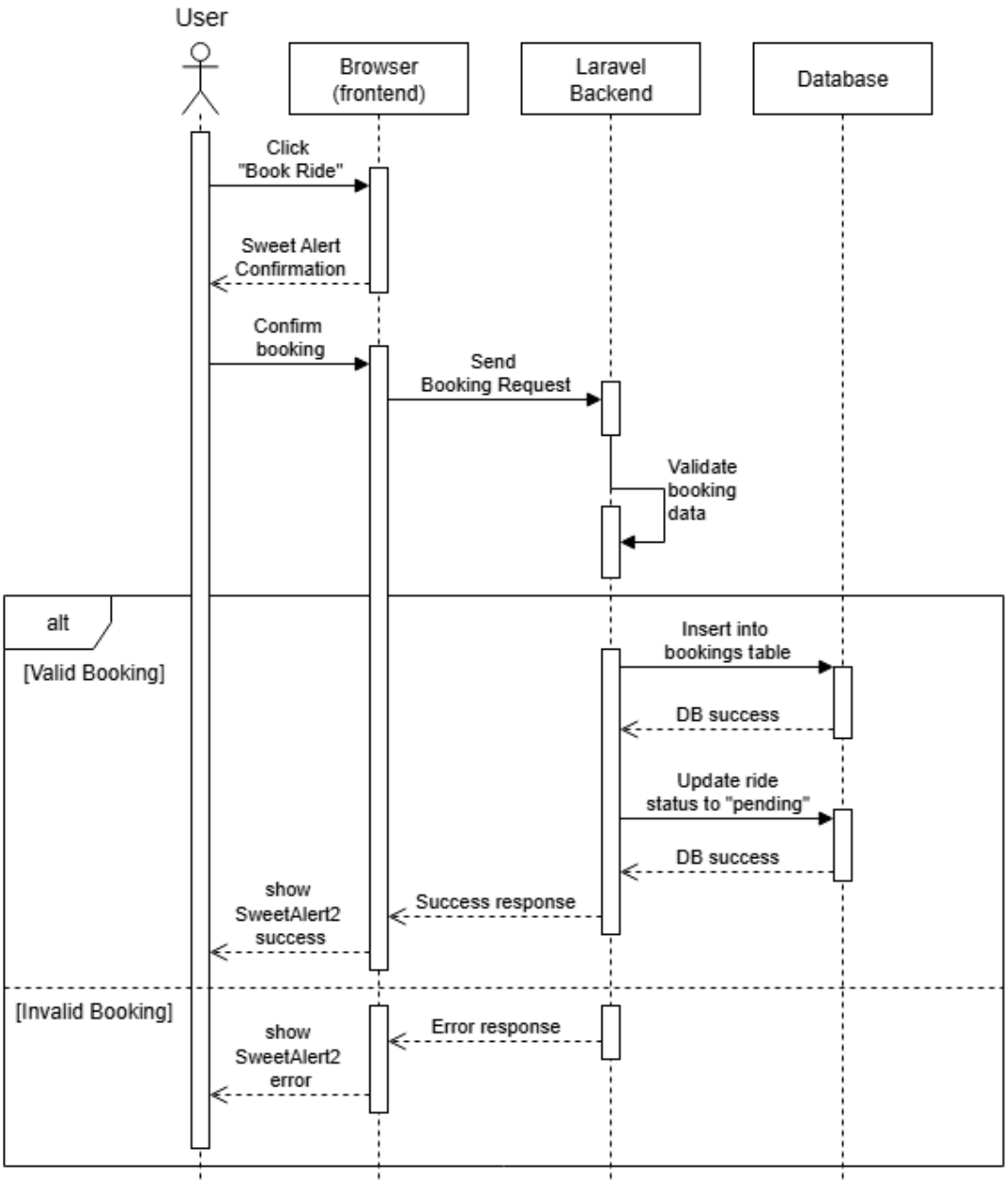


Figure 4.10 Ride booking sequence diagram

4.4.8 Ride Management

The Ride Management operation allows users to manage their rides and bookings from the dashboard, which contains three sections: *Manage Rides*, *Incoming Bookings*, and *Outgoing Bookings*. In *Manage Rides*, users can view, edit, or delete their rides. In *Incoming Bookings*, users can accept or reject booking requests, changing the booking status to either *accepted* or *rejected*. In *Outgoing Bookings*, users can cancel their own booking requests.

Interaction Flow

1. The user navigates to the *Dashboard* page, which contains three tabs: **Manage Rides**, **Incoming Bookings**, and **Outgoing Bookings**.
2. When the user clicks **Manage Rides**, they can choose to view, edit, or delete an existing ride.
    - If deleting, the action is sent via AJAX to the Laravel controller, which validates the request, deletes the ride from the database, and cascades the removal of any related data.
    - Upon successful deletion, the system returns a success response to the browser, which displays a SweetAlert2 notification.
3. In **Incoming Bookings**, the user can accept or reject booking requests.
    - The choice is sent via AJAX to the Laravel controller, which updates the booking status in the database to *accepted* or *rejected*.
    - The updated status is returned to the browser and displayed using SweetAlert2.
4. In **Outgoing Bookings**, the user can cancel a booking request.
    - The cancellation request is sent via AJAX to the Laravel controller, which updates or deletes the booking record in the database.
    - A confirmation is returned to the browser, and the user is notified via SweetAlert2.

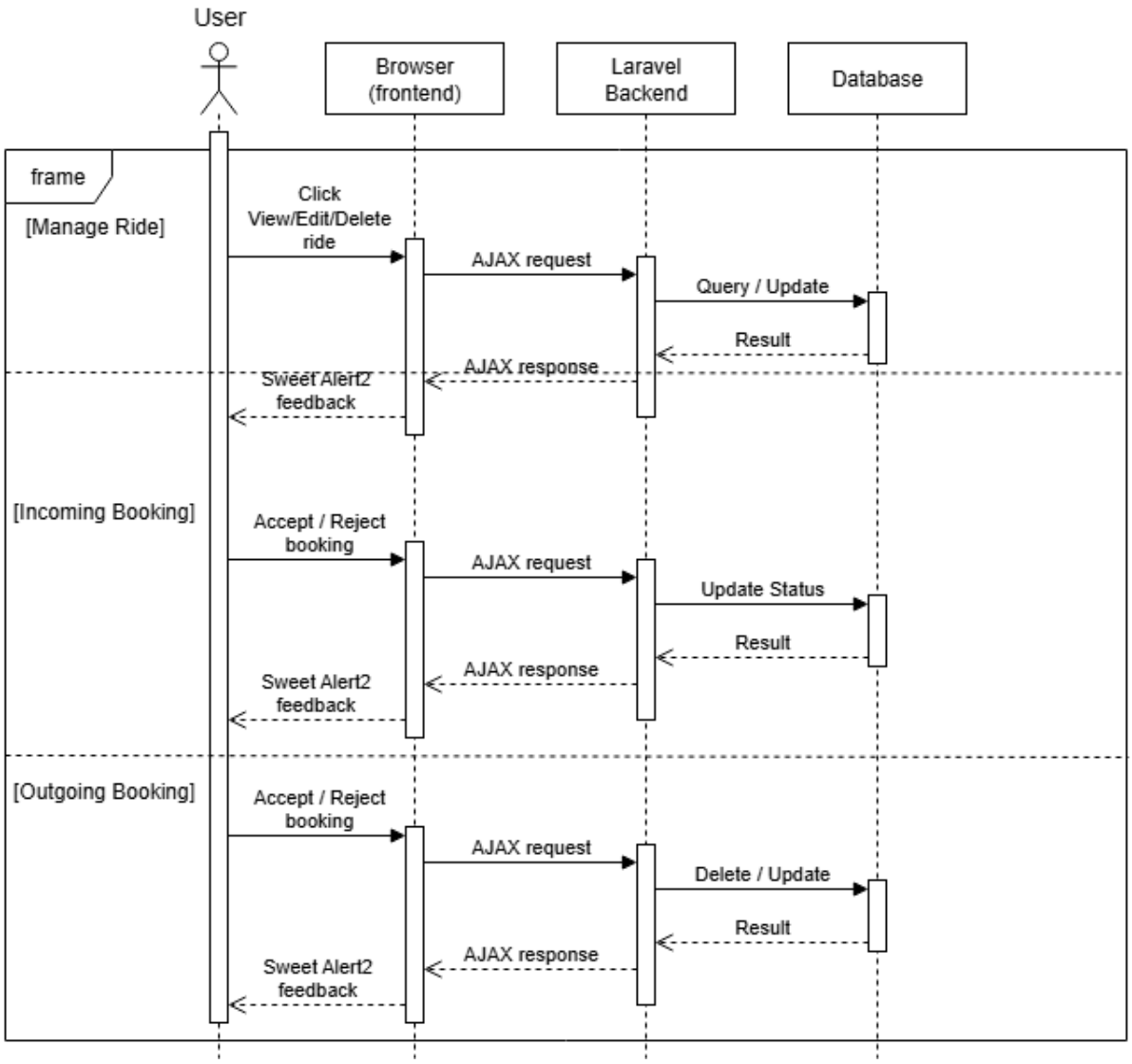


Figure 4.11 Ride management Sequence Diagram

Chapter 5
System Implementation

5.1 Hardware Setup

5.1.1 Local Development Environment

The system was developed and tested locally on a laptop machine. The specifications of laptop were shown in Table 5.1.

Table 5.1 Specifications of laptop

Description	**Specifications**
Model	HP Victus 16
Processor	AMD Ryzen 5-8645HS
Operating System	Windows 11
Graphic	NVIDIA GeForce RTX 4050
Memory	16GB DDR4 RAM
Storage	512GB SSD

- **Development Stack:**
  - Laravel framework set up through Laragon as a local web server environment.
  - Docker was used locally to containerize and test services, particularly for ensuring consistency in environment setup before deployment.
- **Testing Tools:**
  - Ngrok was employed to expose the local server to the internet during development, allowing for online testing of APIs and external integrations (such as Dialogflow chatbot and Google Maps API).

This setup ensured a smooth and isolated environment for iterative development, debugging, and testing.

5.1.2 Deployment Environment

For deployment, both the web application and supporting services were hosted on cloud infrastructure:

- Laravel Cloud:

  The main web application was hosted on Laravel Cloud, ensuring scalability, security, and continuous deployment support.
- Azure database hosting:

  The data within the website was stored and hosted in Azure, deployed via Azure Database.
    - MySQL flexible servers:
        - Instance type: Burstable, B1ms
        - vCores: 1
        - Memory: 2 GiB
        - Storage: 20 GB
        - Disk IOPS: 360

This deployment configuration ensured cost efficiency while providing sufficient resources to handle data request.

5.2 Software Setup

5.2.1 Operating System and Local Development

This layer provides the foundation for the project's development environment. It includes the operating system, local server management tools, and integrated development environments (IDEs) used to build and test the system before deployment.

1. **Operating System – Windows 11**

   The project development is carried out on Windows 11, which provides a stable and user-friendly environment compatible with required tools such as Laravel, MySQL, Docker, and Python.

2. **Local Development Environment – Laragon**

   Laragon is used as the primary local development environment. It simplifies the setup of Laravel projects by providing pre-configured services such as Apache, PHP, and MySQL. Laragon allows rapid project initialization (via *Quick App*) and efficient management of multiple local applications.

    - Download link: https://laragon.org/download

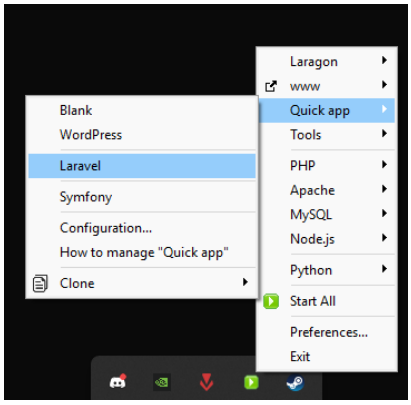


Figure 5.1 Project Initialization via Quick App

3. **Python Runtime for OCR Service (v3.10)**

   In addition to PHP, Python is used to develop the OCR service (FastAPI + PaddleOCR). This service runs separately from the Laravel application but is integrated via API calls. Python ensures compatibility with machine learning and OCR libraries required for timetable extraction.

   - Download link: https://www.python.org/downloads/

4. **Integrated Development Environments (IDEs)**

   Two IDEs are primarily used for development:

   - PhpStorm: Used for Laravel and PHP development, providing advanced code navigation, debugging, and project structure management.
     - Download link: https://www.jetbrains.com/phpstorm/download/?section=windows
   - PyCharm: Used for Python development, particularly for implementing and testing the FastAPI service with PaddleOCR.
     - Download link: https://www.jetbrains.com/pycharm/download/?section=windows

5.2.2 Backend Frameworks and Runtime

This layer defines the frameworks, languages, and runtime environments that power the server-side logic of the system. It includes both the PHP-based Laravel

framework for the core web application and FastAPI (Python) for AI-related services.

- **Laravel (v11.45.1)**
    - Acts as the primary backend framework for the system.
    - Provides MVC (Model-View-Controller) architecture, simplifying code organization and maintainability.
    - Offers built-in support for routing, middleware, validation, authentication, and session handling.
    - Used as the foundation for building ride listing, booking, and user interaction functionalities.
- **PHP Runtime (v8.2.27)**
    - Serves as the execution environment for the Laravel framework.
    - Integrated within **Laragon** for local development and compatible with cloud deployment environments.
- **Composer (v2.8.4)**
    - Dependency management tool for PHP and Laravel.
    - Used to install and manage packages such as Laravel UI, authentication libraries, and external integrations.
    - Installation command:
        - php -r "copy('https://getcomposer.org/installer', 'composer-setup.php');"
        - php composer-setup.php
        - php -r "unlink('composer-setup.php');"
        - sudo mv composer.phar /usr/local/bin/composer
- **Artisan Command-Line Tool**
    - Provides an interface for executing common Laravel tasks.
    - Extensively used for:
        - Database migrations (php artisan migrate)

- Model and controller generation (php artisan make:model, php artisan make:controller)
- Factory and seeder creation (php artisan make:factory, php artisan db:seed)
- Running local servers and cache management (php artisan serve, php artisan cache:clear).

5.2.3 Frontend Technologies

1. **Laravel Blade Templates**
   - Version: Bundled with Laravel 11
   - Purpose/Usage: Provides server-side templating engine for structuring frontend views and rendering dynamic content directly from Laravel controllers.
2. **Tailwind CSS**
   - Version: Latest stable (via CDN)
   - Purpose/Usage: Utility-first CSS framework used to rapidly design responsive and consistent UI components with minimal custom CSS.
   - Code to include in layout file:
     - <script src="https://cdn.tailwindcss.com"></script>
   - Add these in /resources/css/app.css
     - @taildwind base;
     - @tailwind components;
     - @taildwind utilities;
3. **BladewindUI Components**
   - Version: Latest stable release
   - Purpose/Usage: Pre-built Tailwind-based UI component library integrated into Blade templates to accelerate frontend development with clean and reusable elements.

- Installation command: composer require mkocansey/bladewind
- Code to include in layout file:
    - <link href="{{ asset('vendor/bladewind/css/animate.min.css') }}" rel="stylesheet" />
    - <link href="{{ asset('vendor/bladewind/css/bladewind-ui.min.css') }}" rel="stylesheet" />
    - <script src="{{ asset('vendor/bladewind/js/helpers.js') }}"></script>
    - <script src="//unpkg.com/alpinejs" defer></script>

5.2.4 Database

- **Database System**: MySQL
- **Version**: 8.0.30
- **Local Management Tool**: phpMyAdmin (via Laragon environment)
- **Schema & Data Management**: Laravel migrations, seeders, and factories for schema evolution and dummy data generation
- **Deployment Database**: Azure MySQL Flexible Server for production hosting
- Code for Database Migration may refer to my GitHub repository (/database/migrations): https://github.com/JamesOtter/carpool-web OR appendix

5.2.5 Cloud / Hosting

- **Laravel Cloud**

  Used for hosting the production version of the Laravel application.

  Link: https://cloud.laravel.com/

    - Import from GitHub repository
    - Create an application
    - Create an environment
    - Click on deploy
- **Ngrok**

Utilized for temporary hosting during testing phases. It exposes the local Laravel server to the internet via a public URL, which is useful for testing APIs and chatbot integrations before production deployment.

- Download link: https://ngrok.com/docs/getting-started/
- Installation command:
  - ngrok config add-authtoken <your_auth_token>

- **Azure MySQL Flexible Server**

Cloud database hosting solution used in production. Provides managed MySQL service with scalability, high availability, and security features, ensuring the application's database is reliable in production.

