Module Guide for Software Engineering

Team #16, The Chill Guys
Hamzah Rawasia
Sarim Zia
Aidan Froggatt
Swesan Pathmanathan
Burhanuddin Kharodawala

January 22, 2026

1 Revision History

Date	Developer	Notes
November 1st	Aidan Froggatt	Completed Section 11
November 12th	Hamzah Rawasia	Completed Sections 6 and 8
November 12th	Swesan Pathmanathan	Completed Sections 5 and 7
November 13th	Burhan Kharodawala	Completed Sections 3, 4, and 10
November 13th	Sarim Zia	Completed Sections 9, 10, and 12

2 Reference Material

This section records information for easy reference.

2.1 Abbreviations and Acronyms

symbol	description
AC	Anticipated Change
DAG	Directed Acyclic Graph
M	Module
MG	Module Guide
OS	Operating System
R	Requirement
SC	Scientific Computing
SRS	Software Requirements Specification
Software Engineering	Explanation of program name
UC	Unlikely Change

Contents

List of Tables

List of Figures

3 Introduction

Decomposing a system into modules is a commonly accepted approach to developing software. A module is a work assignment for a programmer or programming team (Parnas et al., 1984). We advocate a decomposition based on the principle of information hiding (Parnas, 1972). This principle supports design for change, because the "secrets" that each module hides represent likely future changes. Design for change is valuable in SC, where modifications are frequent, especially during initial development as the solution space is explored.

Our design follows the rules layed out by Parnas et al. (1984), as follows:

- System details that are likely to change independently should be the secrets of separate modules.
- Each data structure is implemented in only one module.
- Any other program that requires information stored in a module's data structures must obtain it by calling access programs belonging to that module.

After completing the first stage of the design, the Software Requirements Specification (SRS), the Module Guide (MG) is developed (Parnas et al., 1984). The MG specifies the modular structure of the system and is intended to allow both designers and maintainers to easily identify the parts of the software. The potential readers of this document are as follows:

- New project members: This document can be a guide for a new project member to easily understand the overall structure and quickly find the relevant modules they are searching for.
- Maintainers: The hierarchical structure of the module guide improves the maintainers' understanding when they need to make changes to the system. It is important for a maintainer to update the relevant sections of the document after changes have been made.
- Designers: Once the module guide has been written, it can be used to check for consistency, feasibility, and flexibility. Designers can verify the system in various ways, such as consistency among modules, feasibility of the decomposition, and flexibility of the design.

The rest of the document is organized as follows. Section 4 lists the anticipated and unlikely changes of the software requirements. Section 5 summarizes the module decomposition that was constructed according to the likely changes. Section 6 specifies the connections between the software requirements and the modules. Section **??** gives a detailed description of the modules. Section 8 includes two traceability matrices. One checks the completeness of the design against the requirements provided in the SRS. The other shows the relation between anticipated changes and the modules. Section 9 describes the use relation between modules.

4 Anticipated and Unlikely Changes

This section lists possible changes to the system. According to the likeliness of the change, the possible changes are classified into two categories. Anticipated changes are listed in Section 4.1, and unlikely changes are listed in Section 4.2.

4.1 Anticipated Changes

Anticipated changes are the source of the information that is to be hidden inside the modules. Ideally, changing one of the anticipated changes will only require changing the one module that hides the associated decision. The approach adapted here is called design for change.

AC1: Matching Algorithm: The matching algorithm focuses on the goal of finding the best available drivers for the riders. This algorithm takes into account the user's location, timetable, time, and distance. For instance, it matches a rider with a driver that has the same timetable and lives in the same area. The current algorithm is based on these parameters, but it will likely change in the future to consider additional variables to increase efficiency and accuracy. Moreover, subtle changes to the heuristics may be expected for further optimization of complexity and performance.

AC2: User Interface: As the application proceeds through its initial phase of development and use, usability testing is expected to ensure user satisfaction. As users provide feedback, changes will be made to the user interface in future phases. Moreover, changes may be required to meet evolving accessibility standards. The layout and navigation workflows are the primary components of user interaction that are expected to evolve based on user feedback.

AC3: Data Model and Schema: As the application updates with new features, the underlying data structures may change. New parameters would be considered for the matching algorithm, and new entities may be added based on future user needs. Therefore, changes to the persistent storage schema and data definitions are anticipated.

AC4: Payment Processing: Additional payment processors and pricing formulas may be integrated to determine price estimates for drivers. An addition of parameters to the cost estimation logic would cause the underlying algorithms to change.

AC5: Operational Configuration: Changes could be made to the build pipelines and deployment configurations to improve the reliability, stability, and automation of the system.

4.2 Unlikely Changes

The module design should be as general as possible. However, a general system is more complex. Sometimes this complexity is not necessary. Fixing some design decisions at the

system architecture stage can simplify the software design. If these decisions should later need to be changed, then many parts of the design will potentially need to be modified. Hence, it is not intended that these decisions will be changed.

