Module Interface Specification for Software Engineering

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

January 21, 2026

1 Revision History

Date	Developer	Notes
November 7st	Aidan Froggatt	Completed Section 6.1, 6.2, 6.3, 6.4
November 11th	Swesan Pathmanathan	Completed Sections 1, 2, 4, 5, 6.5, 6.6, 6.7, 6.8
November 12th	Burhan Kharodawala	Completed Sections 3, 6.9, 6.10, 6.11, 6.12

2 Symbols, Abbreviations and Acronyms

See SRS Documentation at https://github.com/hitchly/hitchly/tree/main/docs/SRS

symbol	**description**
API	Application Programming Interface
CRUD	Create, Read, Update, Delete
DB	Database
JWT	JSON Web Token
ORM	Object-Relational Mapping
UI	User Interface
UX	User Experience
tRPC	TypeScript Remote Procedure Calls
SSO	Single Sign-On
SDK	Software Development Kit
GPS	Global Positioning System
MFA	Multi-Factor Authentication
OTP	One-Time Password
HTTP	Hypertext Transfer Protocol
HTTPS	Secure Hypertext Transfer Protocol
Expo	React Native toolchain for mobile applications
Stripe	Payment gateway service
SQL	Structured Query Language
DBMS	Database Management System
ML	Machine Learning
Software Engineering	Hitchly rideshare system name

Contents

1 Revision History **1**

2 Symbols, Abbreviations and Acronyms **2**

3 Introduction **1**

4 Notation **1**

5 Module Decomposition **3**

6 Authentication & Verification Module **7**
- 6.1 Module 7
- 6.2 Uses 8
- 6.3 Syntax 8
- 6.4 Semantics 9

7 User Profile Module **10**
- 7.1 Module 10
- 7.2 Uses 12
- 7.3 Syntax 12
- 7.4 Semantics 12

8 Route & Trip Module **14**
- 8.1 Module 14
- 8.2 Uses 15
- 8.3 Syntax 15
- 8.4 Semantics 16

9 Matching Module **18**
- 9.1 Module 18
- 9.2 Formal Specification 19
  - 9.2.1 State Definitions 20
  - 9.2.2 Mathematical Definitions 22
- 9.3 Uses 24
- 9.4 Syntax 24
- 9.5 Semantics 25

10 Scheduling Module **30**
- 10.1 Module 30
- 10.2 Uses 31
- 10.3 Syntax 32
- 10.4 Semantics 32

11 Notification Module **34**
11.1 Module . . . 34
11.2 Uses . . . 35
11.3 Syntax . . . 35
11.4 Semantics . . . 35

12 Rating & Feedback Module **35**
12.1 Module . . . 35
12.2 Uses . . . 36
12.3 Syntax . . . 36
12.4 Semantics . . . 37

13 Safety & Reporting Module **37**
13.1 Module . . . 37
13.2 Uses . . . 38
13.3 Syntax . . . 38
13.4 Semantics . . . 38

14 Payment Module **38**
14.1 Module . . . 38
14.2 Uses . . . 39
14.3 Syntax . . . 39
14.4 Semantics . . . 40

15 Database Module **41**
15.1 Module . . . 41
15.2 Uses . . . 41
15.3 Syntax . . . 42
15.4 Semantics . . . 42

16 Pricing Module **44**
16.1 Module . . . 44
16.2 Uses . . . 44
16.3 Syntax . . . 44
16.4 Semantics . . . 45

17 Admin/Moderation Module **46**
17.1 Module . . . 46
17.2 Uses . . . 46
17.3 Syntax . . . 46
17.4 Semantics . . . 46

18 Appendix **47**

3 Introduction

The following document details the Module Interface Specifications for Hitchtly, a rideshare application designed for the McMaster community. Its main purpose is to provide a reliable and trustworthy platform for McMaster students, staff and faculty members to find and provide rideshare services. It does it through its robust matching algorithm which matches users based on their timetable, location time, and preferences. It aims to make commuting to and from university sustainable and cost-effective.
Complementary documents include the System Requirement Specifications and Module Guide. The full documentation and implementation can be found at .

4 Notation

The structure of the MIS for modules comes from Hoffman and Strooper (1995), with the addition that template modules have been adapted from Ghezzi et al. (2003). The mathematical notation comes from Chapter 3 of Hoffman and Strooper (1995). For instance, the symbol := is used for a multiple assignment statement and conditional rules follow the form $(c_1 \Rightarrow r_1 | c_2 \Rightarrow r_2 | ... | c_n \Rightarrow r_n)$.

The following table summarizes the primitive data types used by Software Engineering.

Data Type	**Notation**	**Description**
character	char	a single symbol or digit
integer	$\mathbb{Z}$	a number without a fractional component in (-∞, ∞)
natural number	$\mathbb{N}$	a number without a fractional component in [1, ∞)
real	$\mathbb{R}$	any number in (-∞, ∞)

The specification of Software Engineering uses some derived data types: sequences, strings, and tuples. Sequences are lists filled with elements of the same data type. Strings are sequences of characters. Tuples contain a list of values, potentially of different types. In addition, Software Engineering uses functions, which are defined by the data types of their inputs and outputs. Local functions are described by giving their type signature followed by their specification.
The following enumerated and composite types are used throughout the specification:

- **Role** = {rider, driver, both} — User role enumeration
- **Day** = {Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday} — Day of week enumeration
- **TripStatus** = {pending, active, completed, canceled} — Trip state enumeration

- **SwipeType** = {like, pass} — Swipe action enumeration
- **Location** = (latitude: $\mathbb{R}$, longitude: $\mathbb{R}$) — Geographic coordinates
- **UserRecord** = (userId: $\mathbb{N}$, name: string, email: string, role: Role, faculty: string, year: $\mathbb{N}$, bio: string, profileImageUrl: string, verified: bool) — Complete user profile
- **UpdatedRecord** = UserRecord — Updated user profile (same structure as UserRecord)
- **PreferenceData** = (userId: $\mathbb{N}$, quietRide: bool, musicPreference: string, smokingAllowed: bool, maxDetour: $\mathbb{R}$) — User ride preferences
- **UpdatedPreference** = PreferenceData — Updated preferences (same structure as PreferenceData)
- **TripRecord** = (tripId: $\mathbb{N}$, driverId: $\mathbb{N}$, origin: Location, destination: Location, departureTime: $\mathbb{R}$, availableSeats: $\mathbb{N}$, status: TripStatus) — Complete trip information
- **TripList** = seq of TripRecord — Sequence of trip records
- **TripData** = (origin: Location, destination: Location, departureTime: $\mathbb{R}$, availableSeats: $\mathbb{N}$) — Trip creation input
- **UpdatedFields** = (origin?: Location, destination?: Location, departureTime?: $\mathbb{R}$, availableSeats?: $\mathbb{N}$) — Optional trip update fields (all fields optional)
- **UpdatedTripRecord** = TripRecord — Updated trip record (same structure as TripRecord)
- **ScheduleRecord** = (scheduleId: $\mathbb{N}$, userId: $\mathbb{N}$, daysOfWeek: seq of Day, time: $\mathbb{R}$, origin: Location, destination: Location, isRecurring: bool) — Recurring schedule definition
- **ScheduleList** = seq of ScheduleRecord — Sequence of schedule records
- **scheduleData** = (daysOfWeek: seq of Day, time: $\mathbb{R}$, origin: Location, destination: Location, isRecurring: bool) — Schedule creation input
- **MatchRecord** = (matchId: $\mathbb{N}$, riderId: $\mathbb{N}$, driverId: $\mathbb{N}$, tripId: $\mathbb{N}$, score: $\mathbb{R}$, timestamp: $\mathbb{R}$) — Match result with compatibility score
- **RankedMatchList** = seq of MatchRecord — Sequence of match records ordered by score (descending)
- **SwipeRecord** = (swipeId: $\mathbb{N}$, userId: $\mathbb{N}$, targetId: $\mathbb{N}$, swipeType: SwipeType, timestamp: $\mathbb{R}$) — User swipe action record

- **MatchSummary** = (mutualMatches: seq of MatchRecord, pendingSwipes: seq of SwipeRecord) — Complete match status for a user
- **Confirmation** = bool — Operation success confirmation

5 Module Decomposition

Modules are decomposed according to the principle of information hiding (Parnas et al., 1984). Each module's *Secrets* describes the internal design decision that is intentionally hidden from other modules, while the *Services* field specifies *what* the module provides without revealing *how* the service is implemented. The *Frontend UI*, *API Logic*, and *Database Models* indicate how each module is realized across the system's architecture. *Implemented By* identifies the technologies used, and *Type of Module* classifies each module as a Behaviour-Hiding or Software-Decision module according to the Module Hierarchy (Section 5). Only the leaf modules are included.

Behaviour-Hiding Modules

M1: Authentication & Verification Module

- **Secrets:** Token/session strategy, email-verification workflow, and credential-validation logic.
- **Services:** Allows users to register, log in, and verify McMaster email accounts.
- **Frontend UI:** Login screens, signup forms, verification code UI.
- **API Logic:** `authRouter` (login, register, verify), session helpers, Zod validation.
- **Database Models:** `User` table (email, password hash, verified flag, session tokens).
- **Implemented By:** React Native (Expo) + tRPC backend with Prisma ORM.
- **Type of Module:** Behaviour-Hiding Module.