Link: https://portal.azure.com/#browse/Microsoft.DBforMySQL%2FflexibleServers

- Using Quick Create flexible server
- Setup administrator login and password

5.2.6 External APIs

1. **Google Maps API**
   - **Purpose**: To provide location intelligence features such as address autocompletion, route planning, and distance calculation.
   - **Steps to setup**:
     - Create a Google Cloud Project: https://cloud.google.com/
     - Enable these APIs (Maps JavaScript API, Places API, Directions API, Geocoding API)
     - Generate an API key
     - Code in layout file: <script src="https://maps.googleapis.com/maps/api/js?key=YOUR_API_KEY&libraries=places"></script>

2. **Dialogflow API**
   - **Description**: A conversational AI platform by Google for building chatbots and virtual assistants.
   - **Purpose**: To enhance user interaction by allowing natural language queries to find or book rides.

- **Usage**: Integrated into the Laravel-based frontend as a chatbot, helping users search for rides and providing a direct link to ride details.
- **Link**: https://dialogflow.cloud.google.com
- **Steps to setup:**
    - Create a Dialogflow Agent
    - Set up a google cloud project
    - Generate Service Account Credentials (JSON Key File)
    - Store the JSON Key in Laravel Project (/storage/app/dialogflow/)

3. **PaddleOCR (PP-OCRv5)**
    - **Description**: An open-source Optical Character Recognition (OCR) library developed by Baidu. In this project, **PP-OCRv5** is used for extracting structured text data from uploaded timetables. To ensure efficiency, the model is **preloaded inside a Dockerized FastAPI service**, which acts as a middleware between the Laravel application and the OCR model.
    - **Deployment**:
        - **Local development**: PaddleOCR runs inside a Docker container with FastAPI for easy testing and model reuse.
        - **Production**: The Dockerized FastAPI service will be hosted on **Azure Container Instances**, allowing scalable and containerized deployment.
    - **Docker download link (Window version): https://www.docker.com/**
    - **Paddle OCR installation command (to test locally):**
        - pip install paddleocr
        - pip install paddlepaddle

5.2.7 Version Control and Collaboration

1. **Git**
    - **Description/Purpose**: A distributed version control system used to track changes in source code, manage project history, and support collaborative development.

- **Usage**: Used throughout the project lifecycle to commit, branch, and merge code changes. Helps maintain clear versioning and rollback capabilities.

2. **GitHub**

- **Description/Purpose**: A cloud-based platform for hosting Git repositories with collaboration, pull request, and code review features.
- **Usage**: The project repository is hosted on GitHub to enable team collaboration, centralized code storage, and integration with deployment pipelines.
- **My project repository link**: https://github.com/JamesOtter/carpool-web

5.3 Setting and Configuration

5.3.1 Backend and Database Configuration

- Environment variables (.env in Laravel) for database and API keys [in Local].
  - Make sure the variables are such as below figure.

Figure 5.2 Backend and database configurations in local

- Config file in Laravel [in Local].
  - /config/database make sure it is mysql

Figure 5.3 MySQL configuration

- In Laravel Cloud [production]:
  - Navigate to settings > Custom environment variables, then add these variables

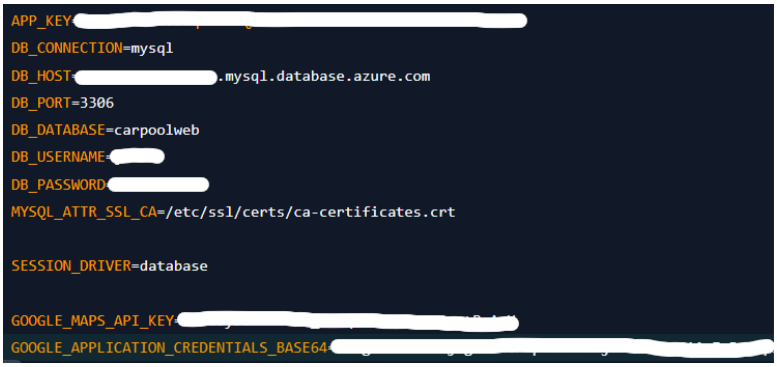


Figure 5.4 Backend and database configuration in deployment

5.3.2 External API Configuration

1. Dialogflow agent integration

- Create Intent from Intents Tab
- In Training phrases add phrases such as figure below.

Figure 5.5 DialogFlow training phrases

- In Action and parameters add such as figure below.

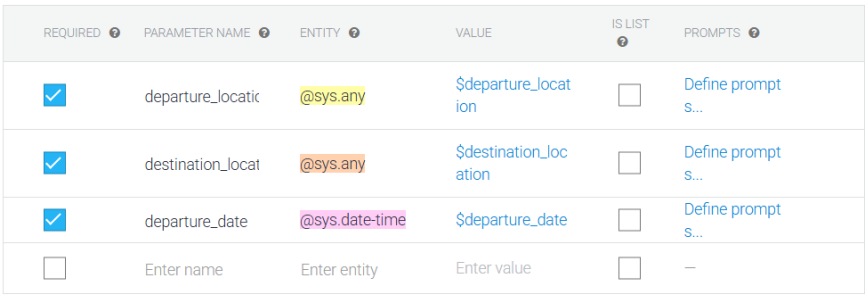


Figure 5.6 DialogFlow actions and parameters

- Go to Fulfilment tab to enable webhook and configure your URL to access this API.

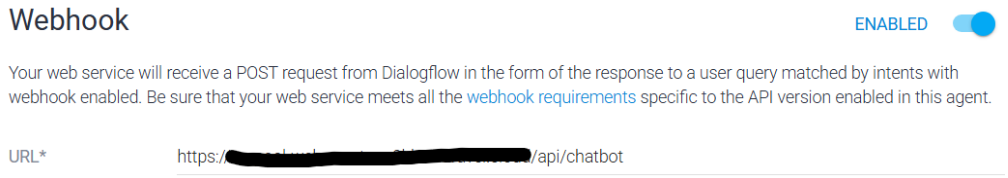


Figure 5.7 DialogFlow webhook

2. Paddle OCR FastAPI endpoint configuration

- In "app.py" use the following code. Thus, the API can be accessed through /ocr-timetable

```
from fastapi import FastAPI

/
app = FastAPI()

/ocr-timetable
@app.post("/ocr-timetable")  new *
async def ocr_timetable():
```

Figure 5.8 FastAPI configuration

5.3.3 Cloud and Hosting Configuration

1. Docker configuration

- Create a requirement.txts to include package to be downloaded

```
python-multipart
fastapi
uvicorn
paddleocr
paddlepaddle
opencv-python
```

Figure 5.9 Docker requirements

- Create a Docker file to run the python code and host the FastAPI server.

```
# Use an official Python base image
FROM python:3.10-slim

# Install system dependencies
RUN apt-get update && apt-get install -y \
    libgl1-mesa-glx \
    libglib2.0-0 \
    libgomp1 \
    && apt-get clean

# Set working directory inside the container
WORKDIR /app

# Copy files
COPY . /app

# Copy your script and any required files into the container
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Run FastAPI server
CMD ["uvicorn", "app:app", "--host", "0.0.0.0", "--port", "80"]
```

Figure 5.10 Docker file to run

- Command to run docker (in Local):
  - docker build -t timetable-ocr

2. **Azure Container Instance**
   - Command to publish local image to Docker hub
     - docker tag <name>/timetable-ocr
     - docker push <name>/timetable-ocr
   - Create Azure Container Instance in azure portal
     - Image source: Other registry
     - Image: <name>/timetable-ocr
     - OS type: Linux
     - Size: 4 vcpu, 4 GiB memory

5.4 System Operation

5.4.1 System Startup and Initialization

- **Local Environment**

1. Select “start all” to start Laragon

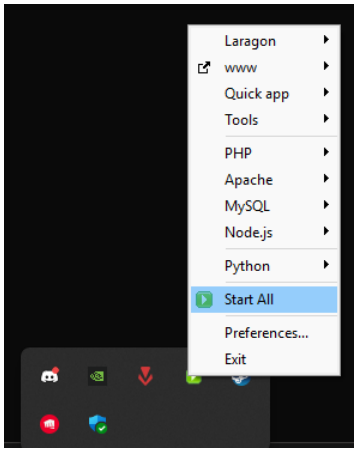


Figure 5.11 System start-up in local environment

2. Go to project level CMD and start local host
    - php artisan serve
3. In same level, open another CMD to tunnel local host to public by using Ngok
    - ngrok http 8000
4. Command to start Docker
    - docker run -p 8010:8010 -v "C:/laragon/www/CarpoolWeb/storage/app/public:/app/storage" timetable-ocr

- **Production Environment**
    1. Deploy website in Laravel Cloud

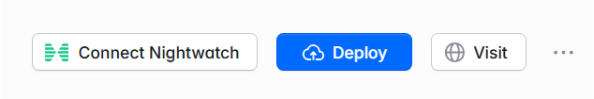


Figure 5.12 System start-up in production environment

2. Start Azure Database in azure portal
3. Start Azure Container Instance in azure portal

5.4.2 User Roles

1. General User

- Every registered account in the system is a **User**.
- A User can act as either:
    - **Driver** – posts ride offers with trip details.
    - **Rider** – posts ride requests or books available ride offers.

- This flexibility allows users to switch roles depending on their need (e.g., one day they drive, another day they ride).

2. Driver Role (when offering rides)

- A **Driver** is a user who creates an **offered ride**.
- Required inputs typically include:
    - Starting point & destination (via Google Maps API)
    - Date & time of departure
    - Available seats
    - Vehicle number and model
    - Suggested fare
- Drivers can:
    - Post and manage their rides in the **Dashboard**
    - Accept bookings from Riders
    - Communicate with Riders (e.g., WhatsApp link)

3. Rider Role (when requesting rides)

- A **Rider** is a user who creates a **ride request**.
- Required inputs may include:
    - Pickup location & destination
    - Preferred time
    - Number of seats needed
- Riders can:
    - Post and manage their ride requests in the **Dashboard**
    - Book available rides posted by Drivers
    - Cancel or update bookings

4. Dual Role (User as Rider & Driver)

- A user can **switch roles dynamically** depending on the context.
- Example:
    - In the morning, a user may act as a **Driver** when driving to campus.
    - In the evening, the same user may act as a **Rider** when booking a ride home.
- This flexibility makes the system **community-driven** instead of a strict platform with separated roles.

5.4.3 Normal Operation Workflow

1. User Registration and Login

The registration process begins when a new user accesses the registration page and provides their details, including name, email, contact number, password, and password confirmation. Once the information is submitted, the system validates the inputs to ensure that the required fields are not empty, the email is in a valid format, and the password matches the confirmation field. If any validation fails, the system immediately returns an error message to guide the user in correcting the input. Upon successful validation, the user's information is securely stored in the system database, and the account becomes active for login.

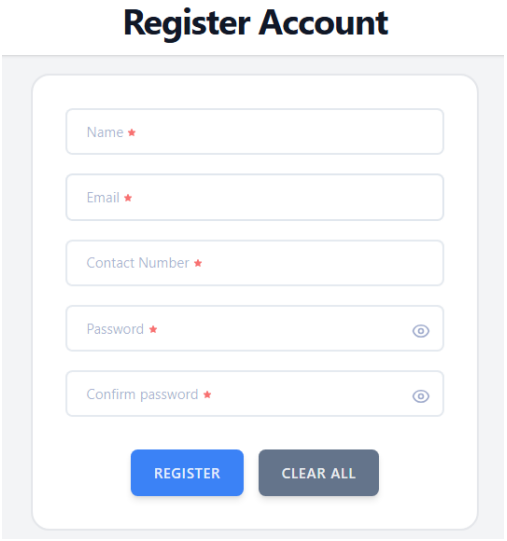


Figure 5.13 Registration form

For login, the user is required to provide only their registered email and password. The system validates these credentials against the stored records in the database using Laravel's built-in authentication mechanism. If the credentials are incorrect, the system displays an error message prompting the user to retry. Once authenticated successfully, the user is granted access to the system's dashboard and can proceed to use the available features such as ride creation, ride search, and booking. This workflow ensures a secure and user-friendly authentication process that forms the foundation for accessing the system.

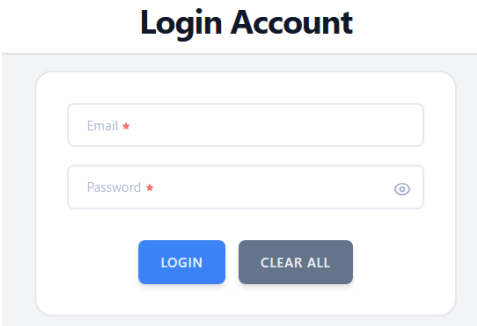


Figure 5.14 Login form

2. **Ride Posting – Basic form**

Once registered and logged in, users can proceed to post a ride within the system. The ride posting feature supports both **ride offers** (drivers offering seats in their vehicle) and **ride requests** (passengers seeking a ride). During this process, the user is required to provide key ride details including the **departure address, destination address, date and time of travel, ride type, number of passengers, and the proposed price**. To ensure accuracy, the system enforces the use of the **Google Maps Autocomplete API**, which validates addresses and prevents invalid or ambiguous entries. Additionally, the system restricts users from selecting past dates or times to maintain the integrity of future ride scheduling.

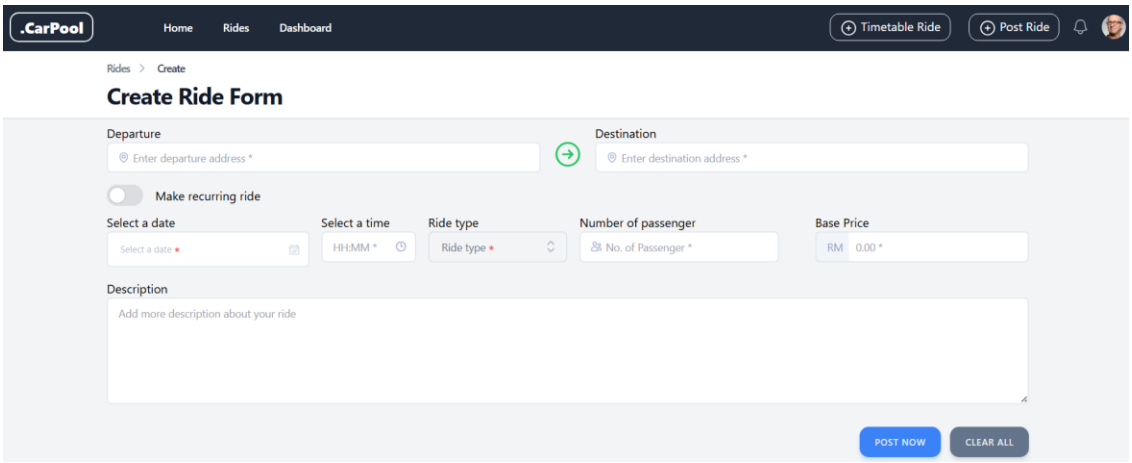


Figure 5.15 Ride Form – One time ride

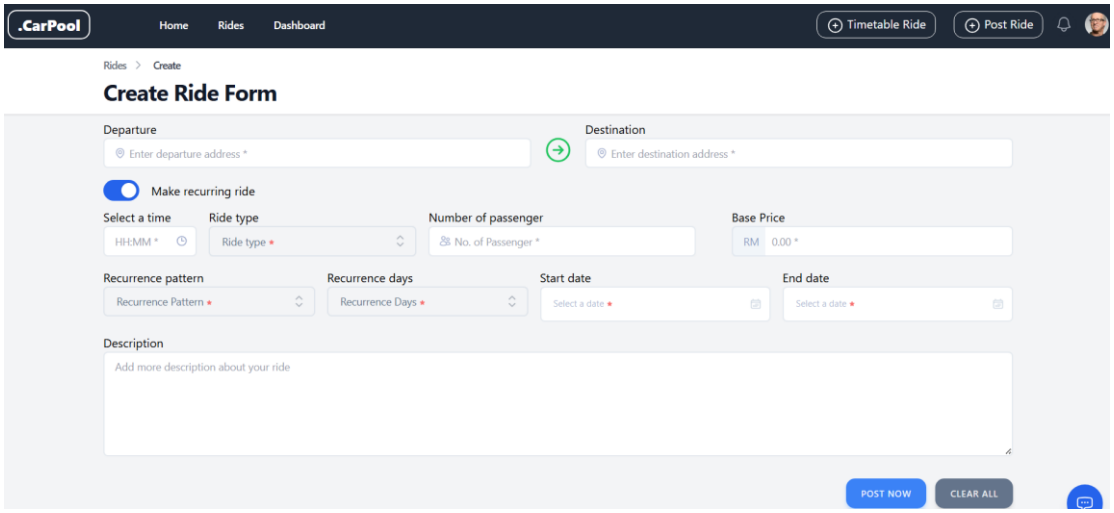


Figure 5.16 Ride Form – Recurring rides

After submission, the ride is stored in the database and becomes **immediately available in the ride listings** for other users to view, though a page refresh may be required to reflect the newly added ride. The system accommodates both **single rides** (one-time trips) and **recurring rides**, where users can define repeated journeys over a specific period. This recurring ride functionality reduces the need for drivers or passengers to repeatedly create identical listings for regular trips such as daily commutes.