UC1: Target Platform: Hitchly is an application developed specifically for mobile devices. Porting the system to a desktop-native environment is an unlikely change as it would require significant modifications to the user interaction design and location-based services.

UC2: Language Support: The application is primarily based on the English language. Architecting the interface to support multi-language localization would require a major structural design decision.

UC3: Implementation Technologies: The core of this application is developed using a specific set of cross-platform frameworks and typed languages. Changing the fundamental development stack or database engine is an unlikely change as it would require a complete rewrite of the application's codebase.

UC4: Verification Requirement: This application is strictly restricted for users affiliated with the specific institution (McMaster University), as the core domain concept relies on this trusted community. Expanding to the general public is not subject to change. Verification of users will consistently rely on institutional email validation.

5 Module Hierarchy

This section provides an overview of the modular design for Hitchly. The hierarchy follows the principle of information hiding, where each module encapsulates a design decision that may change independently. Only the leaf modules shown in Table 1 are implemented.

5.1 Hardware-Hiding Modules There are no hardware-hiding modules in this design. Hitchly relies on high-level framework APIs (Expo, React Native), so no direct hardware abstraction is required.

5.2 Behaviour-Hiding Modules These modules implement the user-visible behaviour of the application. They hide workflow logic, API interactions, and data handling that support Hitchly's core features.

- M1: Authentication & Verification Module
- M2: User Profile Module
- M3: Route & Trip Module
- M4: Matching Module
- M5: Scheduling Module

- M6: Notification Module
- M7: Rating & Feedback Module
- M8: Safety & Reporting Module
- M9: Payment & Cost Estimation Module
- M10: Admin & Moderation Module

5.3 Software-Decision Modules These modules encapsulate internal logic, algorithms, and implementation decisions not visible to users.

- M11: Pricing Module
- M12: Database Module

Level 1	**Level 2**
Hardware-Hiding Module	None
Behaviour-Hiding Module	M1: Authentication & Verification Module M2: User Profile Module M3: Route & Trip Module M4: Matching Module M5: Scheduling Module M6: Notification Module M7: Rating & Feedback Module M8: Safety & Reporting Module M9: Payment & Cost Estimation Module M10: Admin & Moderation Module
Software-Decision Module	M11: Pricing Module M12: Database Module

Table 1: Module Hierarchy

6 Connection Between Requirements and Design

The design of the system is intended to satisfy the requirements developed in the SRS. In this stage, the system is decomposed into modules. The connection between requirements and modules is listed in Table 2.

7 Authentication & Verification Module

7.1 Module

The Authentication & Verification Module manages user identity creation, secure login, and institutional email verification for all system users. This module is implemented using a secure **Authentication Framework**, which provides end-to-end session management, credential validation, and token-based verification. It ensures that only users with verified institutional emails (e.g., `@mcmaster.ca`) can access the system's features.

Frontend The frontend component is integrated into the client application. It provides the user interface flow for login, signup, and verification. User input is collected through secure forms and transmitted to the backend via remote procedure calls. The frontend handles the following:

- Displays login and signup screens.
- Manages form validation and field state.
- Invokes authentication client hooks for session management.
- Handles redirection and session persistence after successful login.
- Presents error messages for invalid credentials or unverified accounts.

Backend The backend is implemented in the application server using the framework's server-side SDK. It manages credential validation, session tokens, domain enforcement, and email verification logic. Key backend operations include:

- Configuring the identity provider for the API.
- Enforcing allowed email domains during registration.
- Generating and validating secure verification tokens.
- Managing persistent user sessions through secure tokens.
- Integrating with the email service for verification links.

Data The data layer persists authentication and verification information within the relational database managed by the persistence layer. The module automatically provisions and maintains required entities:

- `UserIdentity` — stores credentials, roles, and verification state.
- `SessionStore` — stores active user sessions with expiry timestamps.

- `VerificationTokens` — tracks issued and redeemed email verification tokens.

Data integrity is enforced by unique constraints on email addresses and transactional updates during verification.

7.2 Uses

This module ensures secure access to the system by verifying each user's institutional affiliation. It interacts directly with:

- **Frontend UI:** collects credentials and displays verification flows.
- **API Layer:** provides endpoints for signup, login, logout, and verification.
- **Database:** persists user and session data.

All other modules depend on this component to validate user identity before allowing access to protected features.

7.3 Syntax

Exported Constants

- `ALLOWED_DOMAIN = "@mcmaster.ca"` Restricts signup to allowed institutional domains.
- `SESSION_EXPIRY = 86400` Sets the maximum lifetime (in seconds) for user sessions.

