

# **Riding in a Share**

CSSE6400 P01 Capstone Project Report



By Bailey Stoodley, Henry Batt, Mohamad Dabboussi,  
Shristi Gupta, Ferdi Sungkar and Ho Vu Khanh Vy

# Abstract

This report details the development and implementation of "Riding in a Share," a web-based ride-sharing platform designed to enable passengers to request rides and share trips with others heading in similar directions. By promoting shared travel, the system aims to reduce user costs and traffic congestion. Key functionalities include the drivers' ability to approve passenger requests, manage trip information, and estimate costs.

The project employs a monolithic architecture for its simplicity and ease of deployment, leveraging technologies such as Flask for the backend, React for the frontend, and various geocoding and routing tools. This architecture ensures high availability, scalability, and security, which are essential for handling sensitive user data and supporting potential future growth. The system's components work together to optimise route planning, trip matching, and real-time updates, providing a seamless user experience for both drivers and passengers.

The report also evaluates different architectural options and discusses the chosen approach's trade-offs, testing plans, and potential future enhancements.

# Table of Contents

<b>Abstract.....</b>	<b>1</b>
<b>Table of Contents.....</b>	<b>2</b>
<b>Introduction.....</b>	<b>3</b>
<b>Changes.....</b>	<b>4</b>
Single Development Platform.....	4
Window-Based Ride Requesting.....	4
Driver First Control.....	5
User Login System.....	5
<b>Architecture Options.....</b>	<b>5</b>
Monolithic Architecture.....	5
Microservices Architecture.....	6
Event-Driven Architecture.....	7
Overall Chosen Architecture.....	8
<b>Architecture.....</b>	<b>9</b>
Monolithic Core.....	9
Celery Workers.....	9
Geocoding and Routing.....	9
Database.....	10
User Interface.....	11
Backend.....	11
Architecture Diagrams.....	12
<b>Trade-Offs.....</b>	<b>15</b>
<b>Testing Plans.....</b>	<b>16</b>
Functionality Tests.....	16
Quality Attribute Testing.....	19
<b>Evaluation and Critique.....</b>	<b>21</b>
<b>Reflection and Extensions.....</b>	<b>22</b>
<b>References.....</b>	<b>25</b>

# Introduction

As per the project proposal [Riding in a Share \[1\]](#), traffic is increasing globally. People are returning to work, travelling, and utilising their mobile vehicles more often. This system aims to match passengers and drivers travelling to a similar location at a small cost to motivate the driver to utilise this system.

Additionally, the proposal outlined the following quality attributes that are required to be met:

1. **Availability** is a key attribute of the system. Riders and drivers may plan or request rides anytime, with plans potentially changing without notice. Therefore, it is essential that the app is always available to ensure a seamless experience and to prevent any user from being left without a ride or rider.
2. **Scalability** is another crucial quality attribute. While the system may initially have only a few hundred users, the goal is to achieve widespread adoption, significantly increasing the app's load and infrastructure. Consequently, the app must be scalable to accommodate many users and efficiently obtain the necessary resources to manage the increased load.
3. **Security** is a fundamental quality attribute. Due to the sensitive nature of the data stored, the app could attract malicious entities seeking to steal this information. Therefore, the app must be designed to maintain normal operations during attacks and ensure that the data remains secure.

In addition to the non-functional requirements, the proposal outlines some functional requirements to guide a successful Minimum Viable Product (MVP).

## 1. User Interface

The system will provide distinct user interfaces tailored for the two types of users: drivers and riders. This differentiation ensures that each user type can access features and functionalities specific to their needs, enhancing the overall user experience.

## 2. Route Planning for Drivers

Drivers can access a map and search bar to input their destination and departure time. The system will generate the most efficient route based on real-time traffic data, helping drivers navigate optimally and reducing travel time.

## 3. Driver Settings and Notifications

Drivers can configure their settings to specify a minimum deviation distance for rider pick-up. This feature will allow drivers to manage their routes more effectively and avoid significant detours. Additionally, the system will notify drivers when potential riders are within the specified deviation distance, enabling timely and convenient ride arrangements.

## 4. Ride Request Acceptance

The system will include functionality for drivers to accept ride requests from riders. This feature will facilitate the matching process between drivers and riders, ensuring that ride-sharing is seamless and efficient.

## 5. Route Selection for Riders

Riders will be able to select their destination and desired arrival time. Based on these inputs, the system will present multiple route options, offering flexibility and convenience for riders in planning their trips.

## 6. Route Option Details

Each route option presented to riders will include comprehensive details to aid in their decision-making. The displayed information will consist of the following:

- Departure and arrival times
- Cost of the ride
- Information about the driver, such as their name and rating

# Changes

Due to the scope of this MVP and the time constraints to deliver this project, some changes to the above non-functional and functional requirements had to be made. The sections below outline those changes.

## Single Development Platform

### Description

Switching from a separate website and mobile application, instead focusing on a single website with reactive design.

### Reasoning

Focusing on a single application and platform helps streamline the development cycle with a quicker deployment turnaround. It also decreases project overhead and, thus, resource requirements. By ensuring our web application is reactive and able to work on any device, we still support mobile users who use our application without a dedicated mobile app.

## Window-Based Ride Requesting

### Description

Utilise a request window for passengers on the system where users specify their pickup and dropoff locations and a start and end time window of when their trip should start.

### Reasoning

Using pickup windows allows more leniency in the matching algorithm, as users can have larger pickup times with which drivers can be paired. It also helps reward pre-planning and larger availability to increase the likelihood of being matched, as outlined in the proposal where “users can .. pre-plan their trips.” [1]

## Driver First Control

### **Description**

Emphasise drivers managing the final approval of trip matching instead of a two-way approval system.

### **Reasoning**

The driver-first approval system, where a passenger gives their pickup window, and drivers determine who to pick up, helps streamline the pairing process while decreasing the time overhead. This approach ensures that the driver is the only bottleneck in accepting matches instead of a driver approving someone, a passenger approving the approval, and then pairing them together.

## User Login System

### **Description**

Creating an intuitive login system allows users to log into the application securely.

### **Reasoning**

Due to the application's emphasis on security, we determined that creating a login system to store and manage users' data and system access is important. Whilst the proposal states that the "MVP will not include ... user profiles for drivers and riders, [1]" this feature was implemented due to the requirement of system security.

## Architecture Options

Several architectural design patterns were considered when designing the architecture for the "Riding in a Share" application. Each pattern had its advantages and disadvantages, which have been detailed below:

### Monolithic Architecture

#### *Advantages*

With a monolithic architecture, everything is contained within a single codebase. This means developers can easily understand the entire system without navigating multiple repositories or services. The integrated environment reduces the complexity of setting up the development environment and running tests, leading to quicker development cycles. Developers can run the entire application locally, making it easier to catch bugs and ensure all components work well together.