3. Paddle OCR API

The normal operation flow of the PaddleOCR API begins when a timetable image is submitted from the Laravel application via an HTTP POST request. Once received, the image is processed through the FastAPI service running inside a Docker container. The system first applies OpenCV (cv2) techniques to crop the timetable region from the uploaded image, ensuring only the relevant portion is passed for text recognition.

The OpenCV techniques consists of:

a. Convert image to grayscale to simplify further processing
b. Apply binary inverse thresholding: bright pixels become black and darker pixels become white, making text / line pop as white shapes on dark backgrounds.
c. Detect external contours (the outer boundaries of connected white regions) in the thresholded image.
d. Select the largest contour by area, assuming it corresponds to the timetable grid.
e. Compute a bounding rectangle around the largest contour.
f. Crop the original image to that rectangle, isolating the main region of interest.

For an example, figure 5.17 show an image of timetable with excess space. In figure 5.18 show the image of timetable after cropping.

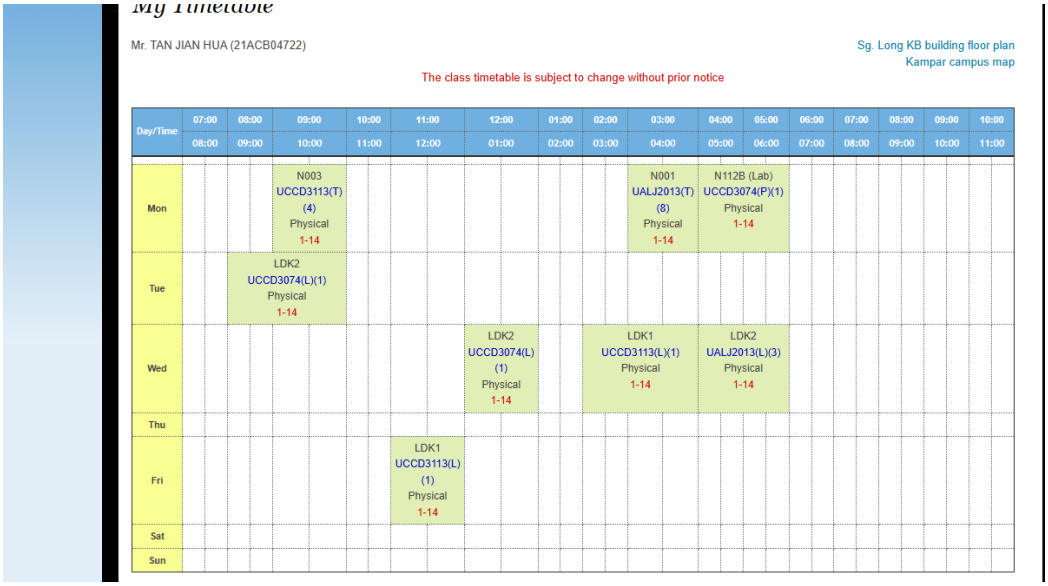


Figure 5.17 Timetable before cropping

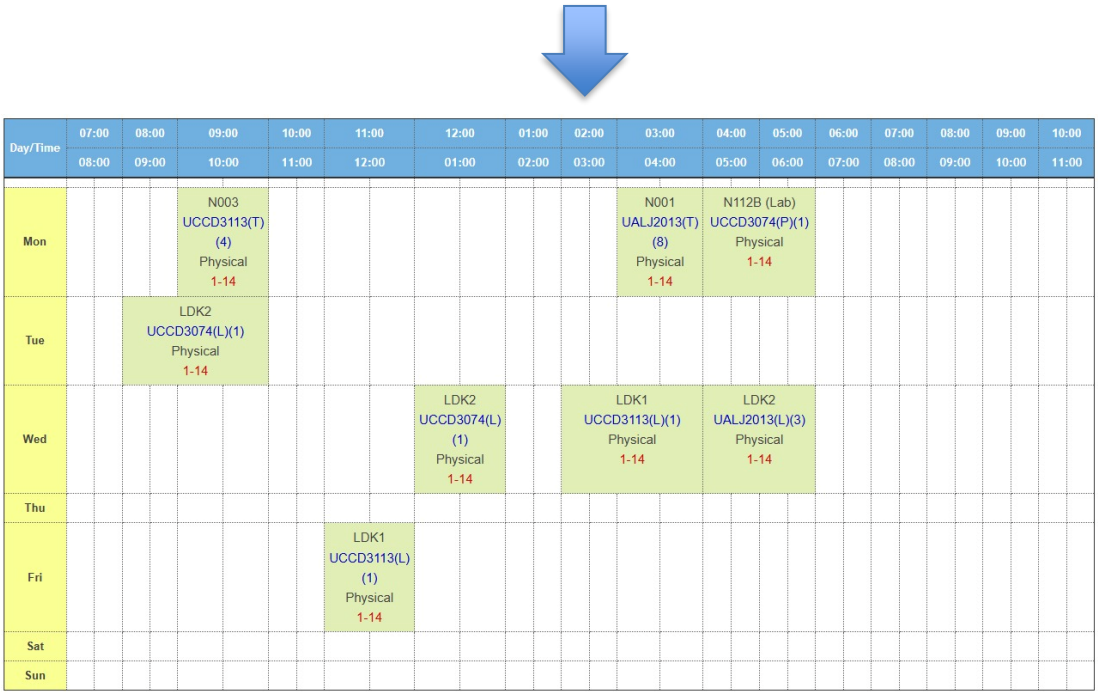


Figure 5.18 Cropped timetable

The cropped timetable is then processed using PaddleOCR, which extracts the text content along with the corresponding bounding box coordinates.

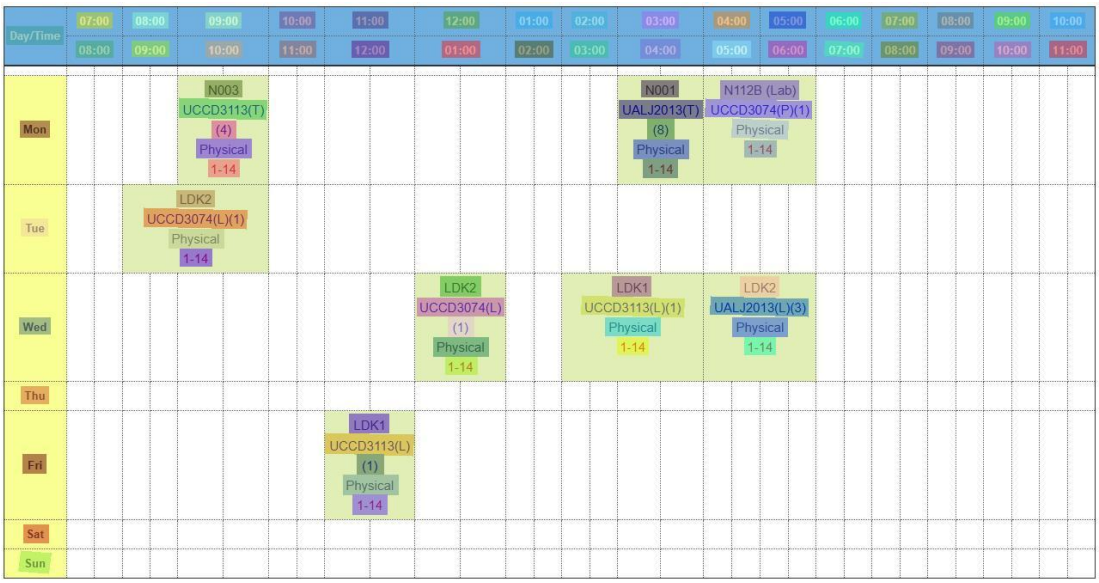


Figure 5.19 Visualized detected text on image

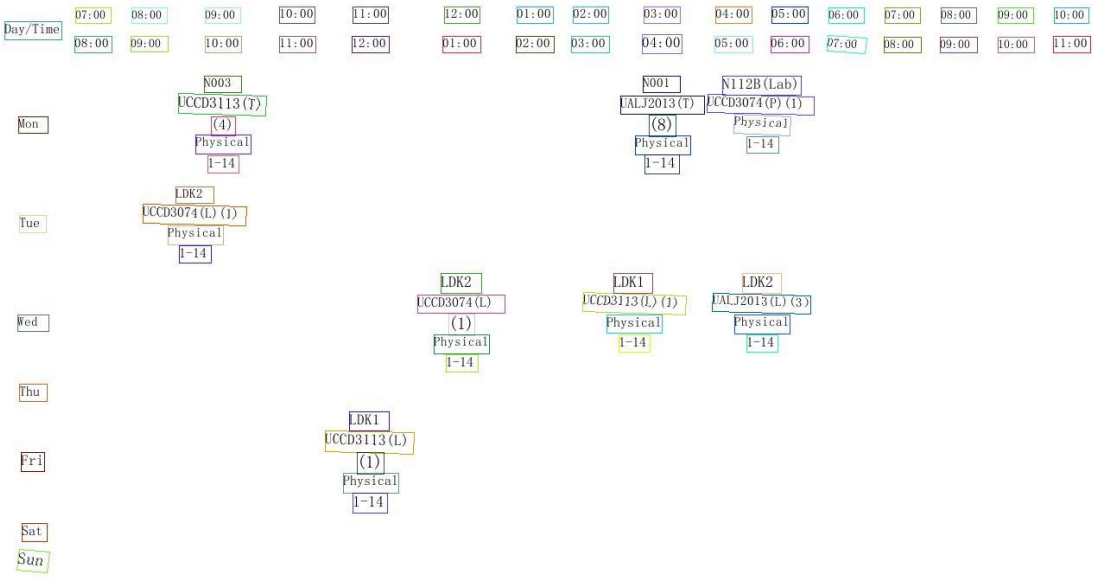


Figure 5.20 Visualized detected text and bounding box

The recognized text is subsequently classified into specific categories, including day, time, and classroom codes. To structure the information accurately, the system maps each extracted classroom code to the appropriate day and time column, referencing the time header that contains the start and end times for each slot.

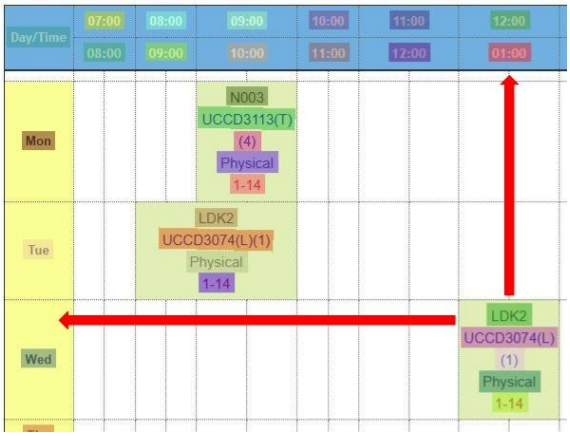


Figure 5.21 Example of classroom code mapping

After mapping, the data is organized into a structured JSON output, where each entry contains details of a class with its associated day, start time, end time, and location (derived from the classroom code). This JSON response is returned on-the-fly to the Laravel application and is directly integrated into the carpooling ride controller, enabling the extracted timetable information to be seamlessly converted into ride creation data. The Laravel backend processes this extracted data to generate ride suggestions, following a set of rules:

- Create rides 30 minutes before a class starts and right after a class ends.
- Skip rides where the gap between two classes is less than or equal to 1 hour.

Given an example in figure 5.22, red boxes indicate the rides will be created.

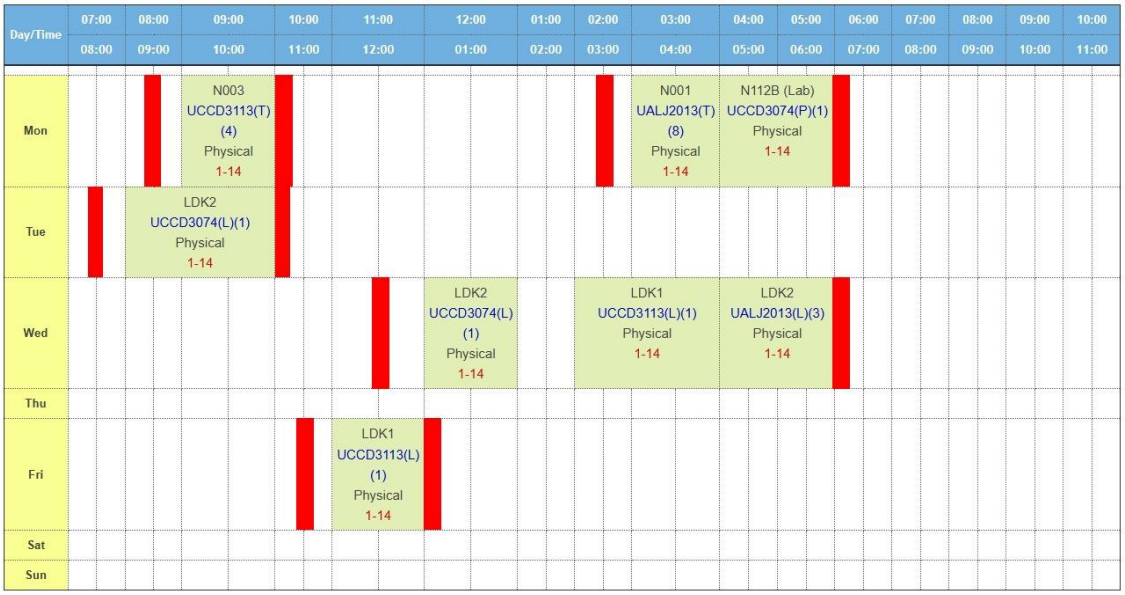


Figure 5.22 Example of rides will be created

4. **Ride Posting – Timetable based**

In the timetable-based ride posting workflow, the process begins when a user uploads an image of their academic timetable into the system.

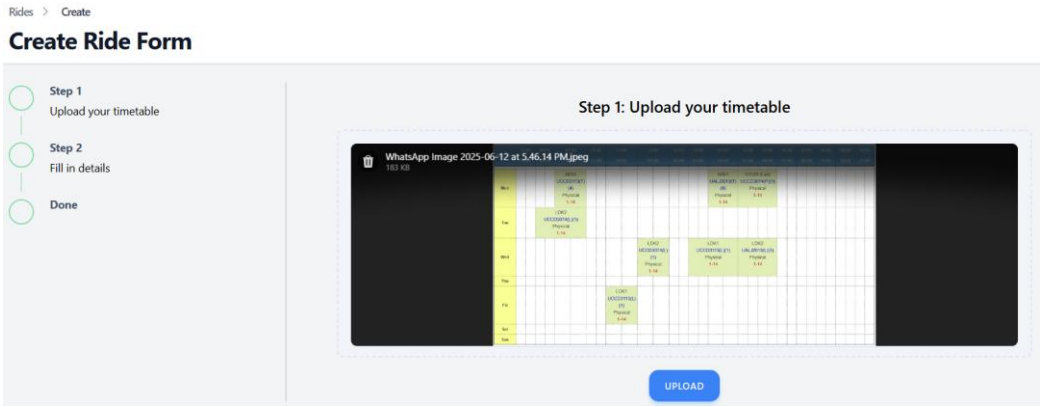


Figure 5.23 Upload timetable form

The uploaded timetable is first processed by the PaddleOCR engine, which extracts the textual information such as class times, subjects, and classroom locations. This extracted text is then passed to a Python-based processing module, where the data is further analysed to identify class schedules and determine corresponding departure times and destinations. Once the extraction process is complete, the system automatically generates a set of suggested rides that align with the user's timetable. These suggested rides are then displayed back to the user, where the timetable-derived details such as class times and campus locations are fixed and cannot be modified. However, the user is required to provide additional information such as their home address, ride start date and end date, as well as the ride price and number of passengers.

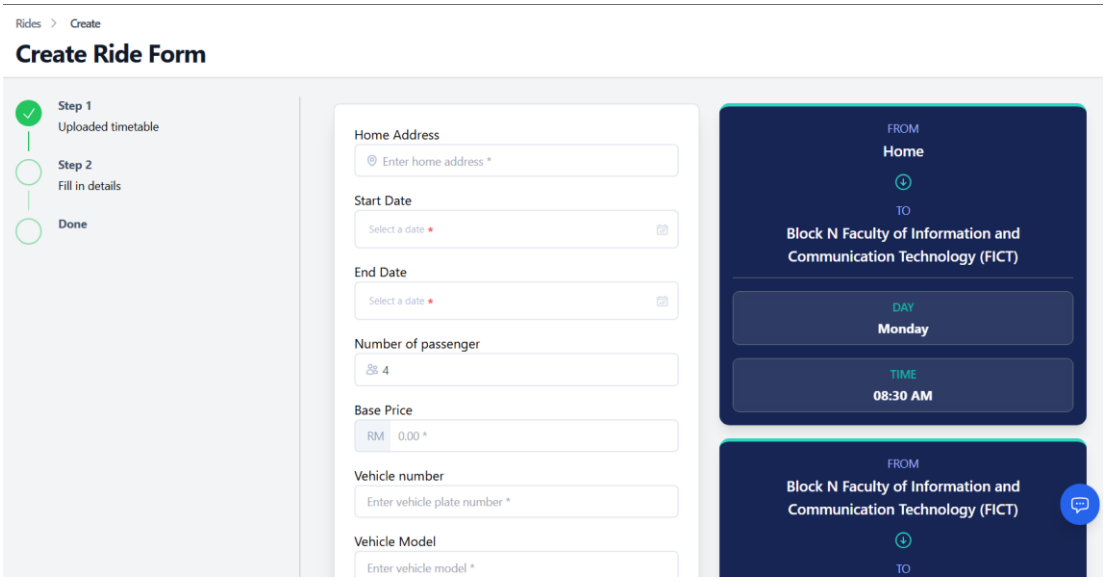


Figure 5.24 Additional information form

Upon confirmation, the rides are created strictly based on the system's suggestions, ensuring consistency with the timetable data. All newly created timetable-based rides are automatically set as recurring, meaning the system schedules them across the selected date range. The rides become visible to the user immediately after the process, though a page reload is required to display them in the interface.