Exported Access Programs

Name	In	Out	Exceptions
`registerUser`	name: string, email: string, password: string	token: string	DuplicateEmail, InvalidDomain
`loginUser`	email: string, password: string	session: string	InvalidCredentials
`verifyEmail`	token: string	success: bool	ExpiredToken
`logoutUser`	session: string	success: bool	InvalidSession

7.4 Semantics

State Variables

- `Users`: Set of `UserRecord` — The collection of all registered users and their states.
- `Sessions`: Set of `SessionRecord` — The collection of currently active sessions.
- `PendingVerifications`: Set of (token: string, email: string) — Map of active verification tokens.

Environment Variables

- User input device (keyboard, network interface).
- Secure transport channel (HTTPS).
- External Email Service (for delivery of verification codes).

Assumptions

- Users possess valid institutional email accounts.
- The persistence layer is initialized and reachable.
- Network connectivity is stable during the authentication handshake.

Access Routine Semantics `registerUser(name, email, password)`:

- **transition:** Adds a new entry to `Users` with `verified = false`. Generates a unique token t and adds $(t, email)$ to `PendingVerifications`.
- **output:** Returns t (to be sent via email).
- **exception:** `DuplicateEmail` if email $\in$ `Users`. `InvalidDomain` if email does not end with `ALLOWED_DOMAIN`.

`loginUser(email, password)`:

- **transition:** Verifies credentials against `Users`. If valid, creates a new session s and adds to `Sessions`.
- **output:** Returns session token s.
- **exception:** `InvalidCredentials` if email not found or password hash mismatch.

`verifyEmail(token)`:

- **transition:** Checks if *token* $\in$ `PendingVerifications`. If found, sets corresponding user in `Users` to `verified = true` and removes token from `PendingVerifications`.
- **output:** `true` on success.
- **exception:** `ExpiredToken` if token not found or time-to-live exceeded.

`logoutUser(session)`:

- **transition:** Removes *session* from `Sessions`.
- **output:** `true`.
- **exception:** `InvalidSession` if session is not active.

Local Functions

- `generateVerificationToken(): string` — Generates a cryptographically secure random string.
- `validateEmailDomain(email: string): boolean` — Returns (*email* endsWith `ALLOWED_DOMAIN`
- `hashPassword(password: string): string` — Applies a one-way cryptographic hash function to the input.

8 Traceability Matrix

This section shows two traceability matrices: between the modules and the requirements and between the modules and the anticipated changes.

Req.	Modules
FR211	M4, M3, M5, M10
FR212	M4, M5, M10
FR213	M4, M2
FR214	M4, M2, M5
NFR211	M4, M10
NFR212	M4, M1, M10
FR221	M5, M3, M10
FR222	M5, M6
FR223	M5, M6, M9
FR224	M6, M5
NFR221	M10, M5
NFR222	M6, M5, M10
FR231	M1, M2
FR232	M1, M10
FR233	M8, M3, M6
FR234	M8, M2, M12
NFR231	M1, M10
NFR232	M1, M10, M12
FR241	M11, M5, M3
FR242	M9
FR243	M9
FR244	M10, M9
NFR241	M9
NFR242	M9
FR251	M2, M1
FR252	M10, M2
FR253	M10, M2, M3
FR254	M2, M4
NFR251	M2, M1
NFR252	M2, M10
FR261	M7, M2
FR262	M8, M12
FR263	M7, M5, M4
FR264	M7, M2
NFR261	M7, M12

Table 2: Trace Between Requirements and Modules

AC	Modules
AC1	M4, M2, M5
AC2	M1, M2, M3, M4, M5, M6, M7, M8, M9, M11, M12
AC3	M10
AC4	M9, M11
AC5	N/A (External Process)