Deploying a monolithic application is straightforward because it involves a single deployable unit. It is not required to manage dependencies and interactions between multiple services, which simplifies the deployment process. Continuous integration and

deployment pipelines can be simpler to set up, as you are dealing with a single artefact to deploy, reducing the chances of deployment-related issues.

In a monolithic architecture, different application components communicate directly within the same process. This reduces the overhead of network calls between distributed services, leading to lower latency and higher performance. This direct communication can be particularly beneficial for applications that require high throughput and quick response times.

### *Disadvantages*

Monolithic applications can be challenging to scale. Since all components are tightly coupled, it is difficult to scale individual application parts independently based on their specific load requirements. This often leads to inefficient resource use, as it might be needed to scale the entire application to handle increased load in one specific part. Additionally, monolithic databases can become bottlenecks as they struggle to handle increasing volumes of data and requests.

In a monolithic architecture, the tight coupling between components can lead to poor modularity. This makes it difficult to isolate changes to specific parts of the application. A change in one part of the system can have unintended consequences in another part, making it risky and complex to update the application. This lack of modularity can slow development as the codebase grows, leading to a phenomenon known as "spaghetti code," where the interdependencies become so tangled that making changes or adding new features becomes cumbersome and error-prone.

In summary, monolithic architectures can be a good choice for smaller applications or projects in their early stages due to their simplicity and performance benefits, which is why they were considered. However, as applications grow in complexity and demand, the disadvantages can outweigh the benefits, often prompting a shift to microservices or other modular architectures that better support scalability and maintainability, which should be considered in the future of the application.

## Microservices Architecture

### *Advantages*

One of the biggest advantages of a microservices architecture is its ability to scale services independently. This means that the system can allocate resources and scale up parts of your application that experience higher demand without needing to scale the entire system. For instance, if the routing engine experiences a spike in traffic, it can scale just that service to handle the load, leading to more efficient resource use and cost savings.

Microservices enhance fault isolation, meaning the failure of one service is less likely to impact the entire application. If one service goes down, the rest of the services can continue to operate, improving overall availability and reliability. This modular approach ensures that critical parts of the application remain functional despite individual service failures, which is crucial for maintaining uptime and a positive user experience.

## *Disadvantages*

The microservices architecture inherently increases system complexity. Managing communication between services typically requires a robust inter-service communication strategy, often involving APIs or message brokers. Ensuring data consistency across multiple databases used by different services adds another layer of complexity, as does handling distributed transactions. Developers need to be well-versed in these areas, and the increased complexity can lead to longer development times and higher chances of bugs.

Deploying a microservices-based application can be quite difficult as each service may have its deployment lifecycle, and coordinating deployments to ensure compatibility and functionality across services requires sophisticated orchestration. This often necessitates additional infrastructure, such as container orchestration platforms and continuous integration/continuous deployment (CI/CD) pipelines, which can be difficult to set up and maintain.

This architecture allows building, deploying, and scaling individual services independently. This aligns with the need to handle varying loads on components like the routing engine and backend. By scaling only the necessary parts of the application, resources are used more efficiently. Additionally, it enhances fault isolation, ensuring that a failure in one service does not bring down the entire application. This significantly improves the overall availability and reliability of the system, which is crucial for maintaining service quality and user satisfaction. By adopting a microservices architecture a robust and scalable solution can be created that adapts to changing demands whilst maintaining high availability.

## Event-Driven Architecture

### *Advantages*

Event-driven architecture promotes loose coupling between components. Each component operates independently and communicates through events, enhancing the system's modularity. Such modularity allows easier updates and maintenance, as changes in one component do not necessarily impact others. This isolation makes developing, testing, and deploying individual components simpler.

The architecture allows the application to scale horizontally. Multiple consumers can be added to process events concurrently, enabling the system to handle increased loads more efficiently. This flexibility is particularly beneficial for applications that experience variable traffic, as it ensures that the system can scale up or down based on demand without significant architectural changes. This is particularly important for the Riding in a Share system as when there are physical events, i.e., the 2032 Olympics, the application will undergo significantly increased demand.

Event-driven systems improve fault tolerance and reliability by processing messages asynchronously. If an event handler fails to process a message, the system can retry the operation until it succeeds. This approach ensures that temporary failures do not result in data loss or system crashes, enhancing the overall reliability of the application.



## *Disadvantages*

Testing an event-driven system can be challenging due to the asynchronous nature of interactions. End-to-end tests need to account for events being processed at different times and potentially out of order. This complexity makes it harder to create reliable test scenarios and ensure comprehensive coverage, which can lead to longer development cycles and more bugs slipping through.

The overall system complexity increases with the use of an event-driven architecture. Managing asynchronous events, ensuring data consistency, and handling complex event flows require complicated design and thorough error handling. Developers need to understand the complicated nature of event processing and coordination, which can increase the learning curve and development time.

Deploying an event-driven architecture involves managing multiple independent components and their interactions. This requires additional tools and infrastructure for deployment, monitoring, and management, such as message brokers and event tracking systems. The need for such management can complicate the deployment process and require specialised skills to manage effectively.

Given the application's need to handle asynchronous interactions between riders and drivers, an event-driven architecture suits the system. The architecture supports loose coupling and modularity, ideal for managing asynchronous interactions. This would help maintain and update different application parts without causing disruptions. The ability to scale horizontally by adding more event consumers aligns with the need to handle increasing volumes of ride requests. This ensures that the system remains responsive and efficient under varying loads. The asynchronous processing and fault tolerance mechanisms ensure the system can reliably handle real-time adjustments based on location and route deviations. Retrying failed events improves the overall reliability and user experience.

By adopting an event-driven architecture, the system would be better equipped to handle the asynchronous and real-time nature of interactions in the application, providing a scalable, reliable, and modular solution.

## Overall Chosen Architecture

As this project is for the purpose of building an MVP, the monolithic architecture emerged as the most viable option due to its simplicity and performance advantages. With everything encapsulated within a single codebase, the team can develop their sections easily, streamlining the development process (especially important due to time considerations). Deployment is straightforward, as there's only one deployable unit to manage, simplifying continuous integration and deployment pipelines. Moreover, direct communication between components reduces latency and enhances performance, which is particularly advantageous for applications requiring high throughput. However, in the future, it is important to note that for a more developed version of this application, microservices would be most suitable. However, as it is more difficult to set up and get going, this was something that was decided against moving forward with.

# Architecture

As per the project requirements (functional and non-functional), the software architecture must be an available, scalable, and secure solution for our ride-sharing platform. The ride-sharing platform follows a **monolithic architecture** that uses detached decoupled workers and a detached database. The monolith was used to enhance system integrity and security, and it has a single-state deployment while remaining scalable. The backend uses no external system, and all data that comes in is the user's HTTPS request, where processing is entirely contained by the systems backend. Hence, the backend uses a completely self-hosted geographic information system, GIS, and ecosystem for scalability, reliability, availability, privacy, and performance.