M2: User Profile Module

- **Secrets:** Role-assignment logic and profile-preference storage design.
- **Services:** Provides profile editing, preferences, and vehicle data management.
- **Frontend UI:** Profile page, edit forms, vehicle info inputs.
- **API Logic:** `userRouter` for CRUD operations, preference update flow.
- **Database Models:** `User`, `Vehicle`, `Preference`.

- **Implemented By:** React Native UI + tRPC + Prisma.
- **Type of Module:** Behaviour-Hiding Module.

M3: Route & Trip Module

- **Secrets:** Trip-storage schema, route normalization, and time formatting.
- **Services:** Allows users to create, list, and cancel trips.
- **Frontend UI:** Trip creation form, trip list UI, route selection screen.
- **API Logic:** `tripRouter` for trip creation, querying, and deletion.
- **Database Models:** `Trip` table (origin, destination, time, seat count).
- **Implemented By:** React Native + tRPC backend + Prisma.
- **Type of Module:** Behaviour-Hiding Module.

M4: Matching Module

- **Secrets:** Scoring algorithm, distance/time weighting, and Strategy Pattern implementation.
- **Services:** Computes driver–rider compatibility and returns ordered match results.
- **Frontend UI:** Swipe-based card UI, match results list.
- **API Logic:** `matchmakingRouter`, MatchEngine (Strategy Pattern).
- **Database Models:** `Match`, `Swipe`.
- **Implemented By:** Node.js backend (tRPC) + algorithm utilities + Prisma ORM.
- **Type of Module:** Behaviour-Hiding Module.

M5: Scheduling Module

- **Secrets:** Recurring-trip generation algorithm and time-window parsing.
- **Services:** Creates recurring or one-time schedules linked to trips.
- **Frontend UI:** Time picker, recurring toggle, calendar UI.
- **API Logic:** Schedule parser and recurring schedule generator.
- **Database Models:** `Schedule` table.
- **Implemented By:** tRPC backend + Prisma.
- **Type of Module:** Behaviour-Hiding Module.

M6: Notification Module

- **Secrets:** Push-token handling and asynchronous event queue logic.
- **Services:** Sends push notifications for matches, cancellations, and reminders.
- **Frontend UI:** In-app notification center, Expo push integration.
- **API Logic:** Notification service with event emitters.
- **Database Models:** `Notification` table.
- **Implemented By:** Expo Push Service + tRPC backend dispatcher.
- **Type of Module:** Behaviour-Hiding Module.

M7: Rating & Feedback Module

- **Secrets:** Reputation computation and weighting logic.
- **Services:** Enables users to submit ratings and text feedback after trips.
- **Frontend UI:** Post-ride rating screen.
- **API Logic:** `ratingRouter`.
- **Database Models:** `Rating` table (linked to `Match`).
- **Implemented By:** React Native + tRPC + Prisma ORM.
- **Type of Module:** Behaviour-Hiding Module.

M8: Safety & Reporting Module

- **Secrets:** Incident-flagging thresholds and safety-response workflow.
- **Services:** Allows reporting of unsafe behaviour and stores incidents for admin review.
- **Frontend UI:** Report button, safety resources screen.
- **API Logic:** Report submission via `reportRouter`.
- **Database Models:** `Report`.
- **Implemented By:** React Native + tRPC + Prisma.
- **Type of Module:** Behaviour-Hiding Module.

M9: Payment & Cost Estimation Module

- **Secrets:** Fare-calculation logic and mock-payment validation.
- **Services:** Estimates ride cost and records payment confirmations.
- **Frontend UI:** Fare display, confirmation screen.
- **API Logic:** `paymentRouter`.
- **Database Models:** `Payment`, `Transaction`.
- **Implemented By:** tRPC backend + Prisma ORM.
- **Type of Module:** Behaviour-Hiding Module.

M10: Admin & Moderation Module

- **Secrets:** Moderation rules, flagging thresholds, and admin-only access logic.
- **Services:** Allows admins to review reports, ban users, and view platform stats.
- **Frontend UI:** Admin dashboard (optional).
- **API Logic:** `adminRouter`.
- **Database Models:** Admin logs, flagged users.
- **Implemented By:** tRPC backend + Prisma.
- **Type of Module:** Behaviour-Hiding Module.

Software-Decision Modules

M11: Pricing Module

- **Secrets:** Fuel-rate constants, distance multipliers, and fare model.
- **Services:** Computes estimated trip price using distance and conditions.
- **Frontend UI:** Fare estimate displayed in trip preview.
- **API Logic:** Cost-calculation service reused by M9.
- **Database Models:** Pricing configuration constants.
- **Implemented By:** Node.js backend utilities + shared pricing helpers.
- **Type of Module:** Software-Decision Module.

M12: Database Module

- **Secrets:** ORM mapping, schema decisions, and migration strategy.
- **Services:** Provides access to all persistent data models via a centralized DB client.
- **Frontend UI:** None.
- **API Logic:** Prisma client configuration and model exports.
- **Database Models:** All tables (User, Trip, Match, Schedule, etc.).
- **Implemented By:** Prisma ORM + PostgreSQL.
- **Type of Module:** Software-Decision Module.

6 Authentication & Verification Module

6.1 Module

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

Frontend The frontend component is integrated into the Expo mobile application. It provides the UI flow for login, signup, and verification. User input is collected through secure forms and sent to the backend via tRPC. The frontend handles the following:

- Displays login and signup screens.
- Manages form validation and field state.
- Invokes BetterAuth client hooks for authentication (e.g., `useSignIn`, `useSignUp`).
- Handles redirect and session persistence after successful login.
- Presents error messages for invalid credentials or unverified accounts.

Backend The backend is implemented in the Express/tRPC API using **BetterAuth's server SDK**. It manages credential validation, session tokens, McMaster domain enforcement, and email verification logic. Key backend operations include:

- Configuring BetterAuth provider for the Hitchly API.
- Enforcing allowed email domains (`@mcmaster.ca`) during registration.

- Generating and validating secure verification tokens.
- Managing persistent user sessions through signed cookies and JWTs.
- Integrating with the email service for verification links.

Data The data layer persists authentication and verification information within the PostgreSQL database managed by Drizzle ORM. BetterAuth automatically provisions and maintains required tables:

- `users` — stores user credentials, roles, and verification state.
- `sessions` — stores active user sessions with expiry timestamps.
- `verification_tokens` — tracks issued and redeemed email verification tokens.

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

6.2 Uses

This module ensures secure access to Hitchly by verifying each user's McMaster University 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 managed through BetterAuth.

All other modules depend on this component to validate user identity before allowing access to trip creation, matching, and messaging features.

6.3 Syntax

Exported Constants

- `ALLOWED_DOMAIN = "@mcmaster.ca"` Restricts signup to McMaster-affiliated emails.
- `SESSION_EXPIRY = 24h` Sets the maximum lifetime for user sessions.

Exported Access Programs

Name	**In**	**Out**	**Exceptions**
`registerUser()`	name, email, password	verificationToken	DuplicateEmailError
`loginUser()`	email, password	sessionToken	InvalidCredentialsError
`verifyEmail()`	verificationToken	successFlag	ExpiredTokenError
`logoutUser()`	sessionToken	confirmation	InvalidSessionError

6.4 Semantics

State Variables

- `isVerified: bool` — indicates whether the user's email has been confirmed.
- `authToken: string` — stores the active session token generated by BetterAuth.
- `sessionState: object` — holds session metadata for frontend persistence.

Environment Variables

- User device input (keyboard, network, mobile UI).
- Secure HTTPS connection to API.
- Email server or transactional service for verification emails.

Assumptions

- All users have valid McMaster email accounts.
- Database migrations and BetterAuth setup scripts have been executed.
- Network connectivity is stable during registration or verification.

Access Routine Semantics `registerUser()`:

- **transition:** Inserts a new pending user into the database and triggers a verification email.
- **output:** Returns a token or link for verification.
- **exception:** Thrown if email already exists or invalid domain detected.

`loginUser()`:

- **transition:** Validates credentials and creates a session.
- **output:** Returns a signed JWT or session cookie.
- **exception:** InvalidCredentialsError if authentication fails.

`verifyEmail()`:

- **transition:** Updates user's `isVerified` state to true.
- **output:** Returns success confirmation.
- **exception:** ExpiredTokenError if verification link invalid or expired.

`logoutUser()`:

- **transition:** Revokes session in `sessions` table.
- **output:** Confirmation of logout.
- **exception:** InvalidSessionError if session token not found.

Local Functions

- `generateVerificationToken(): string` — Creates a cryptographically secure token.
- `validateEmailDomain(email: string): boolean` — Ensures domain ends with `@mcmaster.ca`.
- `hashPassword(password: string): string` — Applies secure hashing for stored credentials.

7 User Profile Module

7.1 Module

The User Profile Module manages all personal and contextual information about a Hitchly user. It allows students, alumni, and faculty to view and edit their personal data, preferences, and role (rider, driver, or both). This module integrates directly with the Authentication & Verification Module for identity linkage and with the Matching Module to supply accurate data for matchmaking.