Table 3: Trace Between Anticipated Changes and Modules

9 Use Hierarchy Between Modules

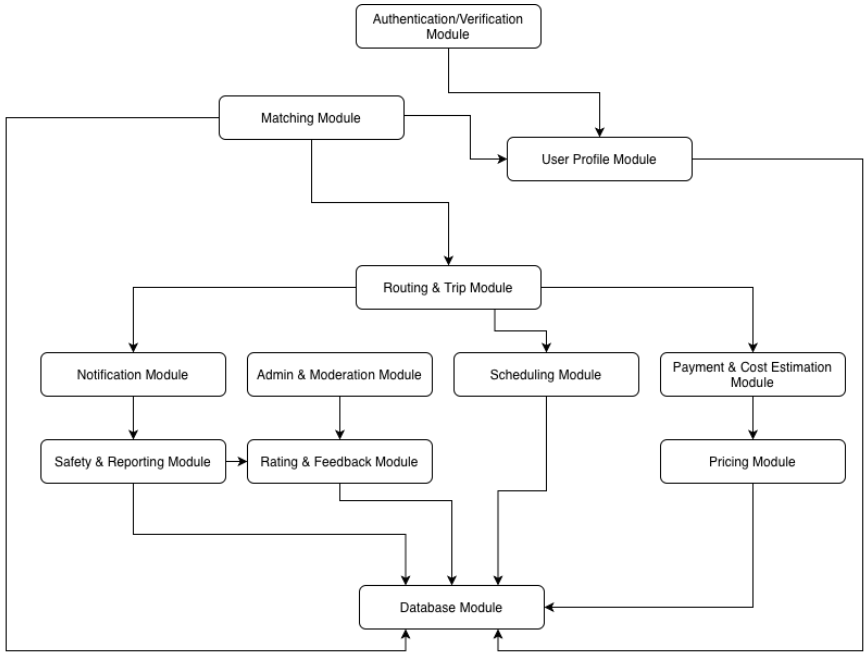

Figure 1: Use hierarchy among modules

10 User Interfaces

This section provides an overview of the important screens of the application.