## Monolithic Core

As stated previously, the application is deployed with a monolithic core. It mainly stores data, fetches data, and spins up workers to perform the matching algorithm.

## Celery Workers

Asynchronous Celery workers were used to perform trip and trip request matching since it is expected to be a time-heavy operation. Haversine distances between every opposing (trip requests for trips, and vice versa) were calculated for every entry that was not pending. However, to lighten the weight of this operation, the algorithm stopped once there were twice the number of seats available in the car to suggest to the user or add to the trip options. A single FIFO queue was used to load requests for processing.

There were two different types of requests that workers could perform since matching trips to trip requests and trip requests to trips have reverse algorithms. The number of workers was dynamically adjusted based on the number of requests in the queue. This design allowed the system to scale according to the current demand, ensuring efficient processing of trip and trip request matching.

This approach ensured that the matching process remained efficient and responsive even as the number of users and requests grew. By limiting the number of suggestions to twice the car's seating capacity, the algorithm avoided unnecessary computations, improving overall system performance. Additionally, the shared FIFO queue and dynamic worker allocation enabled the system to handle varying loads effectively, maintaining high availability and reliability for users.

## Geocoding and Routing

To achieve our goal of a fully self-hosted, non-externally reliant system, we deployed and maintained our own Geocoding and Routing Engine.

Geocoding is the process of converting a text-based address, in our case, a street address, into latitude and longitude. To achieve this, we use a tool called Nominatim [2], which searches through open-sourced street map data provided by OpenStreetMap (OSM).

In the context of our application, this service serves as an essential first step in satisfying functional requirements such as trip matching and routing. The provided JavaScript function

'fetchCoordinates', seen below, demonstrates how the application interacts with the Nominatim API to fetch latitude and longitude based on a given address.

```
export const fetchCoordinates = async (address) => {
  const url = `${ROUTING_ENGINE_URL}/search?q=${encodeURIComponent(address)}&format=json`;
  const response = await fetch(url);
  if (response.ok) {
    const data = await response.json();
    if (data.length > 0) {
      return { latitude: parseFloat(data[0].lat), longitude: parseFloat(data[0].lon) };
    } else {
      console.log("No results found for the address.");
    }
  } else {
    throw new Error("Failed to fetch coordinates.");
  }
};
```

**Figure 1:** Code snippet of the street address to lat/long call.

Our routing engine tool was Open Source Routing Machine (OSRM) [4], a C++ routing engine that determines the shortest path in a network of roads. It is one of the most popular and compatible free and open-source routing engine solutions and allows us to quickly and efficiently formulate potential routes drivers could take to collect and drop off passengers.

These tools join together to form the key components of our geographic information system, simplifying and streamlining the request process for these vital features. Both of these tools utilise free, publicly available, and open-source data through OpenStreetMap (OSM).

These tools were deployed onto our cloud platform through docker images [3] [4], allowing them to be entirely controlled and operated by our system. This ensures that all traffic sent to them is contained inside our system and its components and aligns with the self-contained backend solution we envisioned. This also allows greater scalability and availability options since it is self-managed. As our application scales up, additional routing engines or geocaching APIs can be provisioned and managed to ensure we successfully deal with variable loads. Control over the availability is also gained as the application is no longer reliant on external systems that could have varying downtime. Instead, it can be managed and rectified by the team, or in most cases automatically, to ensure there is minimal downtime and these essential tools are always available.

## Database

The platform uses SQLAlchemy, a robust SQL toolkit and Object-Relational Mapping (ORM) library for Python, integrated with PostGIS. PostGIS extends PostgreSQL with spatial data types and functions, allowing for efficient storage and querying of geographic information, by combining SQLAlchemy with PostGIS, the platform benefits from the versatility of SQLAlchemy's ORM capabilities and PostGIS's spatial data handling capabilities. This integration enables interaction with spatial data within the application, supporting a wide range of location-based features while maintaining the benefits of a relational database management system.

## User Interface

The platform uses React as its user interface library. React's component-based architecture ensures modularity and reusability of the code. It enables efficient development of scalable web applications, which enhance the platform's monolithic core.

Through React's virtual DOM, the platform reduces performance costs and enhances user experience. Because of React's declarative programming style, it provides predictable data management with unidirectional data flow, which makes the codebase robust and maintainable and benefits the monolithic core.

Additionally, React's ecosystem and large community ensure a high-quality application structure and enhance the monolithic core of the platform.

## Backend

The web framework utilised to serve our application is Flask, the Python micro web framework. Whilst Flask is built for Python, we used the PyPy Python implementation instead. PyPy was chosen due to performance benefits gained whilst still being familiar due to being a Python implementation. Due to PyPy's just-in-time compiler, it is on average 4.8 times faster than CPython [6].

Flask is a lightweight and flexible Python Web Framework whose simplicity and modularity help build a robust backend and make the platform easy to scale. By using Flask, the platform benefits from its quick development cycles, easy debugging, and straightforward testing, resulting in its maintainability.

In addition to Flask, Flask-RESTful was also utilised to serve our backend content. Flask-RESTful is a Flask extension that supports quickly building REST API's with a lightweight abstraction [7]. Paired with our Restful API, the backend can communicate with the front end, sending and receiving data. Restful API is an integral part of the platform due to its efficient data exchange and integration with other systems, and it contributes to the monolithic architecture of the platform.

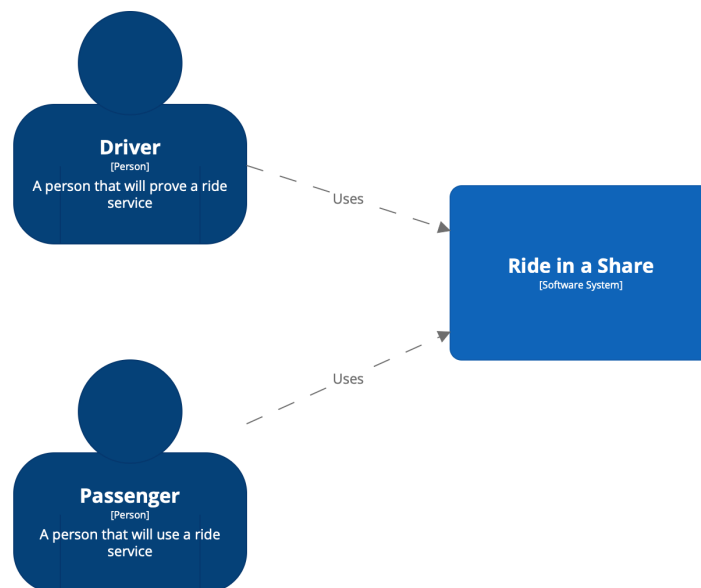
To deploy our production web application, a WSGI (Web Server Gateway Interface) was required to handle communication between the web server and our Flask framework. The chosen gateway interface was Gunicorn, a pre-forked worker-based Unix WSGI server [8]. This ensured our application was production-ready, with multiple workers joining to concurrently serve multiple requests on a single node, thus aiding scalability.