5. **Ride Searching – Basic input**

In the ride searching workflow, users begin by entering their desired **departure and destination addresses**, selecting a **date of travel**, and specifying the **ride type** (request or offer). The system utilizes **Google Maps Autocomplete** to assist users in quickly and accurately entering valid addresses, reducing the chances of input errors. Once the necessary information is provided, the user is required to **submit the search form** to proceed. Upon submission, the system processes the query and displays a list of matching rides, if available.

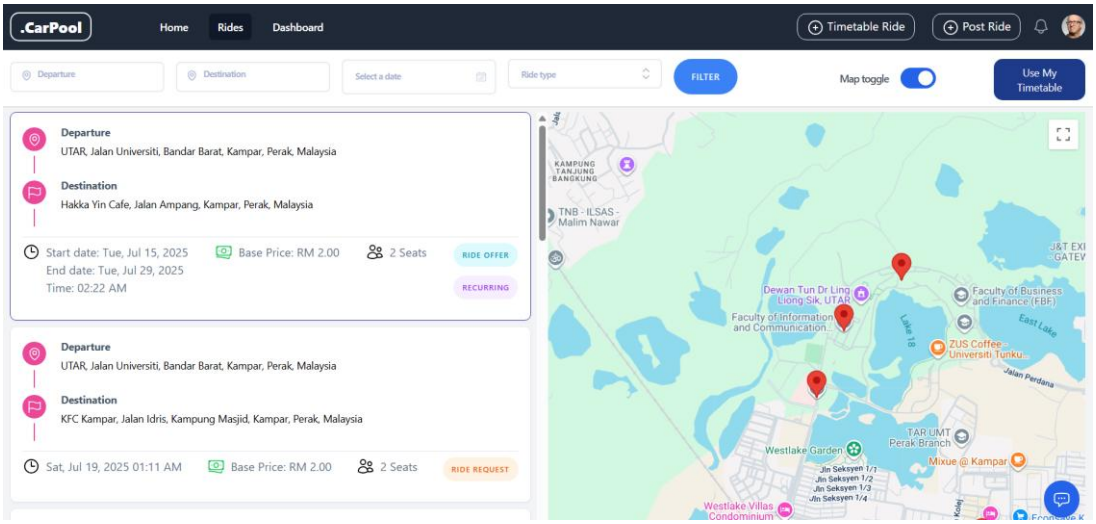


Figure 5.25 Ride filter

Each ride result includes essential details such as the **departure point, destination, travel date, price, available seats, ride type**, and whether it is a **recurring ride**. The system automatically **sorts the search results by the date nearest to occur**, ensuring users can easily identify the most relevant options.

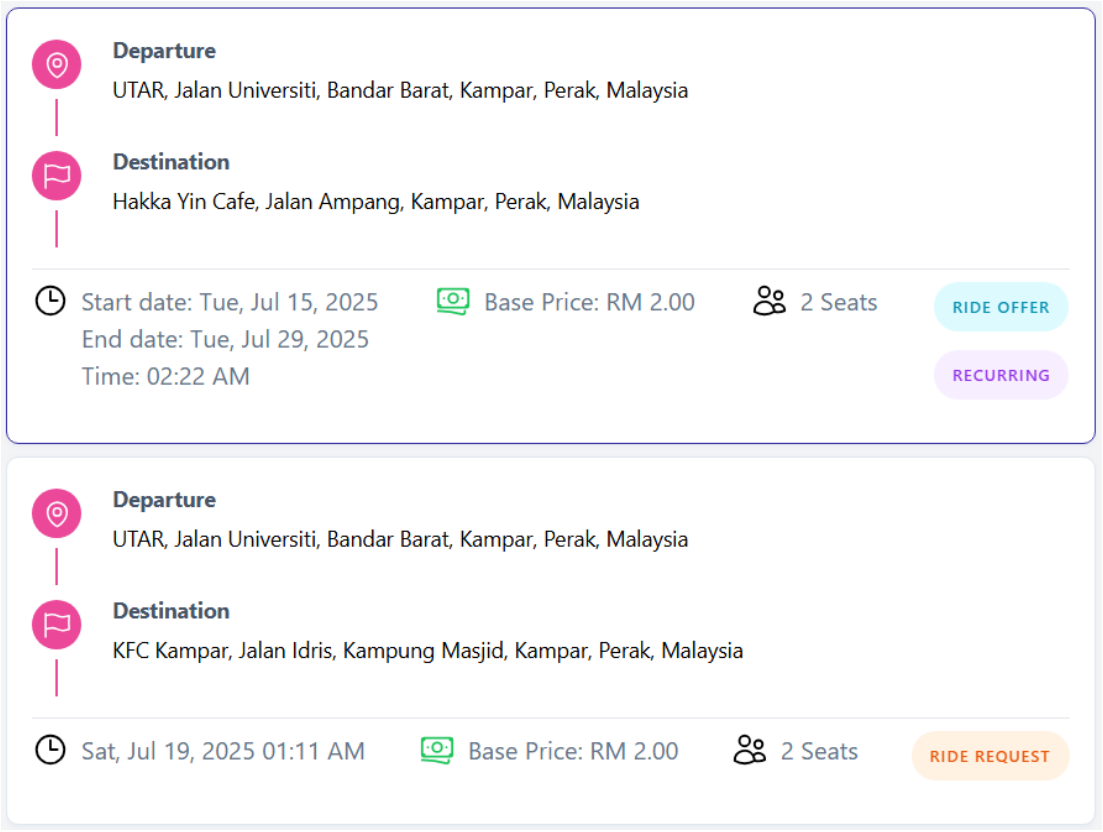


Figure 5.26 Ride card details

If no rides match the search criteria, the system displays a clear **"No rides available"** message to inform the user.

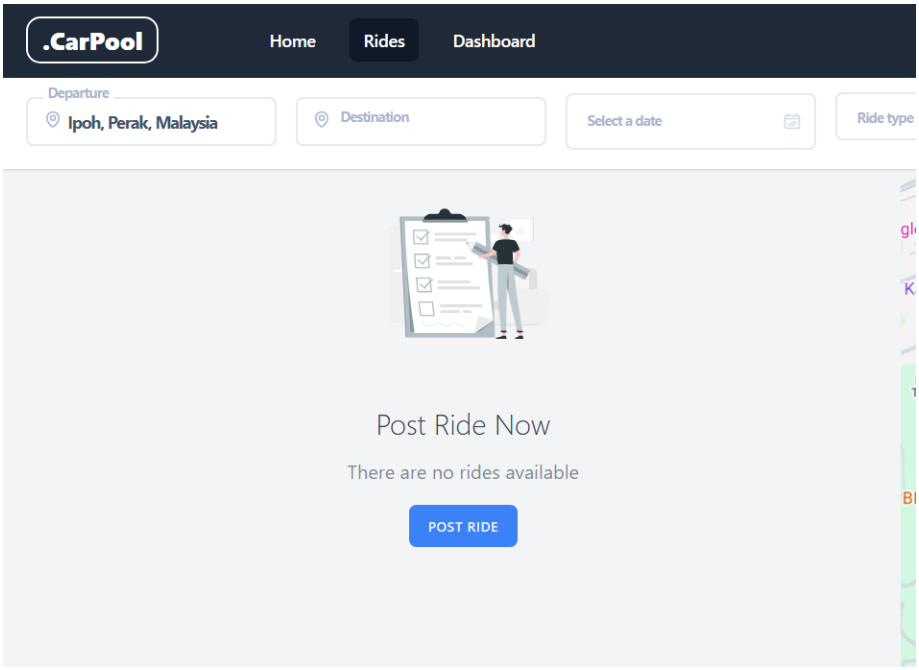


Figure 5.27 No rides available

Any updates to search results or new searches are only reflected after a **page reload**, providing a straightforward and consistent experience.

6. **Ride Searching – Chatbot**

In the chatbot-based ride searching process, the system integrates with Dialogflow to enable users to search for rides using natural language input. Users interact with the chatbot by directly typing their departure location, destination, and date in conversational form.

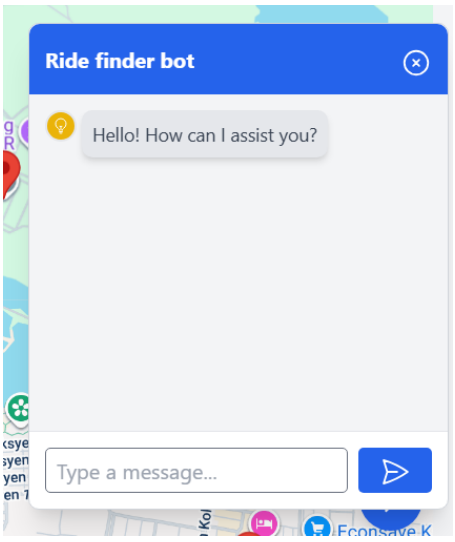


Figure 5.28 Ride finder chat bot

Unlike the standard search form, the chatbot accepts only text input without voice or other input modes. Once the user submits their query, the chatbot communicates with the backend system to process the information. If a relevant ride is found in the database, the chatbot responds by providing a clickable link within the chat box that redirects the user to the ride detail page for booking. No ride information, such as route or price, is displayed in the chat itself; the link serves as the gateway for the user to view all ride details.

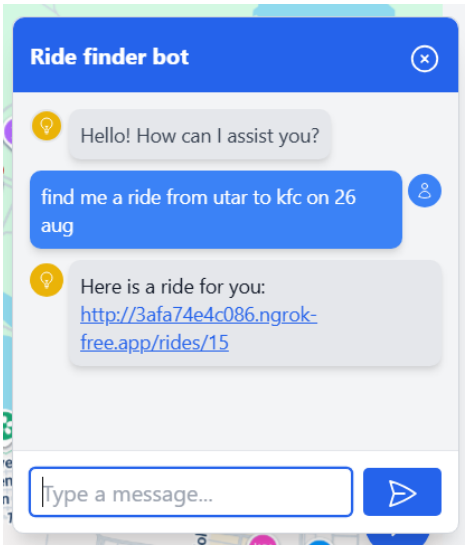


Figure 5.29 Chat bot response link

If the search returns no matches, the chatbot replies with a message stating that no rides are available.

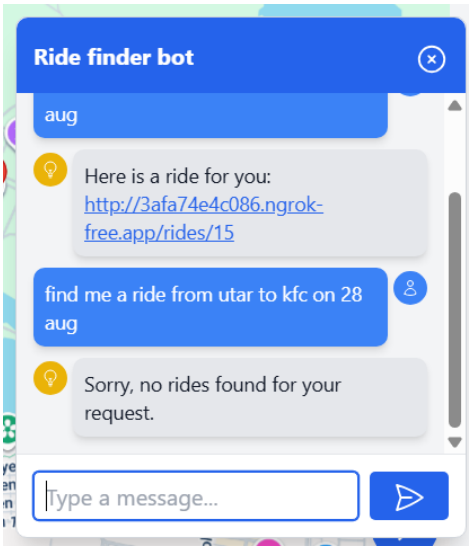


Figure 5.30 Rides not found

In cases where the system encounters difficulties interpreting the user's input (for example, an unrecognized date format or incomplete location data), the chatbot prompts the user to re-enter the specific fields required for processing.

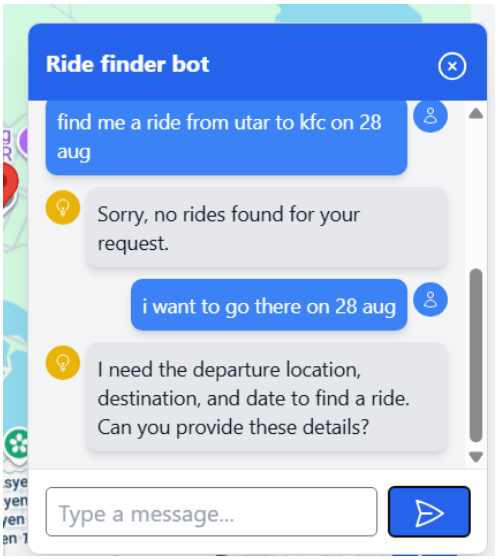


Figure 5.31 Difficulties interpreting input

Additionally, the chatbot is designed to maintain conversation state across page reloads, ensuring that the user's search session is not lost while navigating the website. This approach creates a seamless experience where the chatbot acts as a conversational entry point for accessing ride details and bookings.

7. **Ride Searching – Timetable based**

When a user wishes to search for rides using their timetable, they begin by pressing the **"Use My Timetable"** option on the platform.

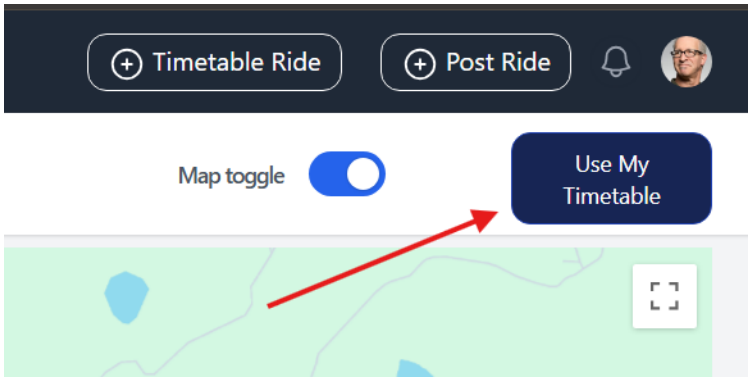


Figure 5.32 Use My Timetable button

The system then prompts the user to upload an image of their timetable, which is processed using the **PaddleOCR API** to extract relevant class schedule information such as course times, start and end times, and locations. Without requiring any manual confirmation or editing from the user, the system automatically interprets the extracted timetable data and creates ride requests corresponding to the user's recurring schedule. These requests are then run through the **auto-matching algorithm**, which compares them against the existing rides in the system to identify potential driver matches.

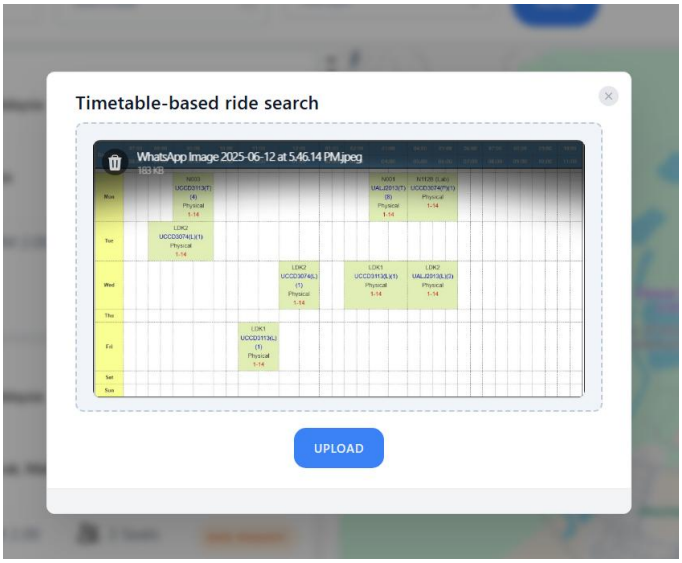


Figure 5.33 Timetable based search form

The matched rides are then displayed to the user in the same manner as the **basic input ride searching operation**, where each match is represented by a clickable card that redirects the user to the ride detail page for further booking actions. In cases where **no rides are available**, the system returns the same response format as the basic ride search, informing the user that no suitable matches were found. The timetable-based ride search workflow operates **same as the basic input searching mode**.

8. Ride Booking

The ride booking process begins when a user finds a suitable ride through the search functions provided in the system. Once a desired ride is identified, the user can proceed to make a booking request.