Frontend The frontend portion resides in the Expo mobile application. It provides interfaces for profile creation, editing, and display, implemented using shared UI components. Key features include:

- Editable profile screen showing name, faculty, year, and role.
- Driver-specific section for vehicle information (make, model, seats).
- Preference selection for quiet/chatty rides, music, and other comfort options.
- Integration with camera/gallery for optional profile photo upload.
- Form validation and error messaging for incomplete or invalid fields.
- Communication with backend via tRPC hooks (e.g., `useGetUser`, `useUpdateUser`).

Backend The backend implements the business logic and data orchestration using Express and tRPC. It exposes endpoints for retrieving, updating, and deleting user data. Main responsibilities include:

- Providing tRPC procedures such as `getUserProfile`, `updateUserProfile`, and `deleteUser`.
- Validating incoming payloads using Zod schemas for type safety.
- Enforcing access control through session validation from BetterAuth.
- Maintaining referential integrity with linked tables (e.g., trips, ratings).
- Broadcasting profile updates to dependent modules (e.g., Matching, Analytics).

Data The data layer defines persistent entities stored in PostgreSQL via Drizzle ORM. Key tables include:

- `users` — stores user metadata such as name, role, faculty, and profile image URL.
- `vehicles` — stores driver vehicle details and capacity.
- `preferences` — stores personal ride preferences and settings.

All tables use foreign-key constraints to maintain one-to-one or one-to-many relationships with user records. Schema migrations ensure consistent structure across environments.

7.2 Uses

This module is used to manage user data required across the entire Hitchly system. It interacts with:

- **Authentication & Verification Module** - links verified user identity to profile records.
- **Matching Module** - provides profile and preference data for scoring algorithms.
- **Trip Management Module** - associates user profiles with created or joined trips.
- **Rating & Feedback Module** - aggregates ride feedback to display reliability metrics.

Through these integrations, the module forms the foundation of personalization and trust within Hitchly.

7.3 Syntax

Exported Constants

- `MAX_BIO_LENGTH = 250` - Limits user biography text.
- `DEFAULT_ROLE = "rider"` - Assigned when a user first registers.

Exported Access Programs

Name	In	Out	Exceptions
`getUserProfile()`	userId	UserRecord	NotFoundError
`updateUserProfile()`	userId, ProfileData	UpdatedRecord	ValidationError
`deleteUser()`	userId	Confirmation	UnauthorizedError
`getUserPreferences()`	userId	PreferenceData	NotFoundError
`updatePreferences()`	userId, PreferenceData	UpdatedPreference	ValidationError

7.4 Semantics

State Variables

- `userProfile` - Stores the current user information in application state.
- `preferences` - Represents the user's ride comfort and behavior choices.
- `vehicleData` - Contains driver-specific car details for eligible users.

Environment Variables

- Mobile UI components and device storage for temporary profile caching.
- Active API connection through HTTPS/tRPC.
- Database connection via Drizzle ORM.

Assumptions

- The user is authenticated and verified before modifying profile data.
- The database schema has been migrated to include required tables.
- Frontend validation is performed prior to API submission.

Access Routine Semantics `getUserProfile()`:

- **transition:** None (read-only).
- **output:** Returns the user's profile record.
- **exception:** NotFoundError if record absent.

`updateUserProfile()`:

- **transition:** Updates profile fields in database.
- **output:** Returns updated record.
- **exception:** ValidationError for invalid input or missing fields.

`updatePreferences()`:

- **transition:** Rewrites preference entries linked to user ID.
- **output:** Confirmation of successful update.
- **exception:** ValidationError on schema violation.

`deleteUser()`:

- **transition:** Removes user and related data (soft delete or cascade).
- **output:** Success confirmation.
- **exception:** UnauthorizedError if requester lacks privilege.

Local Functions

- `sanitizeProfileInput(data)` - Removes disallowed fields and formats strings.
- `mergePreferenceDefaults(prefs)` - Applies default values for missing preference options.
- `calculateReliabilityScore(userId)` - Computes user trust metric from ratings and trip history.

8 Route & Trip Module

8.1 Module

The Route & Trip Module manages trip creation, storage, and retrieval for both drivers and riders. It enables users to post, view, and manage trip listings that include origin, destination, departure time, and available seats. This module provides the core data used by the Matching Module to pair drivers and riders based on spatial and temporal compatibility.

Frontend Implemented within the Expo mobile application, the frontend presents interactive screens and forms that allow users to create, browse, and manage trips. Key features include:

- Trip creation form with origin, destination, date, and time pickers.
- Real-time map visualization of routes using a mapping API (e.g., Google Maps SDK).
- List and detail views for upcoming, active, and completed trips.
- Seat selection and trip cancellation interfaces for drivers.
- Integration with tRPC hooks such as `useCreateTrip` and `useGetTrips`.
- Validation feedback for missing fields or invalid time ranges.

Backend The backend implements business logic and trip operations through tRPC endpoints. It is responsible for managing trip lifecycle states and validating trip details before persistence. Primary responsibilities include:

- Exposing API endpoints for trip creation, retrieval, and deletion.
- Validating trip parameters (valid coordinates, available seats, future departure time).
- Associating trips with authenticated users via BetterAuth sessions.
- Handling role-based permissions (drivers can create trips, riders can request rides).
- Broadcasting trip updates to relevant modules (Matching, Scheduling).

Typical procedures include `createTrip`, `getTrips`, and `cancelTrip`.

Data Trip data is persisted in the PostgreSQL database via Drizzle ORM. The schema supports relational integrity between users, trips, and matches. Primary tables include:

- `trips` — stores trip metadata: origin, destination, time, seats, and driver ID.
- `trip_requests` — records ride requests by riders awaiting confirmation.
- `routes` — optional table caching route geometry and distance metrics.

Each record maintains timestamps for creation and modification, and all inserts are validated against foreign key constraints referencing the `users` table.

8.2 Uses

This module is used by both drivers and riders to manage travel logistics. It interacts directly with:

- **User Profile Module** - links trips to driver and rider profiles.
- **Matching Module** - provides trip data for route and schedule-based matching.
- **Scheduling Module** - integrates recurring trips and time management.
- **Notification Module** - triggers updates for trip confirmations or cancellations.

It serves as the foundation for dynamic route pairing and trip coordination within the Hitchly system.

8.3 Syntax

Exported Constants

- `MAX_SEATS = 5` - Maximum allowable seats per trip.
- `TIME_WINDOW_MIN = 15` - Minimum departure buffer in minutes.

Exported Access Programs

Name	In	Out	Exceptions
`createTrip()`	TripData	TripRecord	ValidationError
`getTrips()`	userId, filters	TripList	NotFoundError
`cancelTrip()`	tripId	Confirmation	UnauthorizedError
`updateTrip()`	tripId, UpdatedFields	UpdatedTripRecord	ValidationError
`getTripById()`	tripId	TripRecord	NotFoundError

8.4 Semantics

State Variables

- `activeTrips` - the current list of trips posted by the authenticated driver.
- `tripRequests` - ride requests awaiting driver approval.
- `tripCache` - locally cached trip data for offline viewing.

Environment Variables

- Mobile device GPS for route mapping.
- Network connection for fetching and updating trip data.
- PostgreSQL database via Drizzle ORM for persistent storage.

Assumptions

- The user is authenticated and verified before creating or editing a trip.
- The map service API key and geocoding are correctly configured.
- All database migrations have been applied prior to runtime.

Access Routine Semantics `createTrip()`:

- **transition:**
  - Creates new trip record from TripData input
  - activeTrips := activeTrips $\cup$ {newTrip}
  - Inserts new trip entry into the database associated with the user
  - Updates tripCache with the new trip record
- **output:** Returns the newly created trip record from activeTrips.
- **exception:** ValidationError if trip data invalid or incomplete.

`getTrips()`:

- **transition:** None (read-only operation).
- **output:** Returns filtered subset of activeTrips or tripCache based on userId and filters. If network available, reads from database and updates tripCache; otherwise returns from tripCache.

- **exception:** NotFoundError if no trips exist for criteria (activeTrips $= \emptyset$ for given filters).

`updateTrip()`:

- **transition:**
  - Locates trip in activeTrips by tripId
  - Updates the trip entry in activeTrips with new fields from UpdatedFields
  - Modifies corresponding entry in database
  - Updates tripCache with modified trip record
- **output:** Returns updated trip record from activeTrips.
- **exception:** ValidationError on invalid field or unauthorized edit (trip not found in activeTrips for current user).

`cancelTrip()`:

- **transition:**
  - Locates trip in activeTrips by tripId
  - activeTrips := activeTrips $\setminus$ {canceledTrip}
  - Marks trip as canceled in database (status := canceled)
  - Removes trip from tripCache
  - Notifies dependent modules (Matching, Notification) of cancellation
- **output:** Returns confirmation of cancellation.
- **exception:** UnauthorizedError if user lacks permission (trip not in activeTrips for current user).

`getTripById()`:

- **transition:** None (read-only operation).
- **output:** Returns specific trip details by ID. Searches activeTrips first, then tripCache if not found. If network available and not in cache, queries database and updates tripCache.
- **exception:** NotFoundError if ID does not exist (trip not found in activeTrips, tripCache, or database).

Local Functions

- `calculateRouteDistance(origin, destination)` - Computes estimated distance and duration.
- `validateTripInput(tripData)` - Ensures all required fields and constraints are satisfied.
- `filterTripsByTime(trips, window)` - Returns only those trips within a certain time range.