Figure 2: Main Login Screen

Figure 3: Register Screen

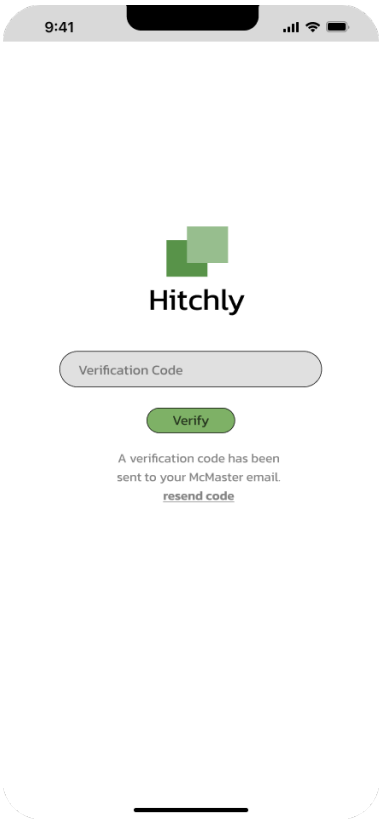

Figure 4: Verification Screen

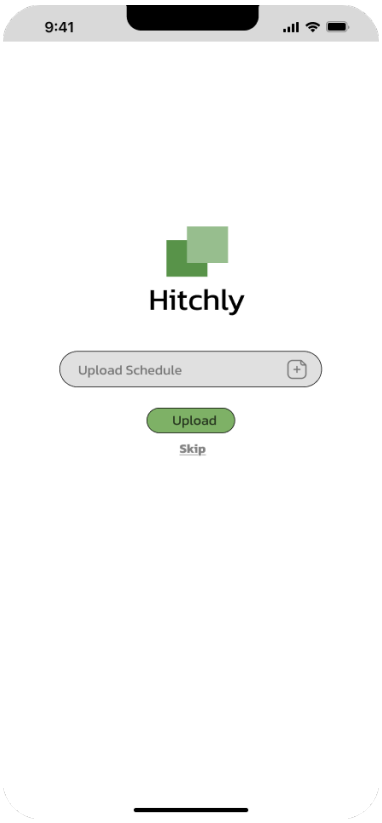

Figure 5: Upload Schedule Screen

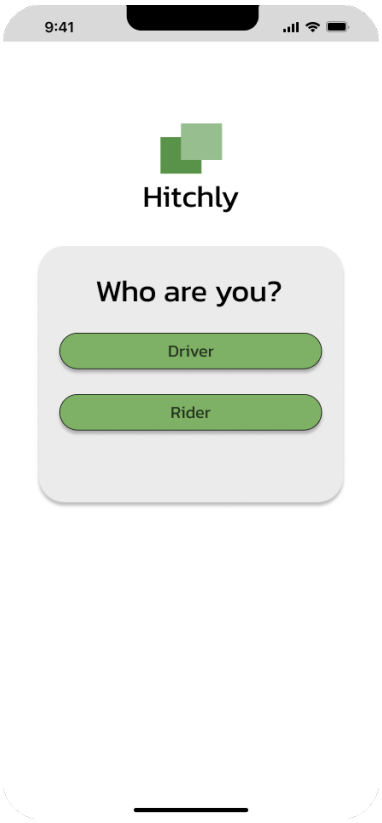

Figure 6: Select User Type Screen

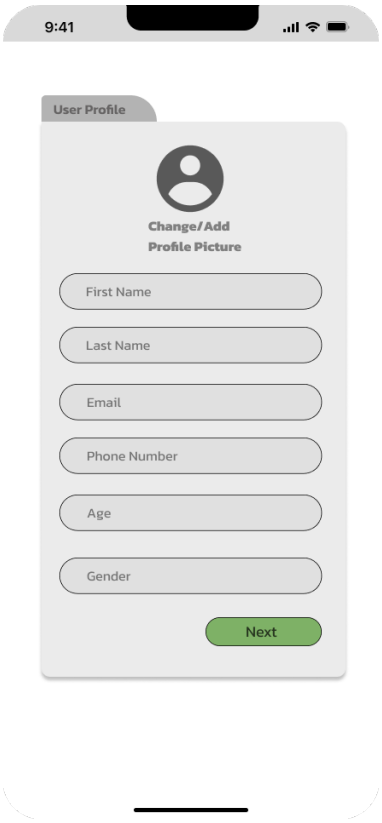

Figure 7: User Prompted to Add Profile Information

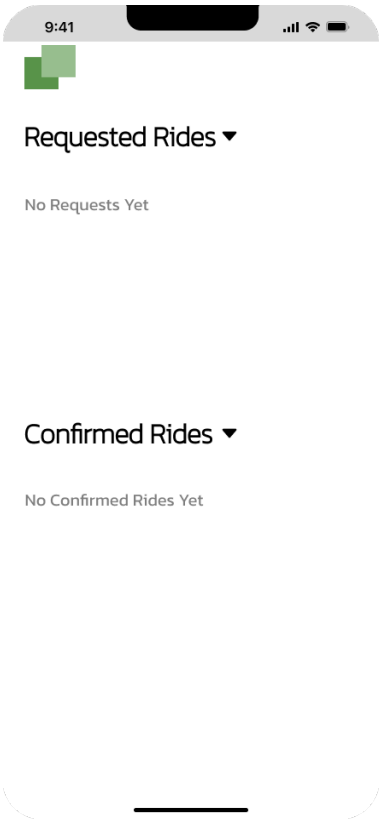

Figure 8: Home Screen

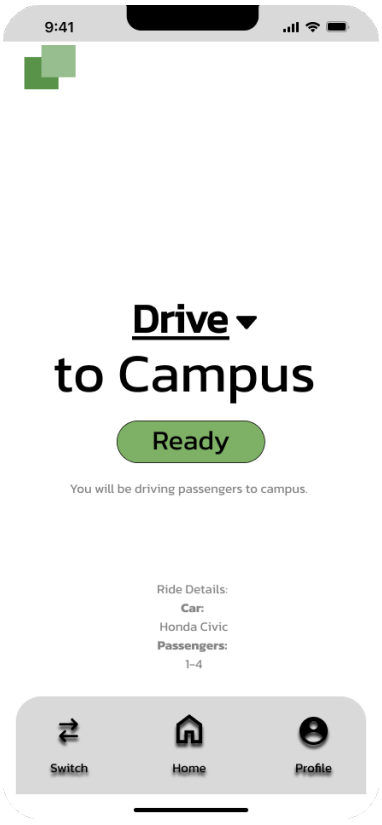

Figure 9: Switch to Driver Screen

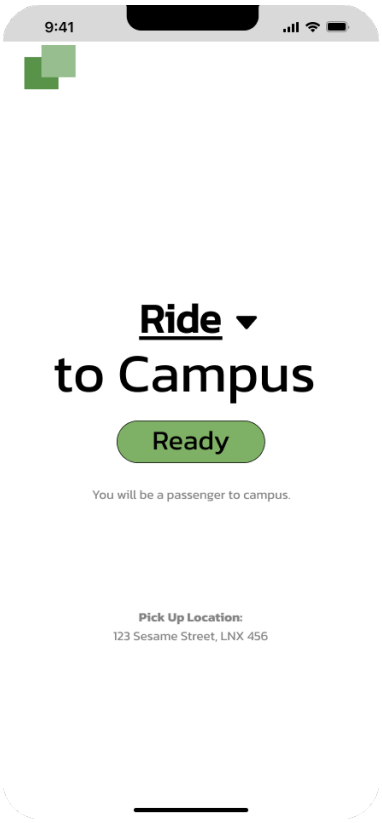

Figure 10: Switch to Rider Screen

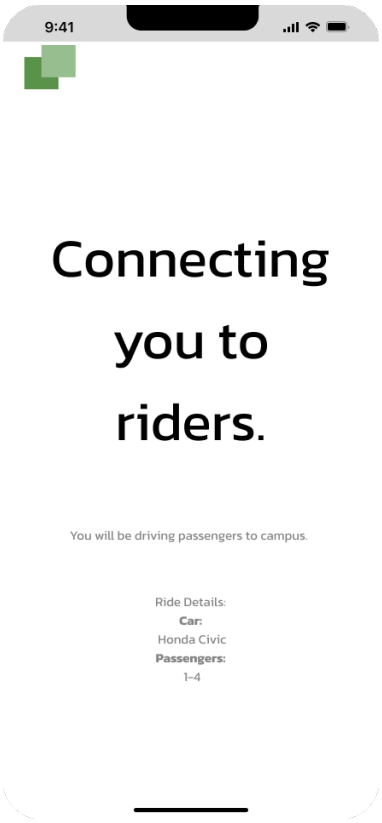

Figure 11: Driver Connection Screen

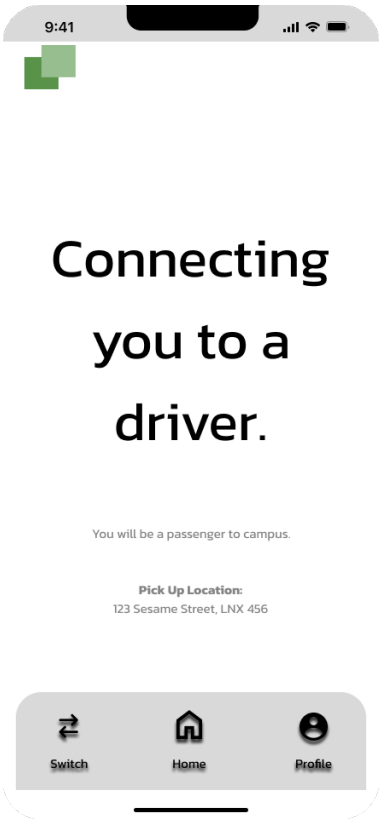

Figure 12: Rider Connection Screen

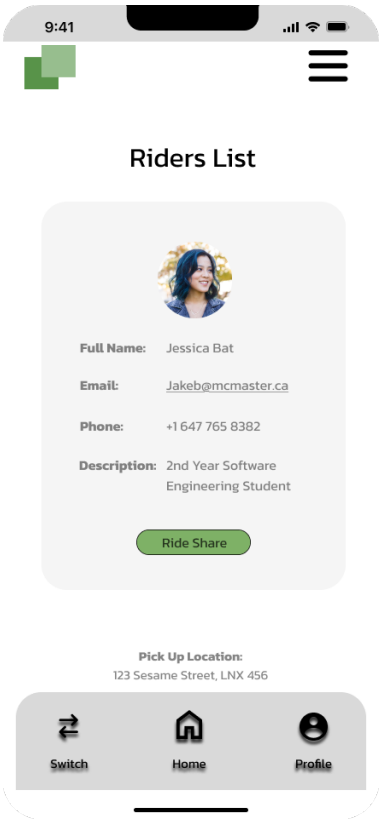

Figure 13: Matched Rider List Screen

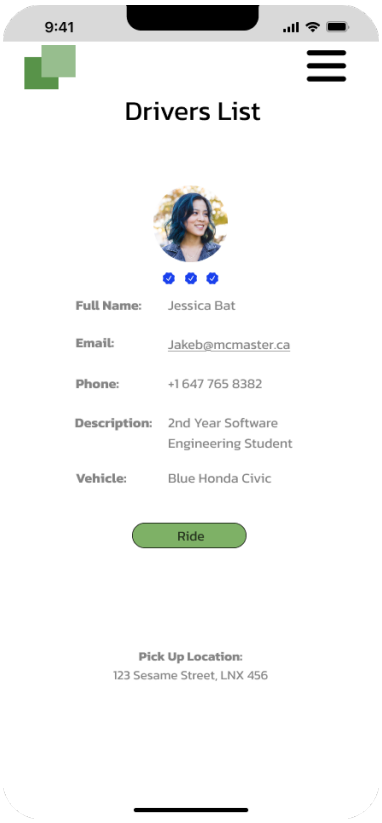

Figure 14: Matched Driver List Screen

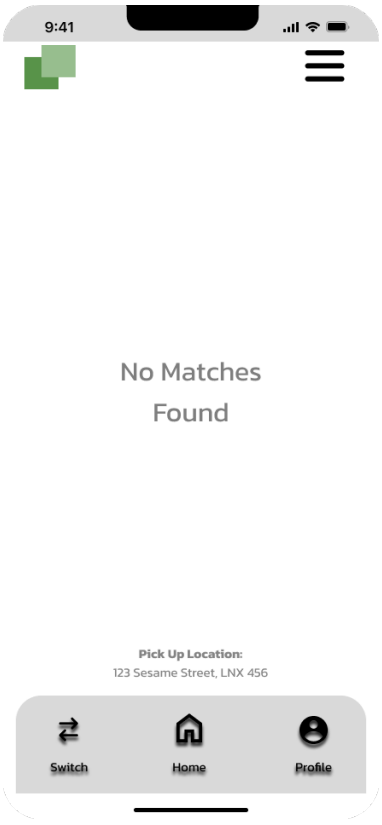

Figure 15: No Matches Found Screen

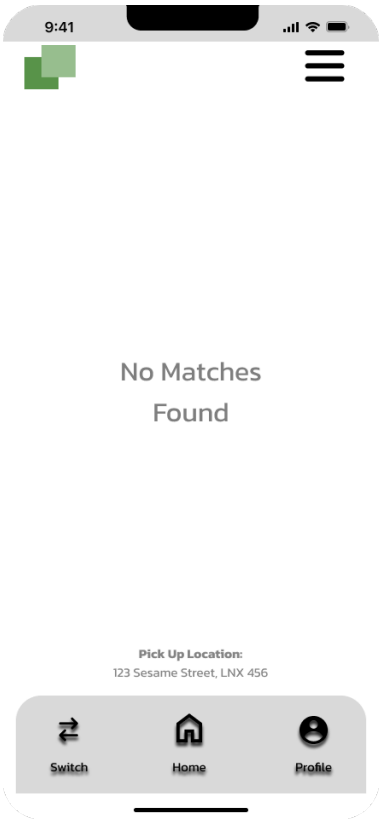

Figure 16: No Matches Found Screen

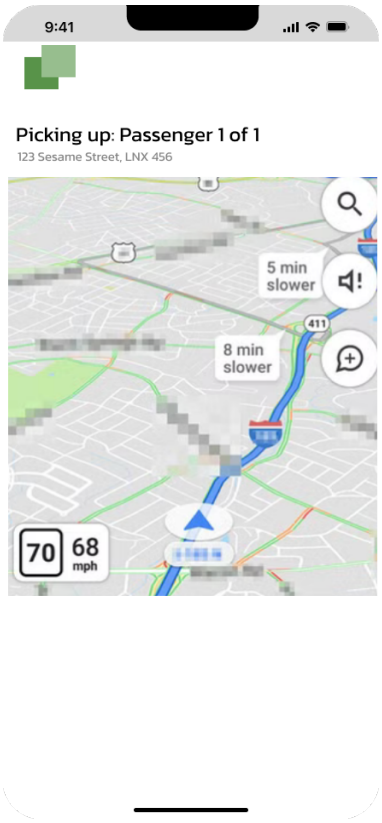

Figure 17: Driver Navigation Screen

Figure 18: Rider Wait Screen

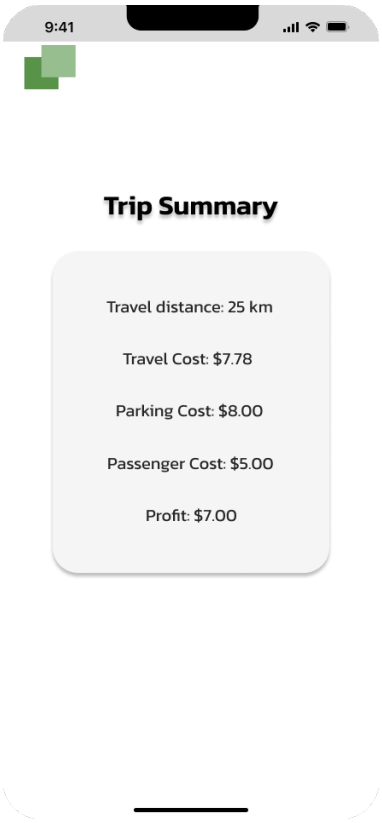

Figure 19: Cost Estimation Screen

11 Design of Communication Protocols