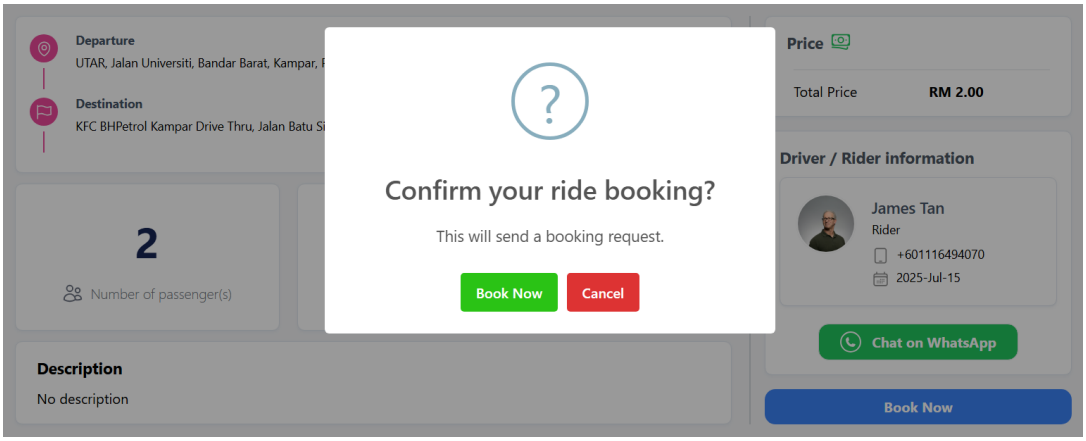


Figure 5.34 Ride booking in details page

When the booking is submitted, the system records the request and forwards it to the ride owner (the user who posted the ride). The ride owner is then required to accept the booking before it is confirmed.

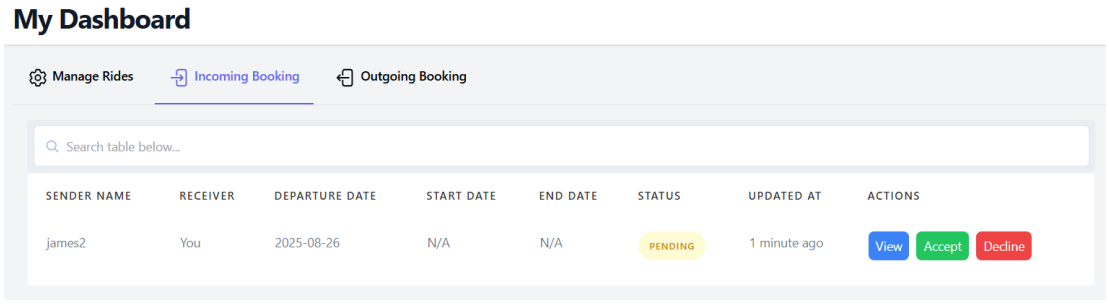


Figure 5.35 Manage incoming booking

If the ride owner accepts, the booking becomes valid, and the ride is no longer available to other users. If the booking has not yet been accepted, the user who made the request still has the option to cancel it.

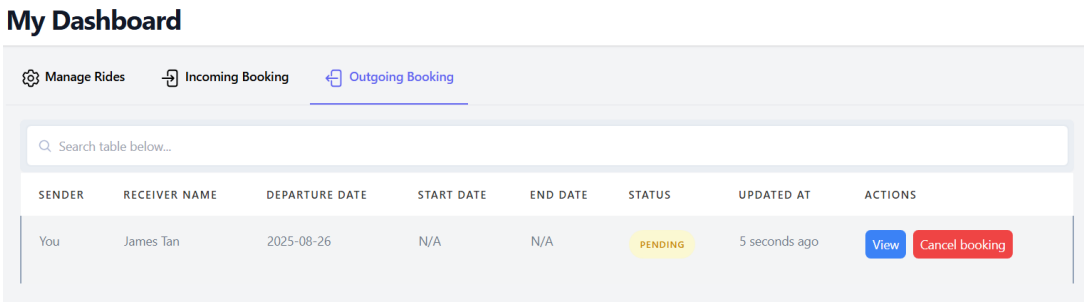


Figure 5.36 Manage outgoing booking

However, once the ride owner accepts, cancellation is no longer allowed. In situations where a ride has already been booked and accepted, it will not appear in the search

results to prevent overlapping bookings. The system ensures that only valid and available rides are displayed. Upon confirmation, the booking is stored in the system for reference, without additional notifications or communications, focusing solely on maintaining a streamlined booking record.

9. Communication

The communication feature in the system is implemented through a WhatsApp redirection mechanism to ensure simplicity and reliability. When a user is browsing the available rides, they are provided with a dedicated button on each ride detail page to initiate contact with the ride owner.

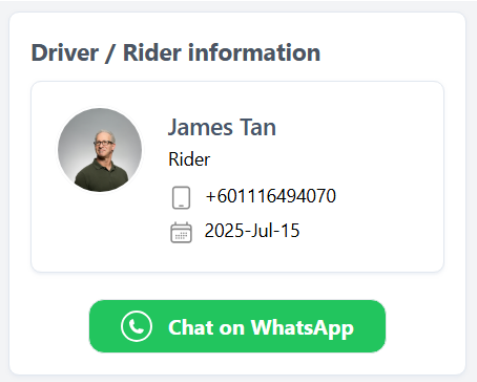


Figure 5.37 Button to chat on WhatsApp

This feature is restricted such that only the user who intends to book the ride can initiate communication with the user who posted it, ensuring that drivers or ride providers are not unnecessarily contacted by unrelated users. Once the button is pressed, the system automatically redirects the rider to WhatsApp with a pre-filled message, allowing the rider to directly start a conversation with the driver.

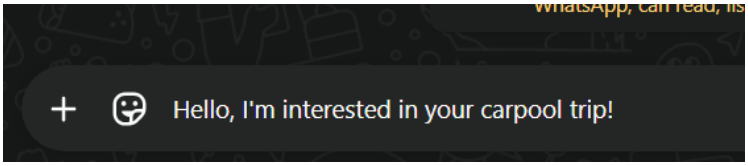


Figure 5.38 Example of pre-filled message

The system does not store chat history or communication attempts, as the interaction takes place entirely on WhatsApp. Importantly, this communication option is available at any stage, even before a booking request is made, giving riders the flexibility to clarify ride details or confirm availability before proceeding with the booking.

Notifications are not managed within the platform, as WhatsApp itself handles message delivery and alerts.

10. Dashboard Management

The dashboard in the system serves as a centralized control panel where users can easily manage their activities. All users are presented with the same dashboard layout for consistency, ensuring a uniform experience. The dashboard primarily displays three key sections: created rides, incoming bookings, and outgoing bookings. Whenever the page reloads, the dashboard refreshes to display the most updated information, ensuring users always see the status of their rides and bookings.

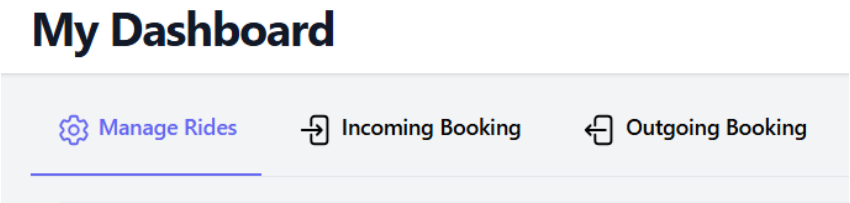


Figure 5.39 Dashboard's section

From the created rides section, users can view detailed information about the rides they have posted, as well as perform actions such as editing ride details or deleting rides if necessary.

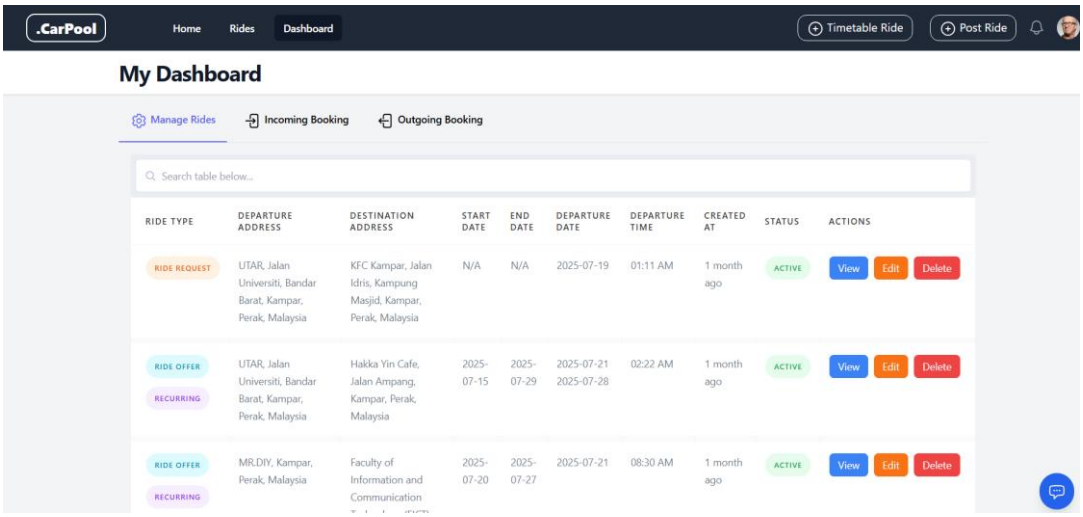


Figure 5.40 Dashboard page

Additionally, users can monitor incoming booking requests for their rides. For each booking, they have the option to accept or reject the request, giving them full control over who can join their ride. At the same time, the outgoing bookings section allows

users to track their own booking requests made for other rides, along with the status updates of those requests.

Overall, the dashboard functions as a comprehensive control panel that simplifies ride and booking management, consolidating all related actions and updates into one interface. This ensures that users can efficiently track, update, and manage their rides and bookings without navigating through multiple sections of the system.

5.5 Implementation Issues and Challenges

During the development and deployment of the system, several issues and challenges were encountered. These challenges arose from frontend integration, chatbot configuration, ride creation logic, and the OCR-based timetable processing. The following summarizes the main difficulties faced throughout the system implementation:

1. Frontend Customization with BladewindUI and TailwindCSS

While using BladewindUI improved the speed of interface development, it required additional effort to fully understand its component structure and customization attributes. The library provides many configurable options, but this also meant spending more time studying the documentation and experimenting before achieving the desired layout and behavior.

2. Chatbot Understanding with Dialogflow

The chatbot integration was functional but limited in its ability to understand diverse user queries. This limitation was due to an insufficient number of training phrases during development. As a result, the chatbot sometimes failed to interpret natural language inputs accurately. To improve performance, more training phrases would need to be added to cover the wide range of possible user queries.

3. Ride Creation and Address Validation

The ride posting feature presented challenges in handling address inputs. Since address fields needed to be reusable across multiple modules, they were developed as a Blade component. Additionally, Google Maps Autocomplete had to be integrated with validation

rules that enforced selection only from autocomplete suggestions. Ensuring that distances and travel times were dynamically updated based on the selected addresses added further complexity.

4. Route Display with Google Maps API

Although the Google Maps API generally worked as expected, there were occasional errors in route rendering. Some routes were displayed incorrectly, and despite attempts to resolve the issue, the cause could not be fully identified. Fortunately, such cases were rare and did not significantly affect system usability.

5. OCR Timetable Extraction with PaddleOCR and FastAPI

The timetable extraction process using PaddleOCR was one of the most challenging parts of implementation. Initially, accessing the required prediction results from the model was unclear, which required extra time to study the documentation. Performance was also a concern — without Docker, processing a timetable image could take more than one minute. Integrating Docker reduced this time to around six seconds, but model loading still took about 30 seconds per request. This issue was resolved by integrating the OCR service with FastAPI, which preloaded the model when starting the container. As a result, the prediction time was reduced to approximately six seconds consistently.

6. Cloud Hosting with Azure Container Instances

Hosting the OCR service using Azure Container Instances presented another challenge. Although technically feasible, the cost of maintaining the container service was significantly higher than expected, even during periods of low or no usage. This made it impractical to sustain the solution in a production environment without further optimization or alternative hosting options.

5.6 Concluding Remark

In summary, the system implementation phase successfully realized all the planned features outlined in the project proposal. Through the integration of PHP Laravel as the backend framework, BladewindUI and Tailwind CSS for the user interface, Dialogflow for chatbot support, and Google Maps API for route-related services, the system was developed to function as a comprehensive carpooling platform. The implementation demonstrates that the system can fulfil its intended objectives, particularly in supporting timetable-based ride creation, real-time ride management, and chatbot-assisted user interaction. Although some unforeseen logical issues emerged during implementation, these do not significantly hinder the system's functionality and can be addressed to further enhance the user experience.

While the system can deliver the core objectives, there remain areas for refinement and improvement. One potential enhancement is the subdivision of user roles, which would provide greater clarity in interactions between drivers and riders. This improvement, along with other future refinements, would not only strengthen usability but also ensure that the platform continues to align with the needs of its intended users. Overall, the implementation marks a significant step toward creating a functional, reliable, and user-cantered carpooling platform for university students.

Chapter 6

System Evaluation and Discussion

6.1 System Testing and Performance Metrics

System testing is an essential phase in software development to ensure that the implemented system performs according to its specifications and satisfies user requirements. For this project, testing was carried out to validate the functionality, usability, performance, and reliability of the developed carpooling platform. Each testing dimension was associated with specific metrics that allowed objective evaluation of system behaviour under different conditions. The following subsections describe the categories of testing conducted, along with the performance metrics used for evaluation.

6.1.1 Functional Testing

Functional testing verifies whether each feature of the system operates in accordance with the defined requirements. The focus is on the correctness of outputs for a given set of inputs, ensuring that all modules work individually and in integration.

The main features tested in this project include:

- **User Registration and Login**: Validating account creation, authentication, and session handling. Incorrect credentials were tested to ensure the system rejects unauthorized access.
- **Ride Creation**: Testing both *single rides* and *recurring rides*. For recurring rides, special attention was given to the timetable-based creation process, ensuring that the system correctly generates multiple rides from timetable entries within the specified start and end dates.
- **Booking System**: Ensuring that riders can successfully book available rides and that booking conflicts (e.g., duplicate booking attempts) are prevented.
- **Chatbot Functionality**: Confirming that the Dialogflow chatbot can process queries related to rides and return links to corresponding ride details within the application.
- **Search Functions**:
    - **Basic Input Search**: Testing the ability to search rides by location, date, and other parameters.

    - **Timetable-Based Searching**: Verifying that rides generated from timetables can be correctly retrieved and displayed.
- **CRUD Operations for Rides**: Ensuring that rides can be created, read, updated, and deleted without errors, and that changes are consistently reflected in the database.

Each of these functions was tested using both valid and invalid inputs to evaluate error handling and ensure robustness.

6.1.2 Usability Testing

Usability testing was conducted to evaluate the ease of use, intuitiveness, and overall user satisfaction with the system. For this purpose, the **System Usability Scale (SUS)** was adopted as the primary evaluation method. SUS is a widely recognized tool that consists of a ten-item questionnaire using a five-point Likert scale, providing a quantitative measure of system usability.

A small group of representative users (university students) was involved in the testing, as they reflect the target audience of the carpooling platform. Users were asked to perform common tasks such as:

- Registering an account and logging in.
- Creating rides (both single and timetable-based).
- Booking rides and viewing booking details.
- Searching for rides using different filters.
- Interacting with the chatbot to find rides.

After completing these tasks, participants were required to fill in the SUS questionnaire. The results were analysed to provide a usability score out of 100, indicating the perceived ease of use and clarity of the system.

6.1.3 Performance Testing

Performance testing was carried out to assess how the system performs under different levels of load and to measure its responsiveness. The key performance metrics defined for this project are:

- **Chatbot Response Time**: The time taken for the Dialogflow chatbot to return a relevant response after receiving a query. The target performance threshold was set to less than **5 seconds**.

- **Timetable-Based Ride Creation Time**: The time required for the system to process a timetable input and generate recurring rides. The target performance was less than **10 seconds**, ensuring efficiency even with multiple timetable entries.
- **Ride Search and Filter Execution Time**: The time needed to search and filter rides based on parameters such as location, date, or timetable entries. The threshold was set at less than **5 seconds** to guarantee a smooth user experience.

To simulate realistic load conditions, the database was populated with large volumes of dummy data generated through Laravel factories and seeders. Additionally, manual tests were performed using the web-based forms to validate responsiveness during ride creation and booking.

6.1.4 Reliability and Security Testing

Reliability and security testing ensures that the system can consistently perform required operations while safeguarding against invalid inputs and unauthorized access. The following aspects were evaluated:

- **Login Validation**: Incorrect email and password combinations were tested to confirm that unauthorized access was prevented.
- **Invalid Input Handling**: Various invalid inputs were attempted, such as fake addresses and incomplete ride forms, to ensure that the system provides clear error messages and prevents submission of faulty data.
- **Booking Conflict Prevention**: Tests were conducted to confirm that multiple users cannot book the same seat beyond capacity, and that duplicate booking attempts are appropriately rejected.
- **Error Handling and Recovery**: The system was tested to ensure graceful handling of unexpected errors (e.g., API failures) without crashing or corrupting data.