9 Matching Module

9.1 Module

The Matching Module is the core algorithmic engine of Hitchly. It is responsible for generating ranked matches between riders and drivers based on route proximity, time compatibility, and ride preferences. This module applies the **Strategy Pattern** to maintain flexibility, allowing future replacement or enhancement of the scoring algorithm (e.g., machine learning or context-aware scoring). Its outputs drive the swipe-based interface, where users browse and select potential ride matches.

Frontend Implemented in the Expo mobile app, the frontend presents matches as interactive swipeable cards in a Tinder-style interface. Key features include:

- Swipe gestures for liking or passing on potential matches.
- Real-time display of top recommended drivers or riders with route, time, and compatibility score.
- API integration through tRPC hooks (e.g., `useGetMatches`, `useSubmitSwipe`).
- Visual match indicators such as "Good Match" or percentage score badges.
- Smooth animations and immediate feedback for mutual matches.

Backend The backend implements the matchmaking logic within the Express/tRPC API, encapsulated in a dedicated service class (`MatchEngine`). The design follows the Strategy Pattern to keep scoring algorithms modular. Primary responsibilities include:

- Retrieving candidate users and associated trips from the database.
- Filtering incompatible pairs by hard constraints: verified McMaster email, opposite roles, overlapping time windows, and nearby routes.

- Applying a scoring strategy (default: WeightedMatchStrategy) that computes a match score (0–100) using normalized factors.
- Sorting and returning the highest-scoring results to the frontend.
- Storing swipe actions and updating mutual matches.

The backend exposes tRPC endpoints such as `findMatchesForUser`, `submitSwipe`, and `getMatchResults`.

Data The data layer maintains persistent records of matches, swipes, and scoring metrics within PostgreSQL via Drizzle ORM. Key tables include:

- `matches` — stores confirmed mutual matches (rider ¡-¿ driver) with timestamp and score.
- `swipes` — records all swipe actions for analytics and recommender tuning.
- `match_history` — archives past matches for feedback and trust scoring.

Foreign keys link match records to both users and trips, ensuring referential integrity.

9.2 Formal Specification

This subsection provides the comprehensive formal specification for the Matching Module using finite state machines and discrete mathematics. The Matching Module is specified as three focused state machines that model distinct aspects of the matching algorithm: match discovery, ride request management, and match viewing.

The state machines are designed to be:

- **Conceptually clear**: Each FSM models a single, well-defined workflow
- **Implementation-ready**: The runtime FSM enforces all edge cases and error handling
- **Documentation-friendly**: Simplified models for specification clarity

Note on Implementation vs. Specification The implementation-level FSM contains additional transitions for error handling, retries, and recovery paths that ensure runtime correctness. For clarity, the documented state machines present simplified conceptual models that focus on the primary workflow, while the runtime FSM enforces all edge cases. This separation follows best practices in formal specification, where abstraction aids understanding while implementation ensures robustness.

The Matching FSM (Figure 1) models the core match discovery workflow:

- Searching for potential candidates

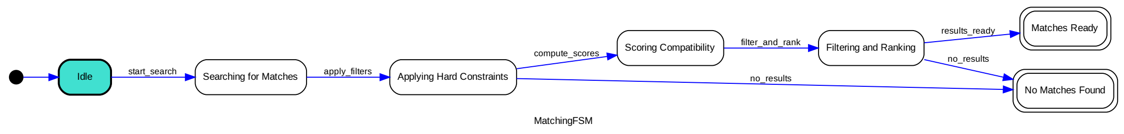

Figure 1: Matching FSM - Handles match discovery, filtering, scoring, and ranking

- Applying hard constraints (verified email, role compatibility, time windows, routes, seat availability)
- Scoring compatibility using weighted factors
- Filtering and ranking results
- Presenting matches or indicating no matches found

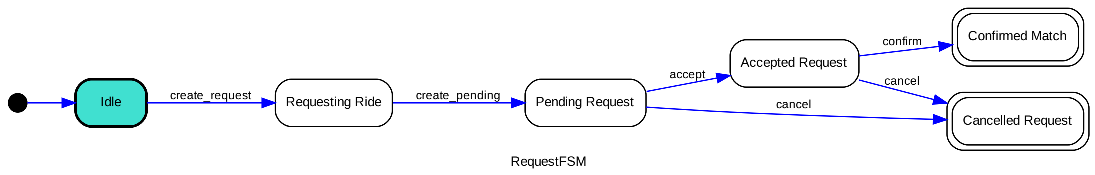

Figure 2: Request FSM - Handles ride request lifecycle

The Request FSM (Figure 2) models the ride request lifecycle:

- Creating a ride request
- Managing pending requests awaiting driver action
- Processing accepted requests
- Confirming matches or handling cancellations

The Viewing FSM (Figure 3) models the simple workflow for viewing confirmed matches:

- Querying accepted ride requests
- Displaying associated driver/rider details

9.2.1 State Definitions

The matching algorithm is modeled as three focused state machines, each handling a distinct aspect of the workflow. This separation improves clarity and maintainability while ensuring each FSM remains conceptually simple.

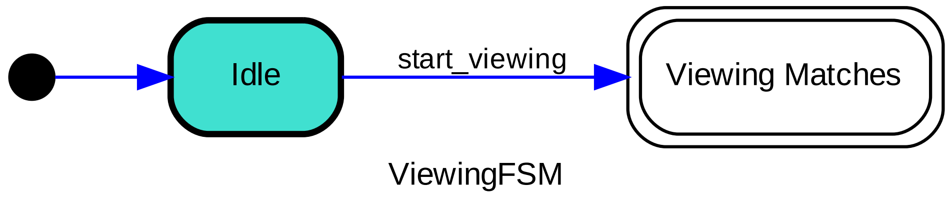

Figure 3: Viewing FSM - Handles viewing confirmed matches

Matching FSM States The Matching FSM handles the core match discovery and ranking process:

- **Idle**: Initial state, waiting for match discovery request.
- **Searching for Matches**: Retrieving potential candidates from database.
- **Applying Hard Constraints**: Filtering candidates using non-negotiable rules:
  - Verified McMaster email
  - Opposite role (rider $\leftrightarrow$ driver)
  - Overlapping time windows
  - Nearby / overlapping routes
  - Sufficient seat availability
- **Scoring Compatibility**: Computing match scores using weighted factors (Schedule, Location, Cost, Comfort, Preferences) and normalizing to 0–100.
- **Filtering and Ranking**: Removing matches below 30% threshold, sorting by score, selecting top 20 results.
- **Matches Ready**: Final state indicating ranked matches are available for presentation.
- **No Matches Found**: Final state indicating no valid matches exist after filtering.

Request FSM States The Request FSM handles the ride request lifecycle:

- **Idle**: Initial state, no active request.

- **Requesting Ride**: Validating ride existence and seat availability, creating request with status `pending`.
- **Pending Request**: Request awaiting driver action (accept/reject).
- **Accepted Request**: Request approved by driver, verifying authorization and updating capacity.
- **Confirmed Match**: Final state indicating mutual match confirmed.
- **Cancelled Request**: Final state indicating request was cancelled (by rider or driver).

Viewing FSM States The Viewing FSM handles viewing confirmed matches:

- **Idle**: Initial state, not viewing matches.
- **Viewing Matches**: Final state where system queries and displays accepted ride requests with associated driver/rider details.

State Machine Interactions While the three FSMs are conceptually separate, they interact in the following ways:

- The Matching FSM produces results that can trigger the Request FSM (when user selects a match).
- The Request FSM produces confirmed matches that can be viewed via the Viewing FSM.
- All FSMs can return to their respective Idle states, allowing the system to handle multiple operations sequentially.
- Error handling and recovery are managed at the implementation level, ensuring robustness while keeping the specification models clean.

9.2.2 Mathematical Definitions

The candidate filtering is formally specified using set theory. Let R be the set of all rides, and let r_r be the rider request.
The set of candidate rides is defined as:

$$\begin{aligned}\text{Candidates} = \{r \in R \mid & \text{status}(r) = \text{“scheduled”} \wedge \\ & \text{maxSeats}(r) \geq r_r.\text{maxOccupancy} \wedge \\ & \text{availableSeats}(r) \geq r_r.\text{maxOccupancy}\} \end{aligned} \tag{1}$$

The filtered candidates set excludes rides with compatibility dealbreakers:

$$\text{Filtered} = \{c \in \text{Candidates} \mid \text{compatibilityScore}(c, r_r) > 0\} \tag{2}$$

The scoring function `computeScore` is formally defined as a weighted sum of normalized factors. For the default preference preset, the weights are:

$$\begin{aligned}\text{rawScore}(r, d) = {} & 2.0 \times \text{scheduleScore}(r.\text{desiredArrivalTime}, d.\text{startTime}) + \\ & 2.0 \times \text{locationScore}(r.\text{origin}, r.\text{destination}, d.\text{origin}, d.\text{destination}) + \\ & 1.5 \times \text{costScore}(r.\text{estimatedCost}, \text{minCost}) + \\ & 0.5 \times \text{comfortScore}(d.\text{currentPassengers}, d.\text{maxSeats}, r.\text{maxOccupancy}) + \\ & 1.0 \times \text{compatibilityScore}(r.\text{prefs}, d.\text{prefs})\end{aligned} \tag{3}$$

where each factor function maps to the interval $[0, 1]$:

- scheduleScore : string $\times$ string $\to [0, 1]$ — Computes time compatibility. Returns 1.0 if driver departs 0-20 minutes after rider's desired time, decreasing linearly for larger differences.
- locationScore : $\text{Location}^4 \to [0, 1]$ — Computes route compatibility based on detour time. Returns 1.0 if detour is within tolerance, decreasing exponentially for excess detour time.
- costScore : $\mathbb{R} \times \mathbb{R} \to [0, 1]$ — Computes cost attractiveness relative to minimum cost among all candidates. Lower cost relative to minimum yields higher score.
- comfortScore : $\mathbb{N}^3 \to [0, 1]$ — Computes seat availability comfort. Returns 0.0 if insufficient seats, otherwise $1.0 - \text{currentPassengers}/\text{maxSeats}$.
- compatibilityScore : $\text{PreferenceData}^2 \to [0, 1]$ — Computes preference compatibility. Returns 0.0 for dealbreakers (smoking, pets mismatches), otherwise computes soft matching score based on music and chatty preferences.