# Architecture Diagrams

The first three levels of C4 Diagrams have been drawn up for the system to further demonstrate the architectures discussed above.

## Ride in a Share Context Diagram

The context diagram below provides a general understanding of how the system interacts with its users and external services, highlighting the roles and interactions within the system's scope to offer a clear picture of the overall functionality and user experience. Drivers interact with the system by logging rides, receiving suggestions for potential passengers to pick up, approving these pickups, viewing the list of confirmed passengers, and accessing the optimal route on the map for efficient pickups and drop-offs. Passengers use the system to log trip requests, receive updates on whether their ride has been accepted by a driver, and view the cost of their trip before confirming the request. Additionally, the system integrates with self-hosted services, such as a geocoding service to convert addresses into geographic coordinates and a routing engine to calculate the best route considering all pickups and drop-offs.

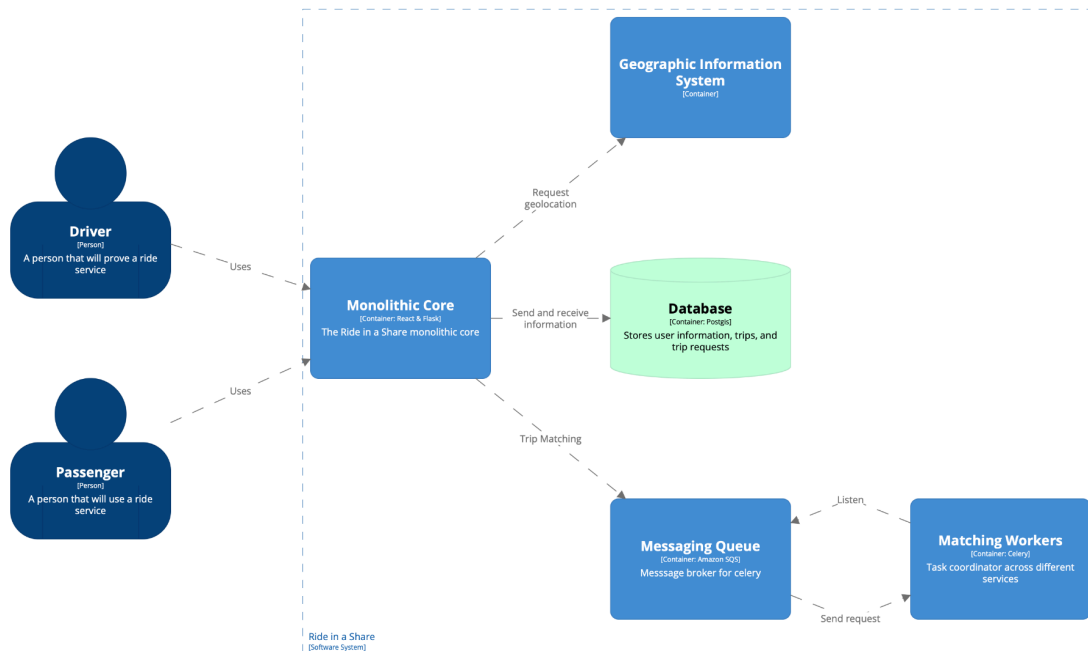


[System Context] Ride in a Share  
Thursday, 30 May 2024 at 12:03 pm Australian Eastern Standard Time

**Figure 2:** System Context Diagram.

## Ride in a Share Container Diagram

The container diagram below illustrates the monolithic core of the system. In this architecture, the monolithic, which encompasses both the front end and back end, directly interacts with the system's database. This monolithic design ensures that the web application handles all user requests and operations internally, without needing to make external requests for additional services. Additionally, the monolithic core communicates directly to the Geographic Information System and the Messaging Queue (workers).

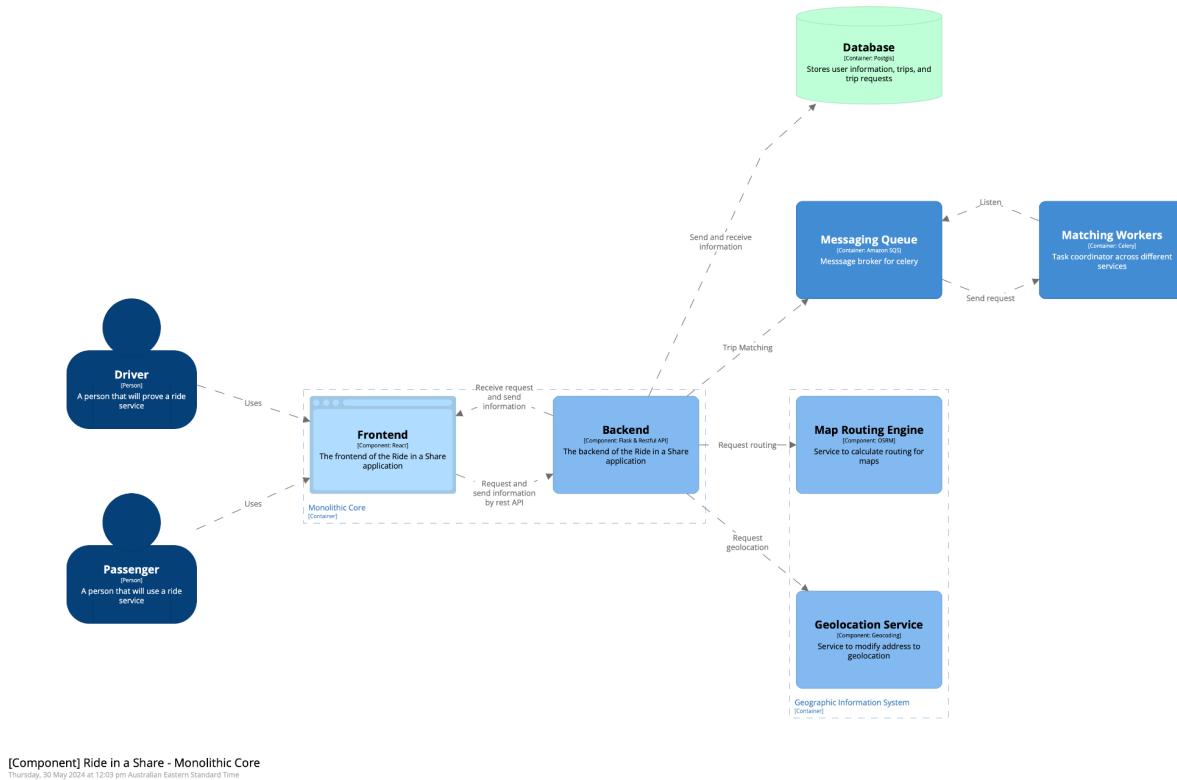


[Container] Ride in a Share  
Thursday, 30 May 2024 at 12:03 pm Australian Eastern Standard Time