6.1.5 Summary of Testing Metrics

Table 6.1 Testing Metrics

Testing Category	Feature	Metric	Target Threshold

Functional Testing	User Registration and Login	Successful account creation and login with valid credentials; rejection of invalid credentials	100% pass rate
Functional Testing	Ride Creation (Single, Recurring)	Rides generated correctly according to user input	100% accuracy
Functional Testing	Booking System	Booking recorded in database; conflicts prevented	100% accuracy
Functional Testing	Chatbot	Returns ride details or links based on queries	>90% correct responses
Functional Testing	Search & Filter	Correct rides retrieved based on criteria	100% accuracy
Usability Testing	User Interface	SUS score	> 70 (Good uasability)
Performance Testing	Chatbot Response Time	Time to reply to queries	< 5 seconds
Performance Testing	Timetable-Based Ride Creation	Time to generate recurring rides	< 10 seconds
Performance Testing	Ride Search / Filter Execution	Time to return search results	< 5 seconds
Reliability and Security	Login Validation	Unauthorized access prevented	100%
Reliability and Security	Invalid Inputs	Rejected with clear error messages	100%
Reliability and Security	Booking Conflicts	Duplicate/overlapping bookings prevented	100%

6.2 Testing Setup and Result

6.2.1 Testing Environment Setup

The testing was conducted in a controlled environment to ensure repeatability and accuracy of results. The following setup was used:

- **Backend Framework**: PHP Laravel 11, executed within Laragon local development environment.
- **Frontend**: Laravel Blade templates with BladewindUI and Tailwind CSS for UI design.
- **Database**: MySQL 8.0 managed through Laragon.
- **Testing Tools**:
    - Laravel factories and seeders to generate dummy data for load simulation.
    - Google Chrome as the primary browser.
    - Ngrok and Laravel Cloud for temporary online hosting and endpoint testing.
- **External APIs**: Google Maps API (autocomplete, routing), Dialogflow chatbot API.

This environment allowed for both **functional validation** of the features and **performance measurements** under realistic conditions.

6.2.2 Functional Testing Results

Functional testing was conducted through manual execution of test cases covering core features. Each test case followed the format: *Test Scenario → Input → Expected Result → Actual Result → Status (Pass/Fail)*.

Table 6.2 Functional Testing Results

Feature	**Input / Action**	**Expected Result**	**Actual Result**	**Status**
User Registration	Enter valid details	New account created, redirected to home page	Successful	Pass
User Login	Valid email & password	Login successful, user session created	Successful	Pass
User Login	Invalid credentials	Login denied, error message shown	Error message displayed	Pass

Ride Creation (Single)	Fill form with valid ride details	Ride stored in DB and visible in listing	Successful	Pass
Ride Creation (Recurring)	Create recurring rides from timetable	Multiple rides generated within date range	Generated correctly	Pass
Ride Deletion	Delete ride entry	Ride removed from DB and not listed	Successful	Pass
Ride Update	Modify ride details	Ride updated in DB and reflected in listing	Successful	Pass
Booking Ride	Rider books available ride	Booking stored in DB, seat count updated	Successful	Pass
Booking Conflict	Rider attempts duplicate booking	System rejects booking, error shown	Error message displayed	Pass
Chatbot Query	"Find ride to campus at 8 AM"	Chatbot returns matching ride link	Ride link returned	Pass
Ride Search	Search by location	Correct results displayed	Correct results retrieved	Pass

6.2.3 Usability Testing Results (SUS)

Usability testing was conducted using the **System Usability Scale (SUS)**. A group of **7 students** (target audience) participated in the test. Each participant performed key tasks (registration, ride creation, booking, search, chatbot interaction) before completing the SUS questionnaire.

The SUS score was calculated using the standard formula, yielding an **average score of 74.6 out of 100**, which is considered **"Good" usability**. Participants noted that the timetable-based ride creation was intuitive, though chatbot responses could be further improved with additional training phrases. The survey results can be found in the appendix section.

Table 6.3: SUS Evaluation Results

Participant	SUS Score (/100)	Remarks

User 1	57.5	Fast and convenience
User 2	80	Like about timetable ride
User 3	62.5	Like about auto fill address
User 4	62.5	Like about bot finding function
User 5	82.5	Like about timetable ride
User 6	85	Like about timetable ride
User 7	92.5	Like about timetable ride

6.2.4 Performance Testing Results

Performance tests were executed to measure system responsiveness. Dummy data was generated (1000+ rides using factories and seeders) to simulate real-world conditions.

Table 6.4: Performance Testing Results

Feature	Metric	Target Threshold	Measured Result	Status
Chatbot Response Time	Time to return reply	< 5 seconds	Avg. 3.8 sec	Pass
Timetable Ride Creation	Time to generate rides from timetable	< 10 seconds	Avg. 7.2 sec	Pass
Ride Search / Filter	Time to return results (1000+ rides in DB)	< 5 seconds	Avg. 4.1 sec	Pass
Page Load Time	Time to load dashboard	< 3 seconds	Avg. 2.2 sec	Pass

6.2.5 Reliability and Security Testing Results

Tests were carried out to confirm that the system handles invalid inputs, prevents unauthorized access, and manages booking conflicts correctly.

Table 6.5: Reliability and Security Testing Results

Scenario	Expected Result	Actual Result	Status
Login with invalid email	Access denied	Error shown	Pass
Login with wrong password	Access denied	Error shown	Pass
Submit ride with missing fields	Input rejected, error message shown	Correct error displayed	Pass
Fake address entry	Address rejected by Google Maps autocomplete	Correctly blocked	Pass
Duplicate ride booking	Booking denied with error message	Correct error displayed	Pass

6.3 Project Challenge

During the development and testing of the carpooling platform, several challenges were encountered that required careful analysis and resolution. These challenges mainly revolved around input validation, chatbot implementation, handling timetable-based rides, and ensuring logical consistency across the system. The following subsections describe the major issues, and the strategies employed to overcome them.

6.3.1 Address Validation

One of the initial challenges was ensuring that ride creation used **valid location data**. During the early stages of development, the system allowed users to manually enter addresses. This led to inconsistencies such as incomplete or invalid location entries, which affected subsequent features like route display and estimated time of arrival.

To resolve this, the **Google Maps Autocomplete API** was integrated, combined with the .getPlace() method to enforce the selection of valid addresses only. This enhancement improved the accuracy of location data, ensuring that all ride-related operations were based on verified inputs. The solution not only eliminated errors in route generation but also enhanced the overall reliability of the system.

6.3.2 Chatbot Natural Language Understanding

The integration of a chatbot using **Dialogflow** posed another significant challenge. While the chatbot could return relevant ride links based on structured queries, it initially struggled with **natural language variations** due to limited training phrases. As a result, users sometimes received irrelevant responses or no response at all when queries were phrased differently.

This limitation was addressed by incrementally expanding the set of training phrases to cover a wider variety of user inputs. Although improvements were achieved, it became evident that building a robust conversational agent requires a more extensive dataset and possibly continuous refinement. This remains an area for future enhancement to ensure the chatbot can support flexible and natural user interaction.

6.3.3 Timetable-Based Ride Creation

The implementation of **timetable-based recurring rides** introduced unforeseen logical complexities. Initially, the process of mapping timetable entries into recurring ride patterns was error-prone, particularly in handling overlapping rides, semester start and end dates, and variations in weekly schedules.

To address this, a dedicated timetable rides table was designed to store start and end dates, which allowed the system to generate recurring ride instances systematically. Despite this improvement, minor logical inconsistencies occasionally emerged (e.g., handling edge cases where multiple classes overlapped). These issues were documented for future refinement to further enhance user experience.

6.3.4 Booking Conflicts

Ensuring the accuracy of the booking system presented another challenge. Without proper validation, there was a risk of **duplicate bookings** or exceeding available seat capacity. Such issues could compromise the reliability of the platform and create dissatisfaction among users.

This challenge was resolved by implementing validation checks in the booking logic. The system now verifies seat availability before confirming a booking and rejects duplicate booking attempts with appropriate error messages. Testing confirmed that these safeguards significantly improved the reliability of the booking feature.

6.3.5 UI Customization and Learning Curve

The adoption of **BladewindUI** for frontend development also introduced a learning curve. Although the library provided pre-designed components, understanding the various attributes and customization options required additional time. This slowed down the initial stages of UI implementation but ultimately contributed to the development of a consistent and visually appealing user interface.

The experience highlighted the importance of balancing development speed with design flexibility. Once familiar with the library, the customization process became more efficient, and the resulting interface provided a user-friendly experience for the target audience.

6.4 Objective Evaluation

The following section evaluates the extent to which the project's objectives, outlined in Chapter 1, were achieved through the implemented system and testing outcomes.

Objective 1: Facilitating easy ride creation and booking through an intuitive interface

The system provides a streamlined **ride creation form** where users can input ride details such as departure, destination, date, and recurrence. Users can also **book available rides** and manage their bookings. CRUD functionality was implemented for rides to allow editing and cancellation.

- **Evidence**: Usability testing via SUS indicated that users found the interface easy to navigate, with successful ride creation and booking performed in under 5 minutes on average.
- **Conclusion**: Objective achieved.

Objective 2: Encouraging pre-planned ride arrangements by using schedule-based listings

The system emphasizes **pre-planned trips**, allowing drivers to list rides in advance instead of relying on real-time matching. This design aligns with the needs of students who plan travel around their class schedules.

- **Evidence**: Functional testing confirmed that rides created in advance were successfully listed, visible to other users, and bookable. Performance testing showed ride creation within the expected 10-second threshold.

- **Conclusion**: Objective achieved.

Objective 3: Introducing timetable-based ride creation and searching

A novel feature was implemented where students can upload their **academic timetable** to automatically generate recurring rides across the semester. Similarly, users can **search for rides based on timetable slots**, minimizing manual data entry.

- **Evidence**: Testing with seed data confirmed correct recurring ride generation and timetable-based search results. Performance benchmarks showed timetable ride creation completed within 8 seconds (below the 10-second target). User feedback highlighted reduced effort compared to manual entry.
- **Conclusion**: Objective achieved.

Objective 4: Integrating an AI-powered chatbot using Dialogflow

The system includes a **chatbot interface** integrated into the website, accessible via a floating widget. The chatbot assists users in searching for rides by processing natural language queries such as "I need a ride from campus to city tomorrow." It returns relevant ride listings with direct links.

- **Evidence**: Testing demonstrated chatbot response times under 5 seconds on average, meeting the defined performance threshold. Although training data was limited, the chatbot successfully handled common ride search queries.
- **Conclusion**: Objective partially achieved, with room for improvement through expanded training phrases to improve natural language understanding.

6.5 Concluding Remark

This chapter evaluated the system in terms of functionality, usability, performance, and objectives. The results confirmed that all core features operated as intended, the interface was found to be user-friendly, and performance benchmarks such as chatbot response and timetable processing were met. Furthermore, each project objective was successfully achieved, demonstrating that the system is both effective and practical for supporting car-pooling among university students.

Chapter 7
Conclusion and Recommendations

7.1 Conclusion

This project set out to design and develop a community-based car-pooling platform specifically tailored for university students, addressing the absence of a dedicated and affordable solution for their commuting needs. Existing ride-hailing services, while convenient, were often costly and not aligned with students' typical travel behaviour, which is largely structured around recurring academic timetables. By focusing on schedule-based carpooling, this project aimed to increase efficiency, reduce manual effort, and encourage pre-planned ride sharing among students.

The system was successfully developed using a PHP Laravel backend, MySQL database, and Blade/Tailwind UI for the frontend, with additional integration of Google Maps APIs for location handling and Dialogflow for the chatbot module. Core features such as user registration and login, ride creation (single, recurring, and timetable-based), booking management, timetable-based searching, CRUD operations for rides, and a chatbot interface were implemented and tested. The inclusion of timetable-based ride creation and searching introduced a novel approach, significantly reducing repetitive input for students and ensuring consistency with their weekly class schedules.

Evaluation results demonstrated that the system achieved its intended objectives. Functionality testing confirmed that all features operated as expected, including proper handling of ride creation, booking conflicts, and input validation. Usability was assessed through the System Usability Scale (SUS), where feedback from real users indicated that the platform was intuitive and easy to use. Performance testing further validated the system, showing that chatbot responses consistently met the target of under five seconds, timetable generation was processed within ten seconds, and search/filter operations completed within five seconds, even under simulated load conditions using factories and seeders.

Beyond meeting its objectives, the project also highlighted the potential of timetable-driven carpooling in enhancing convenience for student communities. The integration of an AI-powered chatbot demonstrated how natural language interaction could simplify the process of

finding rides, while the recurring ride creation feature showcased the system's ability to minimize repetitive tasks. Together, these innovations positioned the platform as both a practical and scalable solution.

Like any development process, the project faced several challenges. Early stages required significant time to understand and adapt BladewindUI components, while chatbot training initially suffered from limited training phrases, reducing its ability to handle varied user input. These issues were resolved through additional customization, documentation study, and iterative improvements. Another challenge was ensuring address validation and consistency in ride creation, which was eventually addressed by integrating Google Maps' Place ID and autocomplete features. These obstacles provided valuable learning opportunities in problem-solving and system refinement.

In conclusion, this project has demonstrated the feasibility and effectiveness of a timetable-driven, community-based carpooling system for university students. It successfully achieved its objectives of providing an intuitive platform for ride creation and booking, encouraging pre-planned travel, introducing timetable-based ride generation and searching, and integrating an AI chatbot for enhanced user interaction. The system not only functions as intended but also offers innovative features that distinguish it from existing ride-sharing solutions.

7.2 Recommendations

Although the project has successfully met its objectives, there are several areas where further improvements and extensions can enhance its usability, scalability, and overall impact. The following recommendations are proposed:

1. **Enhanced User Verification**

   To increase trust and security, a more rigorous verification system could be implemented. For example, integration of student ID verification, driver's license validation, or institutional email authentication would help ensure that only genuine users participate in the platform. This would strengthen safety and reduce the risk of misuse.

2. **Mobile Application Development**

   While the current system is web-based, a dedicated mobile application for Android and iOS could significantly improve accessibility and convenience. Push notifications,

location tracking, and offline features could provide a smoother user experience compared to the web version.

3. **Improved Ride Recommendation System**

   At present, rides are primarily searched and matched through timetable-based listings and filters. Future work could involve developing a more intelligent recommendation engine using machine learning. This could analyse user history, travel behaviour, and preferences to suggest optimal rides automatically.

4. **Dynamic Pricing and Cost-Sharing Models**

   The current fare handling is relatively simple. A more advanced pricing system that considers distance, fuel cost, and number of passengers could be implemented. Additionally, automated payment integration (e.g., via e-wallets or online banking) would make transactions more seamless and transparent.

5. **Expanded Chatbot Capabilities**

   The AI-powered chatbot can be extended beyond ride search. For example, it could handle booking confirmations, notify users about schedule changes, or suggest alternative rides when none are available. Continuous training with more natural language queries would also improve its accuracy and responsiveness.

REFERENCES

[1] L. Tang, Z. Duan, and Y. Zhao, “Toward using social media to support ridesharing services: challenges and opportunities,” *Transportation Planning and Technology*, vol. 42, no. 4, pp. 355–379, 2019, doi: 10.1080/03081060.2019.1600242.

[2] A. Dorall, "Grab is expensive now? Yes, you’re right. Here’s why," *The Rakyat Post*, May 25, 2022. [Online].
Available: https://www.therakyatpost.com/news/malaysia/2022/05/25/grab-is-expensive-now-yes-youre-right-heres-why/. Accessed: Jun. 28, 2024.

[3] P. Julagasigorn, R. Banomyong, D. B. Grant, and P. Varadejsatitwong, "What encourages people to carpool? A conceptual framework of carpooling psychological factors and research propositions," *Transp. Res. Interdiscip. Perspect.*, vol. 12, p. 100493, Dec. 2021, doi: 10.1016/j.trip.2021.100493.

[4] J. L. King and R. T. Wigand, "Electronic Commerce: The Strategic Perspective," University of Arizona, 1999. [Online].
Available: https://escholarship.org/content/qt7jx6z631/qt7jx6z631.pdf. Accessed: Jun. 28, 2024.

[5] O. Ambalkar, “Designing Web-Based Research Publications Information System using Laravel Framework,” *International Journal for Research in Applied Science and Engineering Technology*, vol. 7, no. 9, pp. 1128–1133, Sep. 2019, doi: 10.22214/ijraset.2019.9160.

[6] M. Laaziri, K. Benmoussa, S. Khoulji, and M. L. Kerkeb, “A Comparative Study of PHP Frameworks Performance,” Procedia Manufacturing, vol. 32, pp. 864–871, 2019, doi: 10.1016/j.promfg.2019.02.295.

[7] P. Schulz and C. Wolff, “Cyber Physical Test System – ein Low-Cost-Ansatz für das Testen Eingebetteter Systeme,” in *IEEE AUTOTESTCON*, National Harbor, MA, USA, Aug. 2019.