The raw score is normalized to a percentage (0-100) using the maximum theoretical score:

$$\text{normalizeScore}(\text{rawScore}) = \min\left(100, \left\lfloor \frac{\text{rawScore}}{7.0} \times 100 + 0.5 \right\rfloor\right) \tag{4}$$

where the maximum theoretical score is 7.0 (sum of all weights: $2.0 + 2.0 + 1.5 + 0.5 + 1.0 = 7.0$). The addition of 0.5 before flooring implements rounding to the nearest integer, equivalent to $\text{round}(\text{rawScore}/7.0 \times 100)$.

The final ranked match list is constructed by filtering matches above the threshold, sorting by normalized score, and taking the top candidates:

$$\begin{aligned}\text{validMatches} &= \{m \in \text{scoredMatches} \mid \\ &\qquad \text{normalizeScore}(m.\text{rawScore}) \geq \text{MATCH\_THRESHOLD} \times 100\} \\ \text{matchResults} &= \text{take}(\text{sort}(\text{validMatches}, \text{descending}), \text{MAX\_CANDIDATES})\end{aligned} \tag{5}$$

where MATCH_THRESHOLD = 0.3 (30% minimum), MAX_CANDIDATES = 20, $\text{sort}(S, \text{descending})$ returns sequence S sorted by normalized score in descending order, and $\text{take}(S, n)$ returns the first n elements of sequence S.

9.3 Uses

This module connects directly with:

- **Route & Trip Module** - provides trip data (origin, destination, departure time)
- **User Profile Module** - provides preference data and reliability rating.
- **Swipe Interaction Frontend** - renders and collects swipe input.
- **Notification Module** - notifies users upon mutual matches.

It is the computational bridge between trip listings and user engagement, supporting Hitchly's key matchmaking feature.

9.4 Syntax

Exported Constants

- `MAX_CANDIDATES = 20` - limits results per query.
- `MATCH_THRESHOLD = 0.3` - minimum normalized score (30%) to display.

Exported Access Programs

Name	**In**	**Out**	**Exceptions**
`findMatchesForUser()`	userId, request	RankedMatchList	NotFoundError
`requestRide()`	rideId, pickupLat, pickupLng	Confirmation	NotFoundError, BadRequestError
`acceptRequest()`	requestId	Confirmation	NotFoundError, BadRequestError, ForbiddenError
`cancelRequest()`	requestId	Confirmation	NotFoundError, ForbiddenError
`getMatchResults()`	userId	MatchSummary	None
`computeScore()`	rider, driver	Float (0-100)	AlgorithmError

9.5 Semantics

State Variables The module state is defined by the current state machine state and associated data:

- `moduleState` $\in \{\text{UNINITIALIZED}, \text{INITIALIZING}, \text{READY}, \text{PROCESSING}, \text{IDLE}, \text{ERROR}, \text{SHUT}$ — Current lifecycle state
- `operationState` $\in$ MatchStates $\cup$ RequestStates $\cup$ AcceptStates $\cup$CancelStates $\cup$ ResultsStates — Current operation state, where each set contains the states for the respective operation
- `candidatePool` - list of drivers or riders retrieved for a user (valid in states: QUERYING_RIDES, VALIDATING_CANDIDATES, COMPUTING_ROUTES, SCORING_CANDIDATES)
- `activeStrategy` - reference to the current scoring algorithm (valid when moduleState $\in \{\text{READY}, \text{PROCESSING}\}$)
- `matchResults` - ordered list of potential matches with scores (valid in states: RANKING_MATCHES, MATCH_COMPLETE)
- `operationQueue` - queue of pending operations (valid when moduleState = READY or PROCESSING)
- `currentOperation` - currently executing operation (valid when moduleState = PROCESSING)

Environment Variables

- Active network connection for API queries.
- Database connection for retrieving and updating match data.
- Access to external map service for route distance calculations.

Assumptions

- Each user has an active, verified profile with trip data available.
- Distance and time calculations are performed via a reliable geocoding API.
- Only verified McMaster users participate in matching.

Access Routine Semantics All access routines are specified as state machine transitions. Each routine transitions the module from READY state through operation-specific states to either a completion state or error state.

`findMatchesForUser(userId)`:

- **precondition:** moduleState = READY $\wedge$ userId $\in \mathbb{N} \wedge$ userId > 0
- **state transition sequence:**

$$\begin{aligned}
\text{READY} &\xrightarrow{\text{findMatchesForUser(userId)}} \text{PROCESSING}\\
\text{PROCESSING} &\rightarrow \text{MATCH\_INITIAL}\\
\text{MATCH\_INITIAL} &\rightarrow \text{FETCHING\_PREFERENCES}\\
&\quad \text{action: riderPrefs} := \text{queryPreferences(userId)}\\
\text{FETCHING\_PREFERENCES} &\rightarrow \text{QUERYING\_RIDES}\\
&\quad \text{action: candidatePool} := \text{retrieveCandidates(userId)}\\
\text{QUERYING\_RIDES} &\rightarrow \text{VALIDATING\_CANDIDATES}\\
&\quad \text{action: validCandidates} := \{r \in \text{candidatePool} \mid \text{availableSeats}(r) \geq\\
\text{VALIDATING\_CANDIDATES} &\rightarrow \text{COMPUTING\_ROUTES}\\
&\quad \text{action: } \forall c \in \text{validCandidates} : \text{routeDetails}[c] := \text{getDetourAndRid}\\
\text{COMPUTING\_ROUTES} &\rightarrow \text{SCORING\_CANDIDATES}\\
&\quad \text{action: scoredMatches} := \text{map(computeScore, validCandidates)}\\
\text{SCORING\_CANDIDATES} &\rightarrow \text{FILTERING\_THRESHOLD}\\
&\quad \text{action: validMatches} := \{m \in \text{scoredMatches} \mid \text{normalizeScore}(m.\text{ra}\\
\text{FILTERING\_THRESHOLD} &\rightarrow \text{RANKING\_MATCHES}\\
&\quad \text{action: matchResults} := \text{sort(validMatches, descending)}\\
\text{RANKING\_MATCHES} &\rightarrow \text{MATCH\_COMPLETE}\\
&\quad \text{action: matchResults} := \text{take(matchResults, MAX\_CANDIDATES)}\\
\text{MATCH\_COMPLETE} &\rightarrow \text{IDLE}\\
\text{IDLE} &\rightarrow \text{READY}
\end{aligned}$$

- **postcondition:** moduleState = READY $\wedge$ matchResults $\neq$ null $\wedge$ $|$matchResults$|$ $\leq$ MAX_CANDIDATES
  where MAX_CANDIDATES = 20
- **output:** Returns matchResults (ranked list of matches with normalized scores).
- **exception:** If transition to MATCH_ERROR occurs: NotFoundError (when candidatePool = $\emptyset$ or validCandidates = $\emptyset$ or validMatches = $\emptyset$). Error recovery transitions to READY state.

`requestRide(rideId, pickupLat, pickupLng)`:

- **precondition:** moduleState $=$ READY $\wedge$ rideId $\neq$ null $\wedge \text{pickupLat}, \text{pickupLng} \in \mathbb{R}$
- **state transition sequence:**

$$\begin{aligned}
\text{READY} &\xrightarrow{\text{requestRide}} \text{PROCESSING} \\
\text{PROCESSING} &\rightarrow \text{REQUEST\_INITIAL} \\
\text{REQUEST\_INITIAL} &\rightarrow \text{VALIDATING\_RIDE} \\
&\quad \text{action: ride := queryRide(rideId)} \\
\text{VALIDATING\_RIDE} &\rightarrow \text{CHECKING\_SEATS} \\
&\quad \text{precondition: ride} \neq \text{null} \\
\text{CHECKING\_SEATS} &\rightarrow \text{CREATING\_REQUEST} \\
&\quad \text{precondition: ride.bookedSeats} < \text{ride.maxSeats} \\
&\quad \text{action: insert(rideRequests}, \{\text{rideId}, \text{riderId}, \text{pickupLat}, \text{pickupLng}, \text{status} \\
\text{CREATING\_REQUEST} &\rightarrow \text{REQUEST\_COMPLETE} \\
\text{REQUEST\_COMPLETE} &\rightarrow \text{IDLE} \rightarrow \text{READY}
\end{aligned}$$

- **postcondition:** moduleState $=$ READY $\wedge$ $\exists r \in$ rideRequests $:$ r.id $=$ requestId $\wedge$ r.status $=$ ”pending”
- **output:** Confirmation with requestId.
- **exception:** If transition to REQUEST_ERROR occurs: NotFoundError (ride not found) or BadRequestError (ride full).