**Figure 3: System Container Diagram.**

## Monolithic Core Component Diagram

The component diagram below provides a view of the ride-sharing application's architecture, illustrating how various system components interact to deliver the service. In this architecture, both Drivers and Passengers use the front end. The front end handles user requests and communicates with the back end, which is developed using Flask and RESTful API. The back end processes these requests, interacts with the PostgreSQL Database to store user information, trips, and trip requests, and utilises external services for additional functionalities. Specifically, the Backend sends routing requests to the Map Routing Engine (OSRM) to calculate routes and geolocation requests to the Geolocation Service to modify addresses into geographic coordinates. Additionally, for trip matching, the Backend communicates with a FIFO Messaging Queue. This Messaging Queue facilitates information exchange with matching workers, which use Celery to coordinate tasks across different services.



**Figure 4: System Component Diagram**

## Trade-Offs

Throughout the project, we encountered numerous trade-offs. The first of which is the choice between a monolithic architecture and microservices architecture, which significantly impacts availability, scalability, ease of deployment, and long-term maintainability. As outlined in the architectures section, microservices would have been a better option in the long term, due to the advantages in availability and scalability. However, the monolithic architecture was easier to work with due to ease of deployment and simplicity, and for an MVP these were significantly more important. By adopting the monolithic architecture, the team was able to deliver an MVP in the timeframe of this project and deploy the project as a single unit.

Ensuring the platform was secure was of high priority, as highlighted in one of the most important architectural significant requirements. Our application incorporated a login system and simple password hashing with HTTPS to assist with security measures. In addition, each API call mandated a username and password to verify the authenticity of the user requesting information. However, due to the team's skills and time restraints for delivering an MVP, more robust security measures like multi-factor authentication, session management, and more thorough data encryption were not used to instead focus on other developments of the system and its auxiliary deliverables.

As discussed in the ADR records, we had minimised some functional features due to time limitations. A trade-off that was made and outlined in the changes was removing the mobile application side of the proposal. We made the web application responsive to any device to satisfy the aim of this ASR. This trade-off allowed us to streamline development and focus resources on a single platform, ensuring a consistent user experience across desktops, tablets, and smartphones. While this decision simplified deployment and maintenance by avoiding the complexities of developing and supporting a separate mobile application, it also meant sacrificing some of the performance and usability benefits native mobile apps provide. However, the responsive web application effectively meets user needs by offering a flexible and accessible solution that adapts to various screen sizes and devices.

Another MVP feature that has been adjusted is the differentiation in the driver and passenger sign-up and login pages, which resulted in more design and testing complexity than the initial implementation. Initially, we intended to have only one login user interface for both driver and passenger, and we would always redirect users to the passenger interface by default. This approach would not meet the driver's needs and requires more steps if drivers want to track the trips. Therefore, even if the separation of interfaces sacrifices simplicity and requires more effort in design and testing, the benefits it brings to a seamless experience outweigh the drawbacks.

Additionally, the matching algorithm that the workers ran was not optimised. The MVP's initial focus was to get matching working, which was time-consuming. However, the matching algorithm is sufficient for the demonstration and has doable and clear extensions. This approach allowed for a faster development cycle and a proof of concept but at the cost of increased processing time and resource utilisation. As a result of this tradeoff, the system's efficiency and scalability were compromised in the short term, necessitating future optimisation efforts to handle larger user bases and more complex matching scenarios effectively.

Regarding the map functionality, the backend currently provides a list of passengers on the route based on the Haversine distance (longitude and latitude distances). The trade-off between accuracy and functionality was made to ensure there was semi-reasonable functionality for the MVP. Using the Haversine distance provides a reasonable response in urban areas with many



streets and pathways. However, this approach can result in non-optimal routes that may appear impractical in less dense areas. This method does not account for actual paths and routes available, which can lead to inefficient routing. Ideally, the sorting should be done based on a routing engine that considers real-world paths and road networks rather than relying solely on the general distance metric provided by the Haversine formula. This improvement would lead to more accurate and practical routing solutions. As a result of this trade-off, the system currently prioritises speed and simplicity over precision, making future enhancements necessary to achieve more efficient and realistic route planning.

## Testing Plans

A testing plan was devised to ensure the system meets its functional and non-functional requirements. This plan includes automated and manual tests to validate the system's functionality, performance, and security. The testing framework leverages GitHub Actions for continuous integration and local scripts for convenience, covering various scenarios from user profile creation to trip management. The functional requirements are tested with specific scripts designed to simulate real-world interactions and verify expected outcomes. Additionally, security testing is conducted using SQLMap to safeguard against potential vulnerabilities. The following sections outline the detailed testing procedures and results, demonstrating the robustness and reliability of the platform.

### Functionality Tests

Our tests are automated using GitHub Actions, triggered on every push and pull request to the repository. GitHub Actions automatically handles the setup of the environment and installation of dependencies, ensuring that all tests are executed in a consistent environment.

#### ***How to Use Tests Locally***

Regarding local testing, a unit test script has been developed to simplify the running of tests on a local machine. The steps to use it include:

1. Navigate to the project directory.
2. Run the script by executing `./unittest.sh` in your terminal.

This script directly runs the tests, how it requires that Python be installed and the corresponding dependencies for the application are installed. These requirements are defined in `application/requirements.txt` when can be installed using Python pip.

#### ***Functional Requirements and Corresponding Tests***

##### **1. User Profile Creation**

The system must allow users to create profiles, including necessary details such as name, phone number, and email. Associated tests:

- `test_1_create_passenger`: Verifies that the app can successfully create a passenger profile and returns the correct status code and user details.

- *test\_1\_create\_driver*: Ensures that a driver profile can be created with all required details accurately stored and retrieved.

## 2. Unique User Identification

The system should prevent the creation of duplicate profiles to maintain data integrity. Associated Tests:

- *test\_2\_passenger\_already\_exists*: Checks the system's ability to identify and reject duplicate passenger entries.
- *test\_4\_nondrivers\_cannot\_create\_trip*: Ensures that only valid driver profiles can initiate trip requests, thereby preventing misuse by incorrect user roles.

## 3. Trip Request Management

Passengers should be able to request trips, specifying details like pickup/dropoff location and desired time. Associated Tests:

- *test\_3\_add\_trip\_request*: Confirms that the system can handle new trip requests by storing and acknowledging them correctly.
- *test\_4\_conflicting\_trip\_request\_schedule*: Tests the system's capacity to detect and reject trip requests that conflict with existing schedules.