This section describes the design of the communication protocols implemented in Hitchly. Communication occurs primarily between the client-side mobile application and the backend server infrastructure. The system also integrates third-party interfaces for geolocation and external notification services.

11.1 11.1 Overall Architecture

All communication between the mobile client and the backend follows a request–response model over secure transport channels, ensuring confidentiality and integrity. Data is serialized into a structured format and transferred through defined API endpoints. The architecture adopts a modular service-based design to support scalability and maintainability:

- **Frontend (Client):** Mobile application interface that sends authenticated requests to backend endpoints via secure transport protocols.
- **Backend (Server):** Server-side application exposing routes through an API layer. It validates and processes incoming requests, interacts with the persistence layer, and

returns structured responses.

- **External Services:** Interfaces for mapping and geolocation data, push notification services for alerts, and identity verification providers (e.g., institutional SSO for email validation).

11.2 11.2 Communication Flow

A typical data exchange involves the following sequence:

1. The user initiates an action in the mobile application (e.g., requests a ride, updates schedule, or rates a driver).
2. The frontend constructs a remote procedure call encapsulating the action data and attaches the user's authentication credentials in the header.
3. The server authenticates the request, validates input schemas against defined contracts, and executes the corresponding service logic.
4. The backend queries or updates the persistence layer and returns a success or error payload to the client.
5. The client updates its local state and user interface based on the received data.

11.3 11.3 API Endpoints and Message Types

Hitchly's backend exposes the following logical API groups:

- **Auth API:** Handles user registration, login, institutional email verification, and session management. Exchanges encrypted credentials and session tokens.
- **User API:** Manages user profiles, schedules, and preferences.
- **Matching API:** Accepts commute parameters, executes the matching algorithm, and returns ranked ride offers.
- **Trip API:** Records completed trips, generates summaries, and calculates cost-sharing.
- **Rating API:** Submits and retrieves reputation scores and text reviews.
- **Notification API:** Interfaces with external push services to deliver trip confirmations, cancellations, and real-time updates.