`acceptRequest(requestId)`:

- **precondition:** moduleState $=$ READY $\wedge$ requestId $\neq$ null

- **state transition sequence:**

$$\begin{aligned}
\text{READY} &\xrightarrow{\text{acceptRequest}} \text{PROCESSING}\\
\text{PROCESSING} &\to \text{ACCEPT\_INITIAL}\\
\text{ACCEPT\_INITIAL} &\to \text{VALIDATING\_REQUEST}\\
&\quad \text{action: request := queryRequest(requestId)}\\
\text{VALIDATING\_REQUEST} &\to \text{VERIFYING\_DRIVER}\\
&\quad \text{precondition: request.status = "pending"}\\
\text{VERIFYING\_DRIVER} &\to \text{CHECKING\_CAPACITY}\\
&\quad \text{precondition: ride.driverId = ctx.userId}\\
\text{CHECKING\_CAPACITY} &\to \text{UPDATING\_STATUS}\\
&\quad \text{precondition: ride.bookedSeats < ride.maxSeats}\\
&\quad \text{action: request.status := "accepted"}\\
\text{UPDATING\_STATUS} &\to \text{INCREMENTING\_SEATS}\\
&\quad \text{action: ride.bookedSeats := ride.bookedSeats + 1}\\
\text{INCREMENTING\_SEATS} &\to \text{ACCEPT\_COMPLETE}\\
\text{ACCEPT\_COMPLETE} &\to \text{IDLE} \to \text{READY}
\end{aligned}$$

- **postcondition:** $\text{moduleState} = \text{READY} \land \text{request.status} = \text{"accepted"} \land \text{ride.bookedSeats} = \text{ride.bookedSeats}_{\text{old}} + 1$
- **output:** Success confirmation.
- **exception:** If transition to ACCEPT_ERROR occurs: NotFoundError, BadRequestError, or ForbiddenError.

`cancelRequest(requestId)`:

- **precondition:** $\text{moduleState} = \text{READY} \land \text{requestId} \neq \text{null}$

- **state transition sequence:**

$$\begin{aligned}
\text{READY} &\xrightarrow{\text{cancelRequest}} \text{PROCESSING}\\
\text{PROCESSING} &\rightarrow \text{CANCEL\_INITIAL}\\
\text{CANCEL\_INITIAL} &\rightarrow \text{VALIDATING\_OWNERSHIP}\\
&\quad \text{precondition: request.riderId} = \text{ctx.userId}\\
\text{VALIDATING\_OWNERSHIP} &\rightarrow \text{CHECKING\_STATUS}\\
\text{CHECKING\_STATUS} &\rightarrow \text{UPDATING\_CANCEL}\\
&\quad \text{action: request.status} := \text{"cancelled"}\\
\text{UPDATING\_CANCEL} &\xrightarrow{\text{was accepted}} \text{DECREMENTING\_SEATS}\\
&\quad \text{action: ride.bookedSeats} := \max(0, \text{ride.bookedSeats} - 1)\\
\text{UPDATING\_CANCEL} &\xrightarrow{\text{was pending}} \text{CANCEL\_COMPLETE}\\
\text{DECREMENTING\_SEATS} &\rightarrow \text{CANCEL\_COMPLETE}\\
\text{CANCEL\_COMPLETE} &\rightarrow \text{IDLE} \rightarrow \text{READY}
\end{aligned}$$

- **postcondition:** moduleState = READY $\wedge$ request.status = "cancelled"
- **output:** Success confirmation.
- **exception:** If transition to CANCEL_ERROR occurs: NotFoundError or Forbidden-Error.

`getMatchResults(userId)`:

- **precondition:** moduleState = READY $\wedge$ userId $\in \mathbb{N}$
- **state transition sequence:**

$$\begin{aligned}
\text{READY} &\xrightarrow{\text{getMatchResults}} \text{PROCESSING}\\
\text{PROCESSING} &\rightarrow \text{RESULTS\_INITIAL}\\
\text{RESULTS\_INITIAL} &\rightarrow \text{QUERYING\_REQUESTS}\\
&\quad \text{action: acceptedRequests} := \text{query(rideRequests},\\
&\quad\quad \{\text{riderId} = \text{userId}, \text{status} = \text{"accepted"}\})\\
\text{QUERYING\_REQUESTS} &\xrightarrow{\text{requests found}} \text{FETCHING\_DRIVERS}\\
&\quad \text{action: } \forall r \in \text{acceptedRequests} :\\
&\quad\quad r.\text{driver} := \text{queryUser}(r.\text{driverId})\\
\text{QUERYING\_REQUESTS} &\xrightarrow{\text{no requests}} \text{RESULTS\_COMPLETE}\\
\text{FETCHING\_DRIVERS} &\rightarrow \text{RESULTS\_COMPLETE}\\
\text{RESULTS\_COMPLETE} &\rightarrow \text{IDLE} \rightarrow \text{READY}
\end{aligned}$$

- **postcondition:** moduleState = READY $\wedge$ results $\neq$ null
- **output:** Returns list of confirmed mutual matches (may be empty).
- **exception:** None (returns empty list if no matches found).

`computeScore(rider, driver)`:

- **precondition:** moduleState = READY $\vee$ (moduleState = PROCESSING $\wedge$operationState = SCORING_CANDIDATES)
- **transition:** Pure function, no state modification. Computes score using activeStrategy.
- **output:** Floating-point score between 0 and 100 computed as normalizeScore(rawScore(r, d)) where rawScore is defined by Equation (3) and normalizeScore is defined by Equation (4).
- **exception:** AlgorithmError if scoring function fails (e.g., activeStrategy is undefined).

Local Functions

- `applyFilters(user, candidate)` - evaluates eligibility constraints (status="scheduled", seat availability, compatibility dealbreakers).
- `calculateWeightedScore(rider, driver)` - computes weighted sum using factors (default preset):
  - Schedule score (weight 2.0) - time compatibility
  - Location score (weight 2.0) - route/detour compatibility
  - Cost score (weight 1.5) - cost attractiveness
  - Comfort score (weight 0.5) - seat availability
  - Compatibility score (weight 1.0) - preference matching
- `normalizeScore(raw)` - scales raw score (0-7.0) into [0, 100] using formula: min(100, $\lfloor$(rawScore/7.0) $\times$ 100 + 0.5$\rfloor$), which rounds to the nearest integer.

10 Scheduling Module

10.1 Module

The Scheduling Module manages recurring and one-time ride schedules for drivers and riders. It allows users to automate trip creation, define weekly patterns, and ensures that recurring schedules generate valid trip entries. This module integrates directly with the Trip Module and the Notification Module.

Frontend Implemented in the Expo app, the frontend provides UI controls for defining recurring or one-time schedules.

Key capabilities:

- Time picker for selecting departure time
- Day-of-week toggles for recurring schedules
- Interface to view and cancel existing schedules
- Integration with tRPC hooks such as `useCreateSchedule`, `useGetUserSchedules`
- Form validation for time ranges and required fields

Backend The backend implements scheduling logic via tRPC.

Responsibilities include:

- Parsing user-defined recurrence patterns
- Generating future trips based on schedules
- Ensuring schedules do not conflict with existing trips
- Validating schedule timing (future date, valid days)
- Exposing procedures such as `createSchedule`, `cancelSchedule`

Data The module persists schedule definitions and generated trip associations.

Tables:

- `schedules` — recurring schedule definitions (days, time, userId)
- `schedule_instances` — maps schedules to generated trips
- `trips` — created by this module but stored by Trip Module

10.2 Uses

- Trip Module — creates trip entries from schedules
- User Profile Module — determines valid schedule types
- Notification Module — sends reminders for upcoming rides

10.3 Syntax

Exported Constants

- `MAX_RECURRING_YEARS = 1`
- `VALID_DAYS = [Mon..Sun]`

Exported Access Programs

Name	In	Out	Exceptions
`createSchedule()`	scheduleData	ScheduleRecord	ValidationError
`getUserSchedules()`	userId	ScheduleList	NotFoundError
`cancelSchedule()`	scheduleId	Confirmation	UnauthorizedError
`generateFutureTrips()`	scheduleId	TripList	AlgorithmError

10.4 Semantics

State Variables

- `userSchedules`
- `activeRules`

Environment Variables

- System scheduler
- Device timezone
- Database connection

Assumptions

- Users must be verified before scheduling
- Database schema is correctly migrated

Access Routine Semantics `createSchedule()`:

- **transition:**
  - Creates new schedule record from scheduleData input
  - userSchedules := userSchedules $\cup$ {newSchedule}
  - activeRules := activeRules $\cup$ {parseRecurrenceRule(newSchedule)}
  - Inserts schedule into database
- **output:** Returns schedule record from userSchedules.
- **exception:** ValidationError if schedule data invalid (invalid days, past time, etc.).

`getUserSchedules()`:

- **transition:** None (read-only operation).
- **output:** Returns filtered subset of userSchedules for the given userId.
- **exception:** NotFoundError if no schedules exist for userId (userSchedules $= \emptyset$ for given userId).