## 4. Trip Management

Drivers should be able to create trips, specifying details like start/end window and desired time. Associated Tests:

- *test\_2\_create\_trip*: This test verifies that drivers can successfully create a trip, allowing drivers to specify trip details such as destination, time, and expected route.
- *test\_3\_conflicting\_trip\_schedule*: This test checks the system's ability to handle conflicts in trip scheduling. It's designed to prevent drivers from double-booking themselves or creating overlapping trips

## 5. Retrieval of Trip Data

The system must allow users to retrieve information about trips, including statuses and related details. Associated Tests:

- *test\_5\_get\_trip\_requests*: Ensures that trip requests for a specific passenger can be accurately retrieved and displayed.
- *test\_5\_get\_all\_trips\_for\_user*: Verifies that all trips associated with a specific driver are correctly retrieved and contain accurate details.

## 6. Handling Nearby Trip Requests

Drivers should receive notifications about nearby trip requests that fit within specified times and locations. Associated Tests:

- *test\_1\_get\_all\_nearby\_pending\_trips*: Check whether the system can effectively identify and list nearby trip requests that match the driver's criteria.

## 7. Trip Request Approval Process

The system should allow drivers to approve trip requests. Associated Tests:

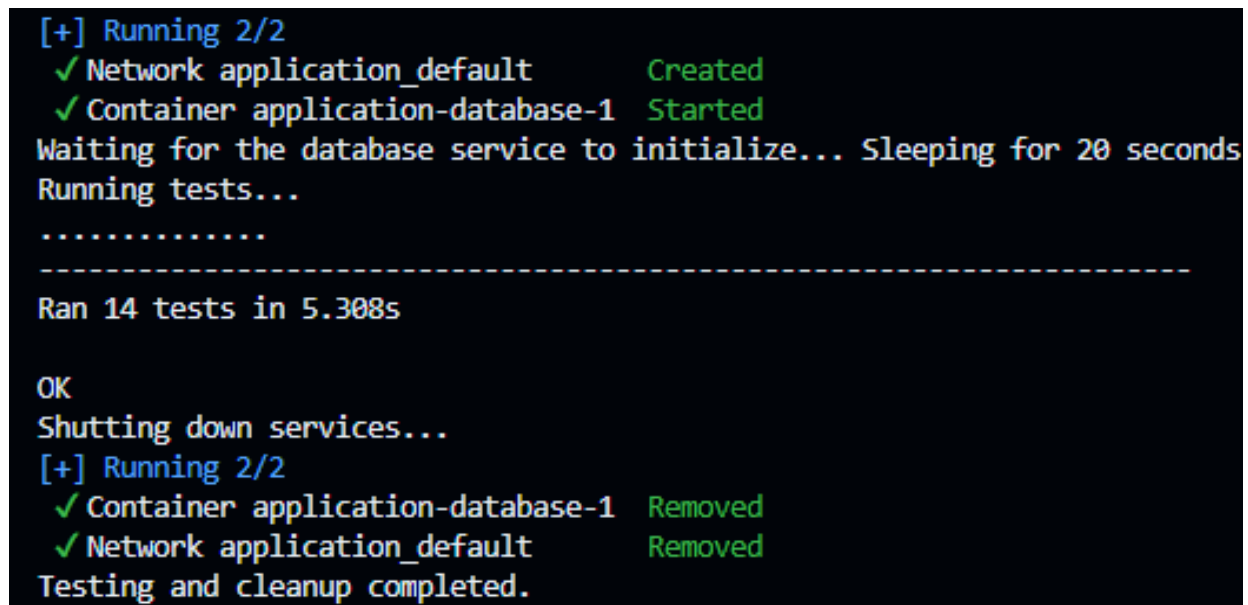
- *test\_2\_approve\_trip\_request*: Tests the functionality where drivers can approve trip requests, and the system updates and acknowledges this action.
- *test\_3\_get\_approved\_trip\_requests*: Confirm that approved trip requests are recorded and retrievable from the system, reflecting their updated status.

### *How Tests Cover Functional Requirements*

Each test is designed to address specific functional requirements by simulating user interactions with the system and verifying the outcomes against expected results. By doing so, the tests ensure that:

- User data is handled correctly, maintaining integrity and privacy.
- Trip requests are managed efficiently, with appropriate checks for conflicts and validity.
- Data retrieval functions are reliable, ensuring users can access their information and trip details when needed.
- System responses are appropriate and informative, aiding in user decision-making and system usability.

**Test result:** A suite of 14 automated tests was performed, covering all major functional areas of the application. All tests successfully passed, demonstrating that the application meets its functional requirements as specified in the project proposal.



```
[+] Running 2/2
✓ Network application_default      Created
✓ Container application-database-1 Started
Waiting for the database service to initialize... Sleeping for 20 seconds
Running tests...
.....
-----
Ran 14 tests in 5.308s

OK
Shutting down services...
[+] Running 2/2
✓ Container application-database-1 Removed
✓ Network application_default      Removed
Testing and cleanup completed.
```

**Figure 5:** Unit Test Results

# Quality Attribute Testing

## Security

For security testing, the team utilised SQLMap, a powerful open-source penetration testing tool known for its effectiveness in identifying SQL Injection vulnerabilities, allowing it to assess the robustness of database interactions against malicious inputs.

**Sqlmap's Role in Our Security Testing:** In our specific case, SQLMap is configured to target the PostgreSQL database system used by our application. We use a scripted test that sends a crafted HTTP POST request to simulate user logins or profile submissions. This request contains typical user credentials formatted in JSON. Sqlmap executes this test with parameters set for high test depth (`--level=5`) and risk (`--risk=3`), ensuring it performs an exhaustive search for vulnerabilities. This comprehensive approach helps us ensure that our application can robustly handle SQL queries without exposing the system to SQL injection attacks.

### Running the Sqlmap Test Script:

1. Make the script executable (if not already set): `chmod +x sqlmap.sh`
2. Execute the script with the required host parameter: `./sqlmap.sh <host-url>`

In the above directions, replace `<host-url>` with the URL where the application is hosted. This script facilitates the environment setup, prepares the HTTP request file, and runs SQLMap against the specified host to perform the security tests.