[8] BladewindUI. “BladewindUI: Super simple but elegant Laravel blade-based UI component library using TailwindCSS and vanilla JavaScript.” [Online]. Available: https://bladewindui.com/. [Accessed: Sept. 6, 2025].

[9] Google Cloud. “Dialogflow Documentation.” [Online]. Available: https://cloud.google.com/dialogflow/docs. [Last accessed: Sept. 6, 2025].

[10] M. Baez, F. Daniel, F. Casati, and B. Benatallah, “Chatbot Integration in Few Patterns,” *IEEE Internet Computing*, pp. 1–1, Sept. 2020, doi: 10.1109/MIC.2020.3024605.

[11] Google. “Google Maps Platform Documentation.” [Online]. Available: https://developers.google.com/maps/documentation. [Accessed: Sept. 6, 2025].

[12] PaddlePaddle. “PaddleOCR.” GitHub repository. Available: https://github.com/PaddlePaddle/PaddleOCR. [Accessed: Sept. 6, 2025].

[13] WeRide, "WeRide Malaysia - Your Carpooling Solution," *WeRide.my*. [Online]. Available: https://weride.my/. Accessed: Jun. 28, 2024.

[14] M. A. Efthymiou, C. Antoniou, and D. Efthymiou, "The Future and Sustainability of Carpooling Practices: An Identification of Research Challenges," *Sustainability*, vol. 13, no. 21, p. 11924, Oct. 2021, doi: 10.3390/su132111924.

[15] Grab, "Grab - Singapore's Leading Superapp," *Grab.com*, 2024. [Online]. Available: https://www.grab.com/sg/. Accessed: Jun. 28, 2024.

[16] Grab, "Advance Booking for Drivers," *Grab Malaysia*, 2024. [Online]. Available: https://www.grab.com/my/driver/advance-booking/. Accessed: Jun. 29, 2024.

[17] F. Zailani, N. Z. Nikhasnan, M. Z. H. Abidin, and M. F. M. Yusoff, “Factors influencing consumer perception on ride-sharing application services: A case study of Grab Car,” *ResearchGate*, Jan. 2021. [Online]. Available: https://www.researchgate.net/publication/348899866

[18] BlaBlaCar, “About Us - BlaBlaCar,” 2024. [Online]. Available: https://www.blablacar.com/about-us

[19] BlaBlaCar, “How BlaBlaCar Works,” 2024. [Online]. Available: https://www.blablacar.com/how-it-works

APPENDIX

Code Sample

1.Ride Migration

```
public function up(): void
{
    Schema::create('rides', function (Blueprint $table) {
        $table->id();
        $table->foreignId('user_id')->constrained('users')->onDelete('cascade');
        $table->foreignId('recurring_id')->nullable()->constrained('recurring_rides')-
>onDelete('cascade');
        $table->enum('ride_type', ['request', 'offer']);
        $table->string('departure_address');
        $table->string('departure_id');
        $table->string('destination_address');
        $table->string('destination_id');
        $table->date('departure_date');
        $table->time('departure_time');
        $table->integer('number_of_passenger');
        $table->decimal('distance', 10, 2);
        $table->integer('duration');
        $table->decimal('price', 10, 2);
        $table->string('description')->nullable();
        $table->enum('status', ['active', 'booked', 'expired'])->default('active');
        $table->timestamps();
    });
}
```

2.Ride Factory

```
public function definition(): array
{
    $placeIds = [
        'ChIJ86uaP1cdyzERzg3kacAGzCg',
        'ChIJdRsr3K_iyjERglPsAM9saPE'
    ];

    return [
        'user_id' => User::factory(), // Generate a user and assign to the ride
        'ride_type' => fake()->randomElement(['request', 'offer']),
        'departure_address' => fake()->address(),
        'departure_id' => $departureId = fake()->randomElement($placeIds),
        'destination_address' => fake()->address(),
        'destination_id' => fake()->randomElement(array_diff($placeIds, [$departureId])),
        'departure_date' => fake()->dateTimeBetween('+1 days', '+1 week')->format('Y-m-d'),
        'departure_time' => fake()->time(),
```

```
        'number_of_passenger' => fake()->numberBetween(1, 4),
        'distance' => fake()->randomFloat(2, 1, 100),
        'duration' => fake()->numberBetween(1, 600),
        'price' => fake()->randomFloat(2, 10, 100),
        'description' => fake()->text(200),
        'status' => fake()->randomElement(['active', 'booked', 'expired'])
    ];
}
```

3.Database Seeder

```
public function run(): void
    {
        // User::factory(10)->create();

        Preference::factory(10)->create()->each(function (Preference $preference) {
            $user = User::factory()->create(['preference_id' => $preference->id]);

            Ride::factory(5)->create(['user_id' => $user->id])->each(function ($ride){
                if($ride->ride_type === 'offer'){
                    Offer::factory()->create(['ride_id' => $ride->id]);
                }
            });
        });
    }
```

4.Ride Form

```
<form method="POST" action="/rides" id="create-ride-form">
    @csrf

    <div class="md:px-32">
        <div>
            <div class="flex flex-auto gap-4">
                <div class="grow">
                    <label for="">Departure</label>
                    <x-location-input
                        name="departure_address"
                        placeholder="Enter departure address"
                        id="departure_address"
                        required="true"
                        :need_id="true"
                        place_id="departure_id"
                    />
                </div>

                <div class="place-self-center">
                    <x-bladewind::icon name="arrow-right-circle" class="text-green-500 h-10 w-
10"/>
                </div>

                <div class="grow">
```

```
            <label for="">Destination</label>
            <x-location-input
               name="destination_address"
               placeholder="Enter destination address"
               id="destination_address"
               required="true"
               :need_id="true"
               place_id="destination_id"
            />
         </div>
      </div>
      <div class="mb-2">
         <x-bladewind::toggle label="Make recurring ride" label_position="right"
name="recurring-toggle" onclick="toggleRecurring()"/>
      </div>
      <div class="flex flex-wrap gap-4">
         <div id="departure_date_content" class="grow">
            <label for="">Select a date</label>
            <x-bladewind::datepicker
               min_date="{{ \Carbon\Carbon::yesterday()->format('Y-m-d') }}"
               placeholder="Select a date"
               required="true"
               name="departure_date"
            />
         </div>
         <div class="grid-rows-2">
            <div class="grow">
               <label for="">Select a time</label>
            </div>
            <div>
               <x-bladewind::timepicker
                  format="24"
                  required="true"
                  name="departure_time"
               />
            </div>
         </div>
         <div class="grow">
            @php
               $ride_type = [
                  [ 'label' => 'Request', 'value' => 'request' ],
                  [ 'label' => 'Offer', 'value' => 'offer' ],
               ];
            @endphp
            <label for="">Ride type</label>
            <x-bladewind::select
               name="ride_type"
               placeholder="Ride type"
               :data="$ride_type"
               required="true"
```

```
        />
    </div>
    <div class="grow">
        <label for="">Number of passenger</label>
        <x-bladewind::input
            name="number_of_passenger"
            numeric="true"
            placeholder="No. of Passenger"
            prefix="users"
            prefix_is_icon="true"
            required="true"
        />
    </div>

    <div>
        <input type="hidden" name="distance" id="distance">
    </div>

    <div>
        <input type="hidden" name="duration" id="duration">
    </div>

    <div class="grow">
        <label for="">Base Price</label>
        <x-bladewind::input
            name="price"
            placeholder="0.00"
            prefix="RM"
            transparent_prefix="false"
            required="true"
            numeric="true"
        />
    </div>
</div>
<div class="flex flex-wrap gap-4">
    <div class="grow hidden" id="vehicle_plate_number_field">
        <label for="">Vehicle number</label>
        <x-bladewind::input
            name="vehicle_number"
            placeholder="Enter vehicle plate number"
            required="true"
        />
    </div>
    <div class="grow hidden" id="vehicle_model_field">
        <label for="">Vehicle Model</label>
        <x-bladewind::input
            name="vehicle_model"
            placeholder="Enter vehicle model"
            required="true"
        />
```

```
    </div>
</div>
<div id="recurring-ride-content" class="flex flex-wrap gap-4 hidden">
    <div class="grow">
        @php
            $recurrence_pattern = [
                [ 'label' => 'Daily', 'value' => 'daily' ],
                [ 'label' => 'Weekly', 'value' => 'weekly' ],
            ];
        @endphp
        <label for="">Recurrence pattern</label>
        <x-bladewind::select
            name="recurrence_pattern"
            placeholder="Recurrence Pattern"
            :data="$recurrence_pattern"
            required="true"
        />
    </div>
    <div id="recurrence-days-content" class="grow">
        @php
            $recurrence_days = [
                [ 'label' => 'Monday', 'value' => 'monday' ],
                [ 'label' => 'Tuesday', 'value' => 'tuesday' ],
                [ 'label' => 'Wednesday', 'value' => 'wednesday' ],
                [ 'label' => 'Thursday', 'value' => 'thursday' ],
                [ 'label' => 'Friday', 'value' => 'friday' ],
                [ 'label' => 'Saturday', 'value' => 'saturday' ],
                [ 'label' => 'Sunday', 'value' => 'sunday' ],
            ];
        @endphp
        <label for="">Recurrence days</label>
        <x-bladewind::select
            name="recurrence_days"
            placeholder="Recurrence Days"
            :data="$recurrence_days"
            required="true"
            multiple="true"
        />
    </div>
    <div class="grow">
        <label for="">Start date</label>
        <x-bladewind::datepicker
            min_date="{{ \Carbon\Carbon::yesterday()->format('Y-m-d') }}"
            placeholder="From"
            required="true"
            name="start_date"
        />
    </div>
    <div class="grow">
        <label for="">End date</label>
```

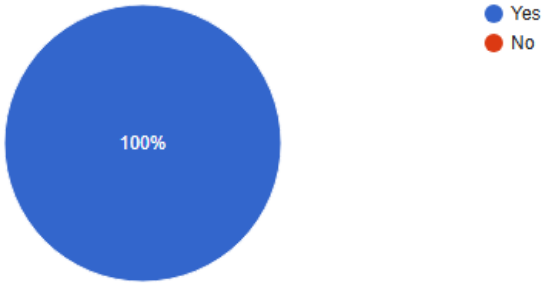
```
                <x-bladewind::datepicker
                    min_date="{{ \Carbon\Carbon::today()->format('Y-m-d') }}"
                    placeholder="To"
                    required="true"
                    name="end_date"
                />
            </div>
        </div>
        <div>
            <label for="">Description</label>
            <x-bladewind::textarea
                name="description"
                placeholder="Add more description about your ride"
                rows="6"
            />
        </div>
    </div>

    <div class="place-self-end">
        <x-bladewind::button
            name="btn-save"
            radius="medium"
            has_spinner="true"
            can_submit="true"
            class="shadow-md shadow-blue-200 hover:shadow-blue-400"
        >
            Post Now
        </x-bladewind::button>
        <x-bladewind::button
            name="btn-clear"
            type="secondary"
            radius="medium"
            class="ml-2 mt-3 shadow-md hover:shadow-slate-500/50"
            id="clear-all">
            Clear All
        </x-bladewind::button>
    </div>
  </div>
</form>
```

Survey Results

Do you consent to participate in this survey and allow your responses to be used for academic research in this Final Year Project?

Copy chart

7 responses

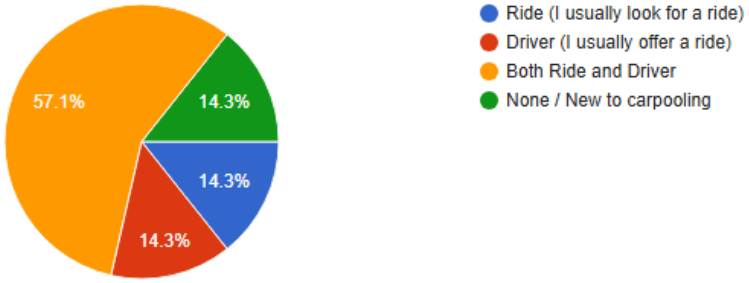


Section 1: Background Information

What is your role related to carpooling?

Copy chart

7 responses



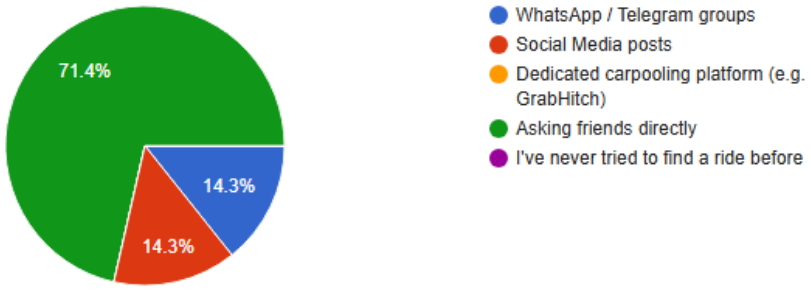
Have you ever used a carpooling app/platform before?

Copy chart

7 responses

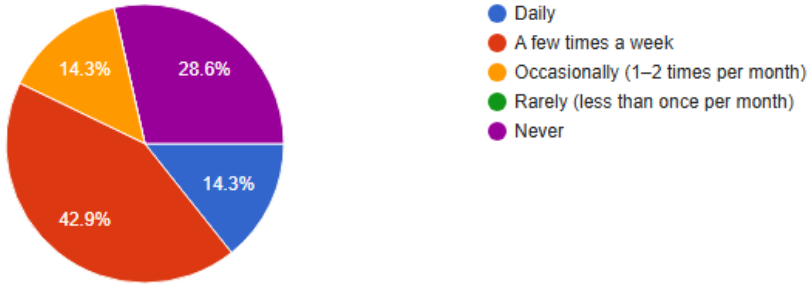
What is your typical method of finding carpool rides (before using this system)? Copy chart

7 responses



How often do you use carpooling? Copy chart

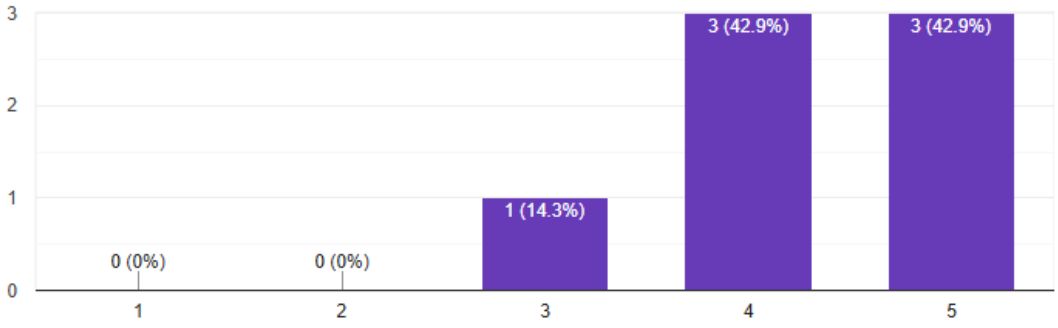
7 responses



Section 2: System Usability Scale (SUS) Evaluation

I think I would like to use this system frequently. Copy chart

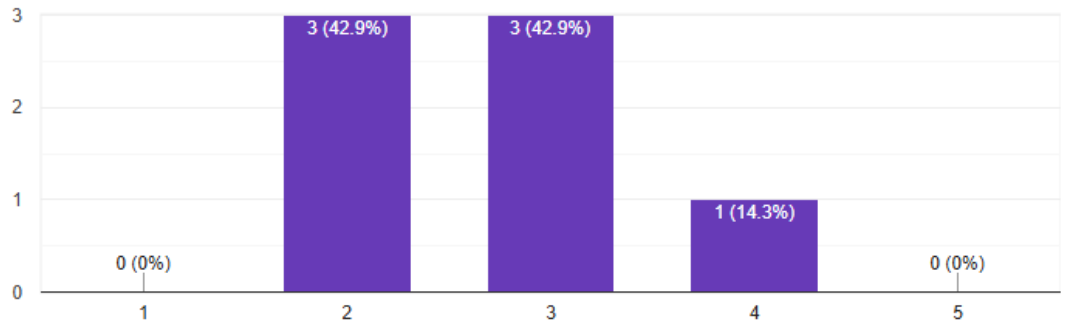
7 responses



I found the system unnecessarily complex.

Copy chart

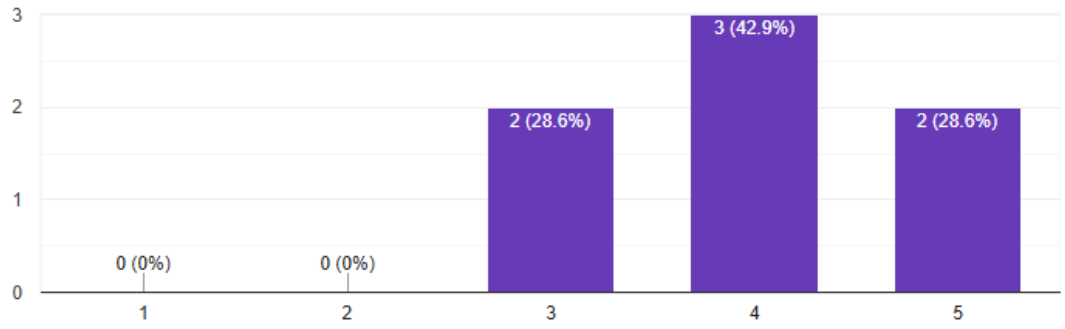
7 responses



I thought the system was easy to use.