`cancelSchedule()`:

- **transition:**
  - Locates schedule in userSchedules by scheduleId
  - userSchedules := userSchedules $\setminus$ {canceledSchedule}
  - Removes corresponding rule from activeRules
  - Updates database to mark schedule as canceled
- **output:** Returns confirmation of cancellation.
- **exception:** UnauthorizedError if user lacks permission (schedule not in userSchedules for current user).

`generateFutureTrips()`:

- **transition:**
  - Locates schedule in userSchedules by scheduleId
  - Retrieves corresponding recurrence rule from activeRules
  - Uses nextOccurrence() to compute future trip dates based on active rule
  - Generates trip entries for each future occurrence (up to MAX_RECURRING_YEARS)
  - Creates trip entries via Trip Module's `createTrip` procedure
- **output:** Returns list of generated TripRecord entries.
- **exception:** AlgorithmError if recurrence rule parsing fails or schedule not found in activeRules.

Local Functions

- `parseRecurrenceRule`
- `nextOccurrence`

11 Notification Module

11.1 Module

The Notification Module handles system alerts including match notifications, trip updates, and reminders.

Frontend Implemented via Expo Notifications.
UI responsibilities:

- Requesting push permission
- Displaying notification center
- Handling tap events

Frontend integration:

- `useRegisterPushToken`
- `useGetNotifications`

Backend Backend procedures manage dispatch and retrieval:

- Storing device push tokens
- Sending push messages via Expo Push API
- Queueing async notification events
- Procedures: `sendNotification`, `markAsRead`

Data

- `notifications` — stored messages
- `push_tokens` — device tokens

11.2 Uses

- Matching Module — notifies mutual matches
- Scheduling Module — sends reminders
- Trip Module — alerts cancellations

11.3 Syntax

Exported Access Programs

Name	In	Out	Exceptions
`sendNotification()`	userId, message	Confirmation	DeliveryError
`getNotifications()`	userId	NotificationList	NotFoundError
`markAsRead()`	notificationId	Confirmation	UnauthorizedError

11.4 Semantics

State Variables

- `queuedNotifications`
- `readStatus`

Environment Variables

- Expo Push Service

Local Functions

- `formatNotificationPayload`
- `enqueueNotification`

12 Rating & Feedback Module

12.1 Module

This module collects and stores ride ratings and feedback, updating reliability scores used by the Matching Module.

Frontend UI elements:

- Post-ride rating screen
- Optional text feedback box
- History of past ratings

Hooks:

- `useSubmitRating`
- `useGetUserRatings`

Backend Backend responsibilities:

- Validating rating payloads
- Linking ratings to matches
- Recomputing reliability scores
- Procedures: `submitRating`, `calculateReliability`

Data Tables:

- `ratings`
- `reliability_metrics`

12.2 Uses

- Matching Module — reliability score as weighted factor
- User Profile Module — shows aggregated rating

12.3 Syntax

Exported Access Programs

Name	In	Out	Exceptions
`submitRating()`	ratingData	Confirmation	ValidationError
`getUserRatings()`	userId	RatingList	NotFoundError
`calculateReliability()`	userId	Score	AlgorithmError

12.4 Semantics

State Variables

- `recentRatings`
- `reliabilityScore`

Local Functions

- `computeWeightedAverage`
- `sanitizeComment`

13 Safety & Reporting Module

13.1 Module

This module manages user-submitted safety reports and flags, assisting platform moderation.

Frontend Includes:

- Report User form
- Safety resources page

Hooks:

- `useSubmitReport`

Backend Backend responsibilities:

- Validating and storing incident reports
- Assigning severity scores
- Procedures: `submitReport`, `resolveReport`

Data Tables include:

- `reports`
- `safety_flags`

13.2 Uses

- Admin & Moderation Module
- Notification Module

13.3 Syntax

Exported Access Programs

Name	In	Out	Exceptions
`submitReport()`	reportData	Confirmation	ValidationError
`getReports()`	userId/adminId	ReportList	UnauthorizedError
`resolveReport()`	reportId, action	Confirmation	PermissionError

13.4 Semantics

State Variables

- `pendingReports`
- `userSafetyStatus`

Local Functions

- `categorizeSeverity`
- `flagUser`
- `closeReport`

14 Payment Module

14.1 Module

This module is responsible for handling in-app user payments. The implementation of this module will be done through Stripe. This acts as a payment gateway to ensure a secure transaction and successful payment.

Frontend The frontend of this application will be implemented using the Expo Client Application. A payment screen would display a form to fill out the required information for processing the payment. This includes the user's full name, mailing address, payment method, and payment information (i.e., Credit Card Number, etc.). For other methods of payment (i.e., Google Pay, PayPal, etc.), the frontend will be implemented to manage redirection. Upon a successful transaction, the user can select an option to go back to the home screen.

Backend The backend will receive the inputs from frontend and validate the user and ride data. It is then responsible for processing the payment via calling payment gateway API to authorize the payment method and details. It will open a secure session to process the payment and get the state (i.e., payment successful, processing, error) to display as the output. Lastly, this component will ensure that the transaction is recorded in the database and that receipts have been sent to the user's emails.

Data The database component is responsible for storing the user's payment history, ride details, and transaction information.

14.2 Uses

This module ensures secure access to Hitchly by verifying each user's McMaster University affiliation. It interacts directly with:

- **User Profile Module:** This is used to retrieve user profile details for verification.
- **Route & Trip Module:** This is used to retrieve trip details like cost, ID, etc.
- **Database Module:** This is used to store trip transaction details into the database.

14.3 Syntax

Exported Constants

- `max_attempt = 4` Maximum attempts to make a transaction before they have to wait for 30 minutes.
- `max_session = 25` Maximum allowed time (in minutes) for a session to process the payment before it throws a processing error.

Exported Access Programs

Name	In	Out	Exceptions
`init_payment()`	user_id, ride_id	Boolean (verified or unverified)	NotFoundError
`process_payment()`	payment_id, payment_method	payment_status	ExpiredSessionError
`get_paymentRecord()`	payment_id, user_id, ride_id	paymentRecord	NotFoundError

14.4 Semantics

State Variables

- `current_amount: float` — holds the current amount for the transaction.
- `session_token: string` — holds the current amount for the transaction.
- `sessionState: object` — holds session metadata for frontend persistence.

Environment Variables

- Secure HTTPS connection to API.
- Secure API connection to payment gateways.

Assumptions

- The users won't intentionally input expired or invalid credentials.
- The API connection is secure and stable throughout the payment process.
- The payment gateways will handle user's confidential information securely.

Access Routine Semantics `init_payment()`:

- **transition:** None
- **output:** Bool (user verified for the specific ride or not)
- **exception:** NotFoundError, user or ride not found.

`Process_payment()`:

- **transition:** Initializes connection with the API.
- **output:** Returns whether the payment is succesful, processing or failed.
- **exception:** ExpiredSessionError if the session fails to process the payment due to session limits.

`get_paymentRecord()`:

- **transition:** Creates a payment record.
- **output:** List of payment records.
- **exception:** NotFoundError if one of the required parameters is missing.

Local Functions

- `generateVerificationToken(): string` — Creates a cryptographically secure token.

15 Database Module

15.1 Module

This module is responsible for handling all database-related functionalities and ensuring smooth communication between the database servers for retrieving and updating data. This includes user information, ride details, transaction logs, etc.

Frontend N/A

Backend The backend layer of this module handles CRUD operations for core system entities and manages communication between the application and the database for smooth retrieval and update of application data.

Data This layer will store a schema of tables used across multiple modules. This includes:

- **User Table:** This stores all critical profile information of the user.
- **Ride Table:** This stores all critical ride details.
- **Payment Table:** This stores the user transactions for each ride.
- **Safety Report Table:** This stores all user safety reports.
- **Admin Table:** This stores the user's role details.

15.2 Uses

- **User Profile Module:** This is used to retrieve and store user profile details.
- **Route & Trip Module:** This is used to retrieve and store trip details like cost, ID, etc.
- **Payment Module:** This is used to retrieve and store user's transaction details.

- **Safety and Reporting Module:** This is used to retrieve and store user reports.
- **Admin Module:** This is used to retrieve and store admin details.

15.3 Syntax

Exported Constants

- `max_connect = 30` This is the maximum number of database connections at a time.
- `max_attempt = 4` Maximum attempts made to connect to db before the connection fails.
- `max_DB_session = 30` 30 minutes limit before the session times out.

Exported Access Programs

Name	In	Out	Exceptions
`addRecord()`	tablename, data	Boolean	NotFoundError, InvalidDataFormatException, DBTimeoutException
`updateRecord()`	tablename, data, record_id	Boolean	NotFoundError, InvalidDataFormatException, DBTimeoutException
`deleteRecord()`	tablename, record_id	Boolean	NotFoundError, InvalidDataFormatException, DBTimeoutException
`getRecord()`	tablename, data_condition	List of data records	NotFoundError, InvalidDataFormatException, DBTimeoutException
`getAllRecords()`	tablename	List of data records	NotFoundError, InvalidDataFormatException, DBTimeoutException

15.4 Semantics

State Variables

- `DBConnections: List<connections>` — List of active database connections.