**Test result:** Sqlmap has tested the parameters it was provided (`{"username": "test", "password": "test", "user_type": "driver"}`) and has determined that it could not successfully perform SQL injection on any of them. This suggests that the parameters are not vulnerable to SQL injection. This outcome indicates that the application has robust security measures in place that properly manage SQL queries and prevent malicious data manipulation.

```
[10:29:22] [INFO] testing 'PostgreSQL AND time-based blind (heavy query)'\n[10:29:22] [INFO] testing 'PostgreSQL OR time-based blind (heavy query)'\n[10:29:23] [INFO] testing 'PostgreSQL AND time-based blind (heavy query - comment)'\n[10:29:23] [INFO] testing 'PostgreSQL OR time-based blind (heavy query - comment)'\n[10:29:23] [INFO] testing 'PostgreSQL > 8.1 time-based blind - Parameter replace'\n[10:29:23] [INFO] testing 'PostgreSQL time-based blind - Parameter replace (heavy query)'\n[10:29:23] [INFO] testing 'PostgreSQL > 8.1 time-based blind - ORDER BY, GROUP BY clause'\n[10:29:23] [INFO] testing 'PostgreSQL time-based blind - ORDER BY, GROUP BY clause (heavy query)'\n[10:29:23] [INFO] testing 'Generic UNION query (NULL) - 1 to 10 columns'\n[10:29:24] [INFO] testing 'Generic UNION query (random number) - 1 to 10 columns'\n[10:29:24] [WARNING] parameter 'Referer' does not seem to be injectable\n[10:29:24] [CRITICAL] all tested parameters do not appear to be injectable\n\n[*] ending @ 10:29:24 /2024-05-27/
```

Figure 6: SQLMap Test Results

## Scalability

### Running the Scalability Test with k6

To test the scalability of the riding-share web application, we utilised the k6 tool, which is designed for load testing. The attached results reflect a test execution designed to simulate a scenario with 100 virtual users (VUs) over a duration of 6 minutes and 30 seconds.

### Running the Test

To run the k6 test, we used a script called ``run_k6.sh``, which contains the necessary commands to execute the k6 test script (``k6.js``). Here's a brief overview of how to run the test:

1. Ensure k6 is installed: If not installed, download and install k6 from the official [k6 website](#).
2. Execute the script: Run the following command in your terminal: `./run_k6.sh`

This script will initiate the k6 test with the specified parameters.

### Analysing the Results

```
Please enter the API endpoint:
http://ridingshare-850758134.us-east-1.elb.amazonaws.com
Running k6 test for the endpoint: http://ridingshare-850758134.us-east-1.elb.amazonaws.com

M K6 .io

execution: local
script: k6.js
output: -

scenarios: (100.00%) 1 scenario, 100 max VUs, 6m30s max duration (incl. graceful stop):
* studier: Up to 100 looping VUs for 6m0s over 3 stages (gracefulRampDown: 30s, exec: simulateUserLogin, gracefulStop: 30s)

■ User Actions

X is status 201
L 99% - ✓ 1707 / X 1

checks.....: 99.94% ✓ 1707 X 1
data_received.....: 555 kB 1.5 kB/s
data_sent.....: 404 kB 1.1 kB/s
group_duration.....: avg=9.18s min=1.2s med=5.79s max=52.58s p(90)=20.04s p(95)=28.8s
http_req_blocked.....: avg=12.05ms min=2.02µs med=3.96µs max=211.17ms p(90)=6.03µs p(95)=206.32ms
http_req_connecting.....: avg=12.04ms min=0s med=0s max=211.09ms p(90)=0s p(95)=206.23ms
http_req_duration.....: avg=8.18s min=208.55ms med=4.77s max=51.58s p(90)=19.08s p(95)=28.13s
{ expected_response:true }.....: avg=8.18s min=533.12ms med=4.78s max=51.58s p(90)=19.09s p(95)=28.15s
http_req_failed.....: 0.05% ✓ 1 X 1707
http_req_receiving.....: avg=81.37µs min=26.54µs med=66.09µs max=8.62ms p(90)=87.57µs p(95)=91.98µs
http_req_sending.....: avg=31.99µs min=12.37µs med=30.19µs max=162.82µs p(90)=38.9µs p(95)=61.26µs
http_req_tls_handshaking.....: avg=0s min=0s med=0s max=0s p(90)=0s p(95)=0s
http_req_waiting.....: avg=8.18s min=208.46ms med=4.77s max=51.58s p(90)=19.08s p(95)=28.13s
http_reqs.....: 1708 4.737549/s
iteration_duration.....: avg=9.18s min=1.2s med=5.79s max=52.58s p(90)=20.04s p(95)=28.8s
iterations.....: 1707 4.734775/s
vus.....: 1 min=1 max=100
vus_max.....: 100 min=100 max=100

running (6m00.5s), 000/100 VUs, 1707 complete and 4 interrupted iterations
studier ✓ [=====] 000/100 VUs 6m0s
k6 test execution completed.
```

Figure 7: K6 Testing Results

The test was run on the deployed application. Here are the key results from the k6 test execution:

- Total Checks: 99.94% success rate with 1707 checks passed and 1 failed.

- HTTP Request Failed: Only 0.05% of requests failed.
- Iterations: 1707 iterations were completed, with an average iteration time of 9.18 seconds.

The results show that the system was able to handle the load reasonably well, with most requests completed within acceptable timeframes. The minor percentage of failed requests indicates areas where further optimisation might be needed to ensure higher reliability under load.

## Availability

To test the system's availability manual testing was conducted and the system's response was observed. To achieve this we simulated a loss of service or crash scenario where an application service was taken offline and monitored to ensure a new service would be provisioned to replace the dropped one, thus testing its reliability, resilience, availability, and ability to recover from a fault. This process happens automatically in the event of a failure ensuring the system remains operational and available.

During this testing after a service was killed off a new one was automatically spun up within minutes, demonstrating that the system can be quickly recovered and maintain its availability.

## Evaluation and Critique

The MVP for the “Riding in a Share” application has implemented several key functionalities that support the functional requirements stated in the proposal. The application greatly enhances the ride-sharing experience through effective matching and routing algorithms and distinct user interfaces for drivers and passengers. These interfaces allow each type of user to perform necessary tasks based on their specific needs. Sections 1 and 2 of the functionality tests highlight how the application allows users to create separate profiles for drivers and passengers and prevents duplicate profiles. Moreover, using React for the application's front end promotes modularity and components' reusability. This would be beneficial when extending the application to support mobile devices.

As the original proposal outlines, the application also allows drivers to add trips and passengers to request rides. The application's architecture allows drivers and passengers to easily and quickly retrieve trip information from the database, thus enhancing its usability. This is also supported by the unit tests listed in sections 3, 4, and 5 of the testing plan. Moreover, the application allows the driver to see ride requests from nearby passengers quickly. As evidenced in section 7 of the testing plan, the monolithic core is well-suited to handle the acceptance of ride requests by the drivers. The API endpoints facilitate the flow of necessary business logic and data between the back and front end. However, currently in the MVP, the rider cannot select a route based on cost and arrival times. Since this functionality will be similar to the driver side, the architecture can support this well.