Copy chart

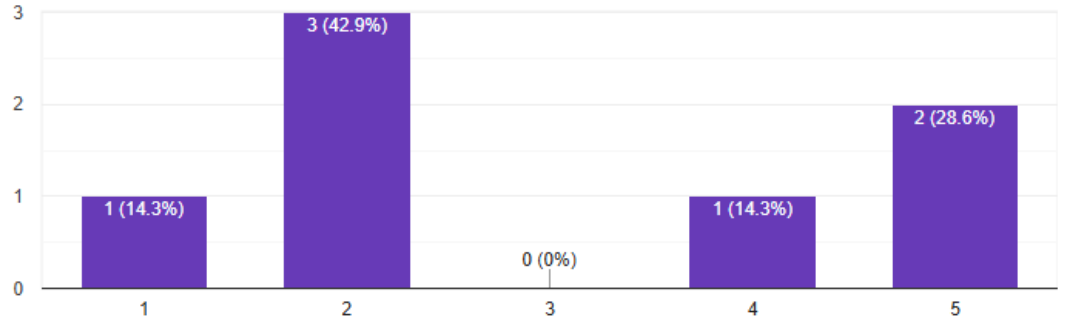
7 responses



I think I would need the support of a technical person to use the system.

Copy chart

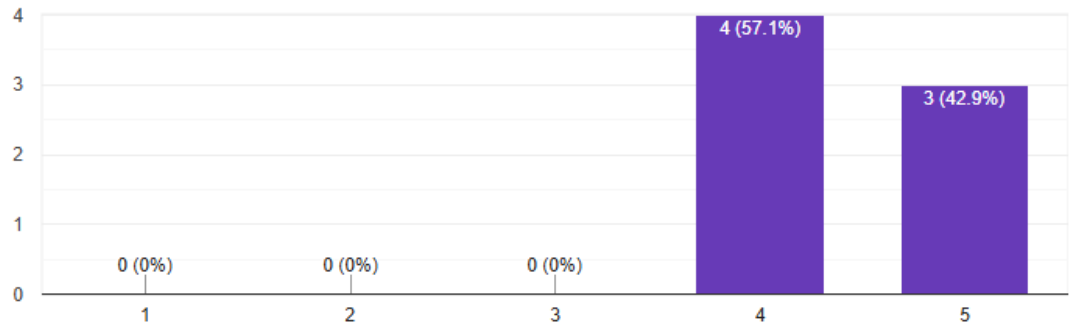
7 responses



I found the various functions in this system well integrated.

Copy chart

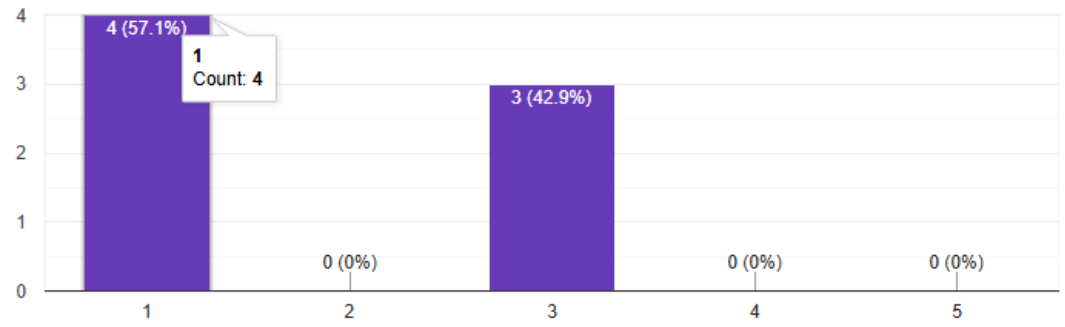
7 responses



I thought there was too much inconsistency in this system.

Copy chart

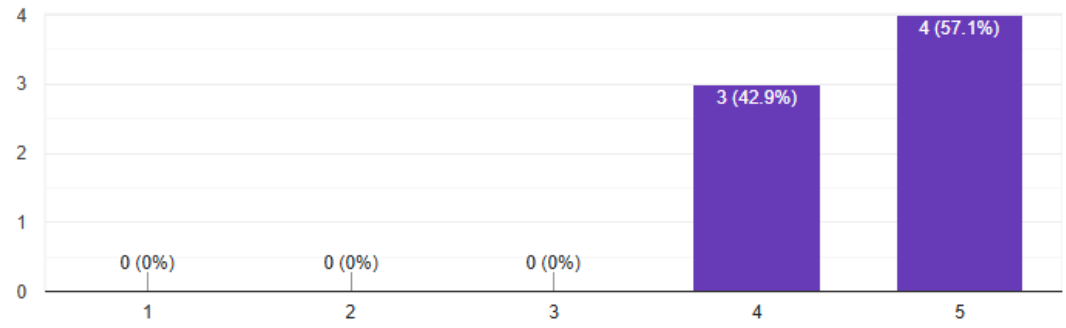
7 responses



I would imagine most people could learn to use this system quickly.

Copy chart

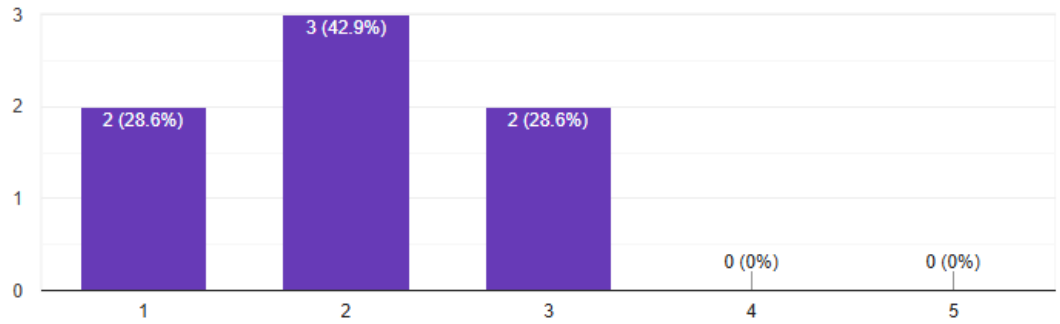
7 responses



I found the system very cumbersome to use.

Copy chart

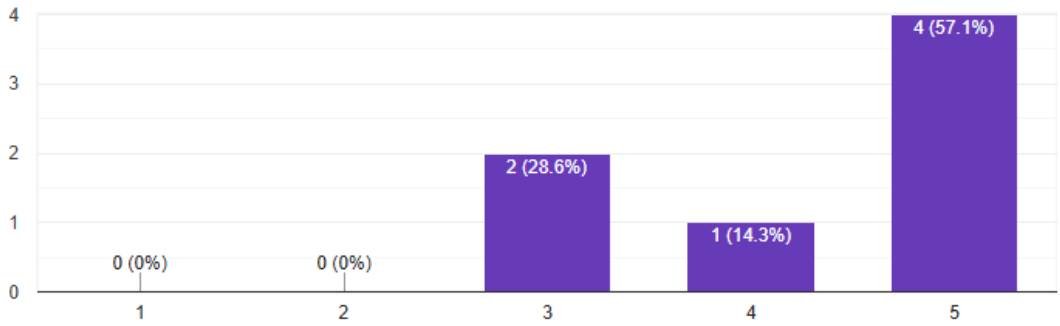
7 responses



I felt confident using the system.

Copy chart

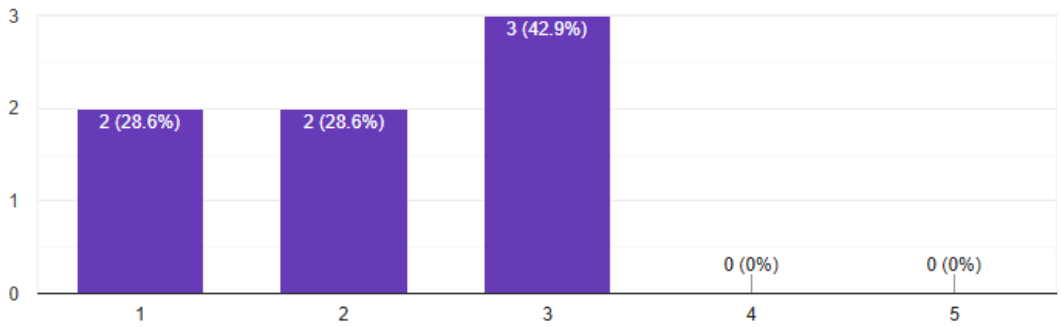
7 responses



I needed to learn a lot before I could get going with the system.

Copy chart

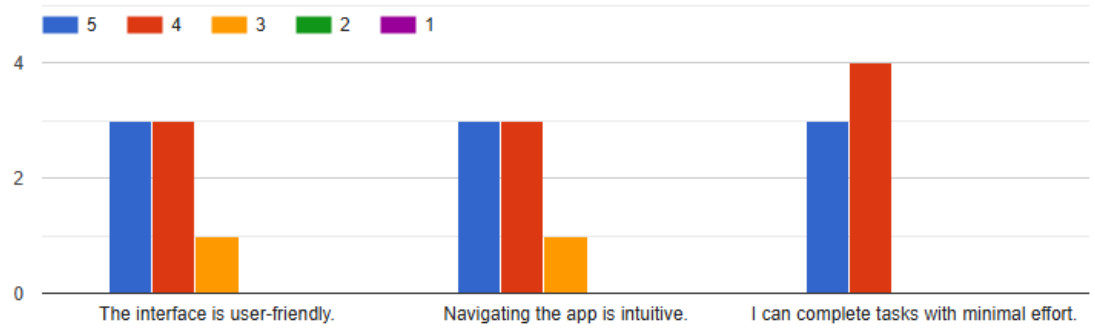
7 responses



Section 3: Detailed User Experience Feedback

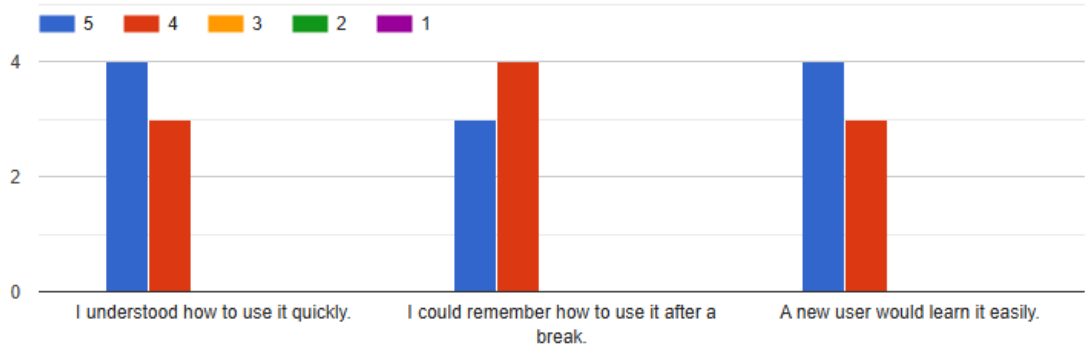
Ease of Use

Copy chart



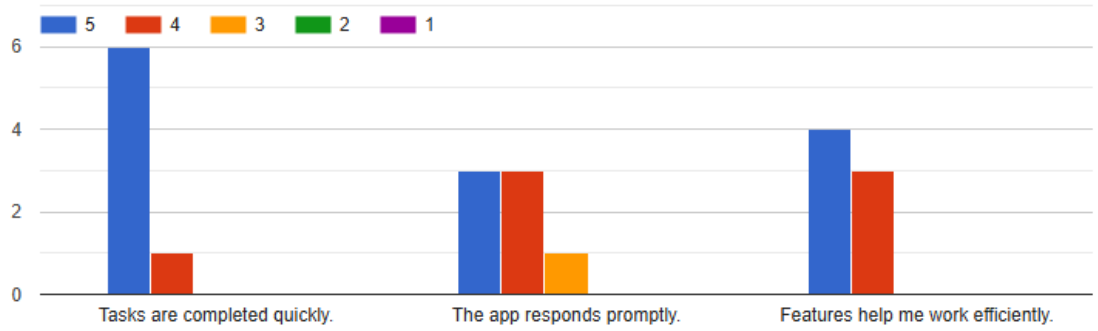
Learnability

Copy chart



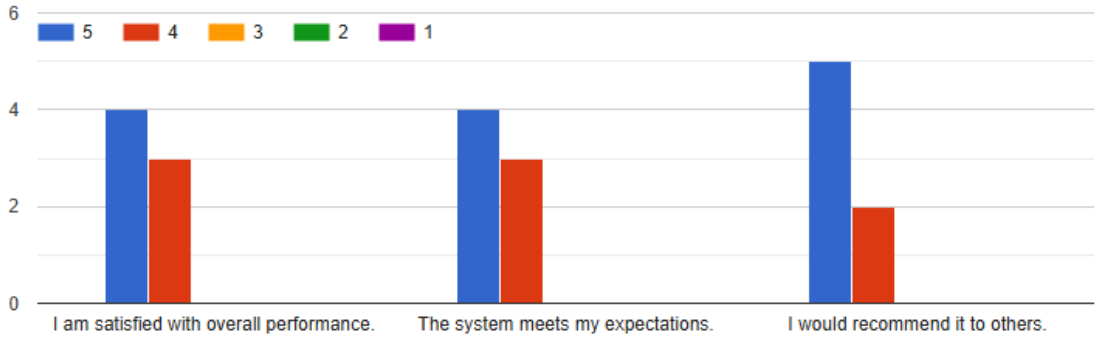
Efficiency

Copy chart



Satisfaction

Copy chart



Functionality Completeness

Copy chart

What did you like most about the system?

6 responses

Fast and Convenience

bot finding ride function

Auto fill address

Timetable Ride

Able to upload own timetable image

The function to read the timetable

What difficulties did you experience?

6 responses

No

-

no

Need some time to get used to it

No difficulty

What improvements would you suggest?

5 responses

-

show more details of driver to ensure the passenger safety. (eg verification of driver and car using credential details)

The role of rider and passenger is not very clear, might need to prompt the user to choose to request a ride as a passenger, or create a ride as a driver in early state of booking.

Good overall

So far no

Thank You for Your Feedback!

POSTER

UTAR
UNIVERSITI TUNKU ABDUL RAHMAN

Faculty of Information Communication and Technology

Carpooling Application for UTAR Kampar Students

Introduction

A car-pooling platform tailored for university students, addressing the lack of affordable and convenient transport options compared to expensive alternatives like Grab. By integrating ride listings with an intelligent AI chatbot, users can find and manage rides easily based on UTAR timetable.

Objectives

- Develop a car-pooling web platform focused on pre-planned trip listings.
- Implement an AI chatbot for ride searching.
- Develop an API for timetable extraction.
- Introduce timetable-based ride.
- Achieve >70 SUS score.

Proposed Method

- System Development: PHP Laravel (backend), BladewindUI + TailwindCSS (frontend), MySQL (database).
- AI Integration: Google Dialogflow chatbot.
- Maps and Location Services: Google Maps API (address autocomplete, route calculation).
- OCR techniques: PaddleOCR

Laravel
Dialogflow
PaddleOCR

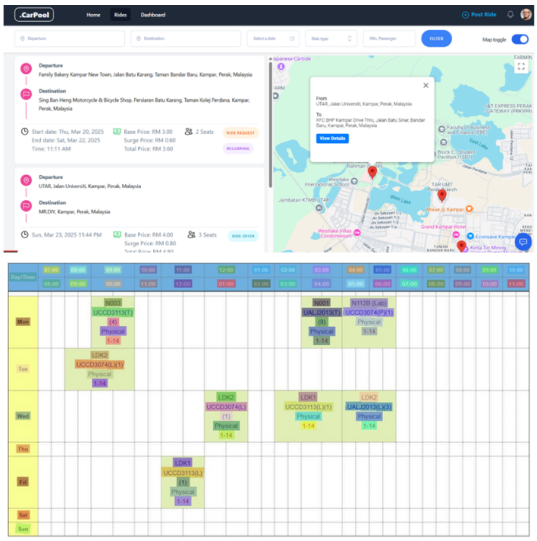
Conclusion

The core system, including:

- user registration / login
- ride posting using UTAR timetable
- ride booking
- chatbot-assisted ride searching

has been successfully developed and tested.

The project demonstrates feasibility, scalability, and addresses real-world student commuting problems effectively.



Project Developer: Tan Jian Hua
Project Supervisor: Dr. Ng Hui Fuang