All messages follow a standardized structure containing the operation status, payload data, and metadata:

```
{
  "status": "success",
  "data": { ... },
  "timestamp": "YYYY-MM-DDTHH:MM:SSZ"
}
```

11.4 11.4 Security and Reliability

Security and reliability are ensured through:

- **Transport Security:** Encryption in transit using TLS standards.
- **Authentication:** Bearer tokens or equivalent session credentials included in headers of all authorized requests.
- **Input Validation:** All incoming payloads are validated against strict schemas to prevent injection attacks or malformed data.
- **Error Handling:** Standardized error codes with human-readable messages returned to the client.
- **Rate Limiting:** Middleware limits excessive requests to prevent abuse and ensure service availability.
- **Retry Mechanism:** Client-side logic automatically retries transient network failures to improve resilience.

11.5 11.5 Design Considerations

The communication design of Hitchly emphasizes the following principles:

- **Consistency:** Shared type definitions across frontend and backend guarantee end-to-end type safety and contract adherence.
- **Extensibility:** Additional endpoints (e.g., payment interfaces or advanced analytics) can be introduced without modifying existing client modules.
- **Scalability:** The stateless architectural design supports horizontal scaling of the application server.
- **Low Latency:** Lightweight message serialization and asynchronous notification delivery minimize perceived user delay.

This design ensures secure, efficient, and maintainable communication between Hitchly's mobile application, backend, and external services, supporting its goals of safety, reliability, and sustainability for commuters.

12 Timeline

The following timeline details our implementation for Rev 0. Each task is broken down to the module level. Each tasks has roles and responsibilities defined along with estimated development duration.

Table 4: Phase 1

Task	**Module(s)**	**Status**	**Responsible**
Creating Hitchly Wireframes for UI	M2	Done	Burhan, Sarim
Make initial version of matching algorithm	M4	Done	Hamzah, Sarim
Set up authentication and application bones	M1	Done	Aidan

Table 5: Phase 2: Front End and User Roles

Task	**Module(s)**	**Status**	**Responsible**
Develop login, registration, and session flows	M1	Weeks 11–12	Aidan
Build user profile screens and preferences interface	M2	Weeks 11–12	Swesan
Create upload, settings, and account management pages	M2	Weeks 11–12	Burhan
Build navigation, layouts, and key frontend components	M2	Weeks 12-13	Hamzah
Integrate basic user-to-database connections for profile initialization	M12	Weeks 12–13	Sarim

Table 6: Phase 3: Core Ride-Share Logic and Trip Flow

Task	**Module(s)**	**Status**	**Responsible**
Implement matching algorithm and rider-driver compatibility logic	M4	Weeks 13-14	Sarim, Hamzah
Build Routing & Trip creation (origin, destination, seat count)	M3	Weeks 13-14	Burhan
Implement Scheduling module (recurring rides, time windows)	M5	Weeks 13-14	Aidan
Integrate Payment & Cost estimation (distance, time, pricing model)	M9	Weeks 13-14	Swesan
Build Pricing module (fare calculations, dynamic pricing rules)	M11	Weeks 13-14	Swesan
Add Rating & Feedback system for both riders and drivers	M7	Weeks 13-14	Sarim
Implement Admin & Moderation tools (flagging, dispute handling)	M10	Weeks 13-14	Burhan, Aidan
Build Notification module (push/in-app updates for trips)	M6	Weeks 13-14	Hamzah

Table 7: Phase 4: Safety, Database Integration

Task	**Module(s)**	**Status**	**Responsible**
Implement Safety & Reporting module (incident reports, flagging)	M8	Weeks 14–15	Burhan
Complete full Database Module integration across all features	M12	Weeks 14–15	Aidan, Sarim
Perform end-to-end testing of all ride flows	M1–M12	Weeks 14–15	All
Optimize performance, resolve bugs, and finalize UI polish	M1–M12	Weeks 14–15	All

Table 8: Phase 5: Testing, and Finalization

Task	**Module(s)**	**Status**	**Responsible**
Module Testing & Bug Fixing for Safety	M8	Weeks 15–16	Burhan
Perform integration testing for Database Module	M12	Weeks 15–16	Aidan, Sarim
Perform integration testing for frontend and backend	M1–M12	Weeks 15–17	All
Perform regression testing and Finalize UI/UX	M1–M12	Weeks 15–17	All
Prepare release build, documentation, and deployment setup	M1–M12	Weeks 18-19	All

References

David L. Parnas. On the criteria to be used in decomposing systems into modules. *Comm. ACM*, 15(2):1053–1058, December 1972.

D.L. Parnas, P.C. Clement, and D. M. Weiss. The modular structure of complex systems. In *International Conference on Software Engineering*, pages 408–419, 1984.