As highlighted before, the application uses a self-hosted OSRM as the routing engine and handles geocoding through the Nominatim service from OpenStreetMap. While the MVP

provides the driver with a route covering all the pickup and drop locations of the passengers, currently, this routing is not optimised based on real-time traffic data. Optimising routes based on factors like real-time traffic data and user preferences can be computationally intensive and may require specialised data processing capabilities to ensure it happens promptly. Thus, the existing monolithic architecture could become a bottleneck, especially as the number of users and requests increases. To support such a feature, a more distributed architecture in terms of microservices would be beneficial. Separating the routing and optimisation logic into dedicated services would allow for better scalability and availability.

In the event that an error was to occur or the system went offline, a new instance of the application would be automatically provisioned. This was showcased in the video demonstration of this project. This process took about 2 minutes, meaning the system could recover quickly from unexpected outages. Additionally, the self-hosted auxiliary services ensured the system had constant access to these core specialised functions, further enhancing availability. Hence, the non-functional requirement and availability were sufficiently met for this application.

In terms of scalability, the system is designed to respond in different ways depending on the request load. In the case of many concurrent users, the application will scale up horizontally to handle additional incoming requests. If numerous trips and requests were created, resulting in an increased backlog of matching to occur, additional workers are spawned to manage the load and decrease the total clear time. This scalability approach is sufficient for our MVP; however, could be expanded upon in the future in various ways. One such manner is the reworking of the self-hosting of our geographic information system tools, as whilst it is currently not a bottleneck, its implementation made it harder to scale if the applications load were to consistently increase.

In terms of the non-functional requirement of security, there was a lot of room to improve in this area. The application currently only handles minimal security measures as outlined before, such as username and password logins, and requires every API call to have this information to ensure that the appropriate person is requesting that information. Considering the sensitivity of locational data, where someone would be at a specific time, this is concerning. As discussed above more security measures should be taken, such as multi-factor authentication and session management to handle this sensitive data more securely.

## Reflection and Extensions

The design and implementation of the Riding in a Share platform have provided valuable insights into software architecture, development processes, and project management. This experience has identified some key lessons and potential improvements for future projects.

One of the most significant lessons we learned is that establishing clear API expectations at the beginning of the project is essential. Defining what information needs to be given to the backend and what outputs are expected in specific formats helps streamline the development process and ensures consistency across the platform. It took the team a week to reimplement the new API as the format didn't support tracking and retrieving the trip's data and failed to function the required features. Starting with detailed API documentation could enhance communication between different development teams and improve overall system integration, saving time and avoiding spending time on implementing unused components.

Another important lesson related to how vital the optimisation of the calculation metric for the project is. While usable, our calculation using the Haversine formula for the trip matching feature may not always provide the precision needed for optimal route planning and trip coordination. Additionally, it is a time-heavy operation using multiple complex maths, such as arctan. Future iterations of the project could benefit from integrating more sophisticated geospatial algorithms or leveraging real-time traffic data to enhance accuracy.

Another lesson learned is the complexity and necessity of integrating internal assistive services from the outset. Services such as advanced geocoding, real-time navigation, and user authentication systems are crucial for the platform's functionality, user experience and development. By prioritising the installation and integration of these services early in the development process, we could mitigate potential delays and complications later on.

Regarding implementing the application's front end, we learned that it is vital to communicate well and make a clear plan from the beginning of what to implement and where. In the beginning, the team split the work of implementing each page without communicating more deeply about how the pages would work with one another. As a result, some of us have a different perspective on how the application flow will be implemented, resulting in an issue that would take us a few days to fix. We realised that communicating with each other initially on how the application would flow would save us days of fixing and re-implementing some pages. This issue took us a few days to fix and reimplement the pages between and the connections. We realised it would have been better to unify the flow from the start and determine tasks for each member earlier. Besides, considering possible issues that may have happened to the delivery features for each iteration would have also benefited the teams.

One of the most confusing problems for the frontend team was the styling of the pages. Two related issues occurred: we started on it late-stage and did not have any precise discussion about the style before starting. Initially, we thought the styling process would not take long, so we decided to style the application at the end when most of the pages were implemented. This resulted in the inconsistency of styling on some pages since they are implemented by different people, so they would have different structures. We learned that it would have been a better practice to implement styling as we go, not at the end of the process. Furthermore, As we did not discuss the styling thoroughly, conflicts occurred as the members responsible for the styling part did not like how the team wanted. This resulted in editing some pages and re-implementing the list of trips and request pages, as those were not dynamic and didn't fit the screen, affecting the user's experience. In conclusion, those issues are not serious and improvable. Still, they consume a lot of time and effort to fix and influence the user's experience, which delays the team's progress in finalising the project.

By identifying and learning from these mistakes, we expect to enhance the process speed and professionalism in future projects and works.

Despite these learning challenges and possible extensions, there was still a lot learned from this project. Multiple people have to improve their development skills to create a reactive web design that is both scalable and secure. The management and processing of geospatial data was a very informative topic where we learned a lot about different ways to store, process, and compute over these datasets to find efficient combinations of locations.

Whilst the systems architecture could have benefited from further pre-consideration, it was still a valuable experience to design a full, and in many ways complex system due to the various interconnected parts with the opportunity to hone skills in regards to the deployment and maintenance of services.



If we were to start again a decision we might further consider is decoupling the front and backends from a core monolith into a true microservice-based architecture where each application has further refinement over the scaling. Reworking the self-hosting of our geographic information system tools would also be beneficial as whilst it was currently not a bottleneck its implementation made it harder to scale if the load were to drastically increase consistently.

Additionally, greater discussions and planning over the user interface needs to occur with the entire team taking care to ensure the layout is consistent without being identical and the user's workflow on the app is enjoyable. As part of this consideration, the API communication between the front and back ends should be stripped back and reconsidered to simplify its structure and make it easier to maintain and extend in the future.

## References

- [1] P. Kodali, “Riding in a Share,” *CSSE6400 Proposals*. Available: <https://csse6400.github.io/project-proposal-2024/s4583705/proposal.html>.
- [2] Nominatim , “Nominatim Documentation,” *Nominatim Manual*. Available: <https://nominatim.org/release-docs/develop/>
- [3] Mediagis, “mediagis/nominatim-docker,” *GitHub*. Available: <https://github.com/mediagis/nominatim-docker>
- [4] OSRM, “OSRM API Documentation,” *OSRM API Documentation*. Available: <https://project-osrm.org/docs/v5.24.0/api/>
- [5] OSRM, “Docker osrm/osrm-backend,” *Docker Hub*, 2024. Available: <https://hub.docker.com/r/osrm/osrm-backend/>
- [6] PyPy, “PyPy Documentation,” *PyPy*, 2024. Available: <https://www.pypy.org/>
- [7] Flask-RESTful, “User’s Guide,” *Flask-RESTful Read The Docs*, 2024. Available: <https://flask-restful.readthedocs.io/en/latest/>
- [8] Gunicorn, “Gunicorn,” *Gunicorn*, 2024. Available: <https://gunicorn.org/>