Environment Variables

- Secure HTTPS connection to API.
- Secure API connection and communication the database server.
- Secure connection to local/cloud backup services for data backup.

Assumptions

- The Postgress database is avaiable and securely stores data.
- The database has sufficient storage capacity.
- The database automatically runs its data backup services to ensure that no data is lost.

Access Routine Semantics `addRecord()`:

- **transition:** A new entry is added to the specified table.
- **output:** Bool (True or False, depending on the state of the transition).
- **exception:** NotFoundError (table not found), InvalidDataFormatException, DBTimeoutException.

`updateRecord()`:

- **transition:** An existing entry is updated in the specified table.
- **output:** Bool (True or False, depending on the state of the transition).
- **exception:** NotFoundError (table or entry not found), InvalidDataFormatException, DBTimeoutException.

`deleteRecord()`:

- **transition:** An existing entry is deleted in the specified table.
- **output:** Bool (True or False, depending on the state of the transition).
- **exception:** NotFoundError (table or entry not found), InvalidDataFormatException, DBTimeoutException.

`getRecord()`:

- **transition:** None.
- **output:** Returns a list of indicated records.
- **exception:** NotFoundError (table not found), DBTimeoutException.

`getAllRecords()`:

- **transition:** None.
- **output:** Returns a list of all records.
- **exception:** NotFoundError (table not found), DBTimeoutException.

Local Functions

- `db_connect(): bool` — This is to initialize a connection with the database service.
- `checkFormat(): bool` — This function checks if the inserted data aligns with the specified schema.

16 Pricing Module

16.1 Module

This module handles the initial cost estimation functionality of the application.

Frontend This layer provides the UI for displaying the factors influencing the price per ride, along with a price estimation for it. It dynamically updates the price as new rides join the ride for a specific ride.

Backend This layer provides the algorithm to use those factors and accurately calculate the cost estimation for each rider for a given trip. It interacts with the Google Maps API to fetch location and distance values. Additionally, it sends this data to the payment module for processing the final payment for each rider.

Data The database component is responsible for storing the following table:

- **Pricing Table:** Contains data related to pricing as per the parameters.
- **Ride Table:** Contains data related to ride details.

16.2 Uses

- **User Profile Module:** This is used to retrieve the user type.
- **Database Module:** This is used to retrieve and store pricing details.
- **Route & Trip Module:** This is used to retrieve and store trip details like cost, ID, etc.
- **Payment Module:** The final cost estimation from this module is sent to the payment module.

16.3 Syntax

Exported Constants

- `max_session_time = 30` 30 minutes limit before the API session times out.

Exported Access Programs

Name	In	Out	Exceptions
`get_ride_summary()`	rideID	ride_summary	NotFoundError
`estimate_cost()`	start_location, end_location, rider_count	fare_estimate	InvalidLocationException, APITimeOutException
`update_fare()`	new_rider_count	updated_fare	APITimeOutException

16.4 Semantics

State Variables N/A

Environment Variables

- Secure HTTPS connection to API.
- Secure Maps API connection for accurate estimation of travel time and distance.
- Secure API connections to get daily fuel rates.

Assumptions

- The external API services function properly with minimal downtime.
- The fuel estimates online are accurate.
- The internet connection stays stable during the cost estimation process.

Access Routine Semantics `get_ride_summary()`:

- **transition:** None.
- **output:** Returns a ride summary object.
- **exception:** NotFoundError (ride not found).

`estimate_cost()`:

- **transition:** Retrieves information from API and updates database tables.
- **output:** Fare_estimate, returns an estimated price value.
- **exception:** InvalidLocationException, APITimeOutException.

`update_fare()`:

- **transition:** Retrieves information from API and updates database tables.
- **output:** Updated_fare, returns an estimated price value.
- **exception:** APITimeOutException.

Local Functions

- `validate_parameters(start_location, end_location, rider_count): bool` — Check if the values are not negative and are valid inputs.

17 Admin/Moderation Module

17.1 Module

This module handles the distribution of user roles and management of admin controls. This also controls and monitors activity and ensures strict policy enforcement.

Frontend This layer is responsible for providing an admin dashboard with flagged content, user complaints, user roles, and application analytics.

Backend This layer is responsible for handling communication with RESTAPI endpoints using AdminRouter to manage user accounts and rides. Additionally, it runs the logic for handling role-based authentication.

Data

- **Admin Table:** This would include profile information about admin.
- **Ride Table:** This would include information about users with warnings/bans.
- **User Analytics Table:** This includes a dataset of user and application statistics.

17.2 Uses

- **Database Module:** This is used to retrieve and store pricing details.
- **Authentication & Verification Module:** This is used to manage admin login and permissions.

17.3 Syntax

Exported Constants

- `warning_threshold= 3` Maximum warnings before putting an account ban.

Exported Access Programs

Name	In	Out	Exceptions
`warn_user()`	user_ID, reason	Bool (check if user is successfully warned)	NotFoundError, InsufficientAdminPrivileges
`get_Reports()`	None	Returns a dataset of user reports	NotFoundError, InsufficientAdminPrivileges
`get_analytics()`	None	Returns a dataset of user and application statistics	NotFoundError, InsufficientAdminPrivileges

17.4 Semantics

State Variables N/A

Environment Variables

- Secure HTTPS connection to API.
- Secure API connection for stable admin controls and management.

Assumptions

- All admin permissions function properly.
- All database services are full functional and stable.
- All API connections are functional and stable.

Access Routine Semantics `warn_user()`:

- **transition:** Adds a new record with a warning in the reports table.
- **output:** Bool (check if user is successfully warned).
- **exception:** NotFoundError (user_ID not found), InsufficientAdminPrivileges.

`get_Reports()`:

- **transition:** None.
- **output:** Returns a dataset of user reports.
- **exception:** NotFoundError (empty dataset), InsufficientAdminPrivileges.

`get_analytics()`:

- **transition:** None.
- **output:** Returns a dataset of user and application statistics.
- **exception:** NotFoundError (empty dataset), InsufficientAdminPrivileges.

Local Functions

- `check_privilege(admin_ID, endpoint): bool` — This function validates admin privileges and returns a bool to indicate whether they have sufficient privileges or not.

18 Appendix

Appendix — Reflection

The information in this section will be used to evaluate the team members on the graduate attribute of Problem Analysis and Design.

The purpose of reflection questions is to give you a chance to assess your own learning and that of your group as a whole, and to find ways to improve in the future. Reflection is an important part of the learning process. Reflection is also an essential component of a successful software development process.

Reflections are most interesting and useful when they're honest, even if the stories they tell are imperfect. You will be marked based on your depth of thought and analysis, and not based on the content of the reflections themselves. Thus, for full marks we encourage you to answer openly and honestly and to avoid simply writing "what you think the evaluator wants to hear."

Please answer the following questions. Some questions can be answered on the team level, but where appropriate, each team member should write their own response:

1. What went well while writing this deliverable?

   - **Aidan Froggatt:** For me, the deliverable went well because I already had strong familiarity with our codebase and architectural decisions. Since I set up the initial project foundation—Turborepo structure, tRPC API, database schema, and BetterAuth configuration—I had a clear understanding of how the modules needed to be defined and integrated. This made it easier to produce accurate, consistent MIS entries. I also found that working through each module clarified how our system would scale and how responsibilities should be divided, which improved the overall coherence of the document.
   - **Swesan:** The previous documents (such as the SRS and the earlier drafts) had already established much of the conceptual groundwork, which made defining the modules relatively straightforward. Since we entered this deliverable with a strong understanding of the system's structure, identifying responsibilities and writing the Module Guide flowed naturally. I personally worked on Section 5 (Module Hierarchy) and Section 7, and having those earlier documents as references made the process efficient and focused.
   - [**Burhanuddin Kharodawala**]**:** We had a good discussion as a group which helped me develop a better understanding of the overall content of this deliverable. Moreover, the template had clear understandings for most of the sections I worked on.

2. What pain points did you experience during this deliverable, and how did you resolve them?

   - **Aidan Froggatt:** One of the main challenges for me was determining the correct level of abstraction for each module. Because I am so close to the actual implementation, it was easy to accidentally include too many low-level details. I had to step back and focus on "what" the module exposes rather than "how" it works internally. Another pain point was aligning terminology with the Module Guide—for example, ensuring naming consistency and clearly defining frontend, backend, and data responsibilities. I resolved these issues by revisiting the SRS and MG structure and adjusting my MIS sections to fit the expected style and rigor of the course.

- **Swesan:** One of the main challenges was determining which modules were truly necessary and how to group or separate functionality without creating modules that were too abstract or too fragmented. Our initial draft had 15 modules, but after careful discussion and iterative refinement, we narrowed it down to 12 well-defined modules that better reflect the system's actual behaviour and design principles. We resolved this by comparing responsibilities, identifying overlaps, and merging modules where appropriate while ensuring each remaining module had a clear secret, purpose, and boundary.
- [**Burhanuddin Kharodawala**]**:** The document had a few ambiguous instructions which I had difficulty understanding. The module decomposition section especially was the one thing that was a little confusing. I was unsure of what the expectations were for the module. As in, what is a module (A feature or functionality)? At the end of the TA meeting however, I was able to discuss this with the TA and confirm the expectations for the modules.

3. Which of your design decisions stemmed from speaking to your client(s) or a proxy (e.g. your peers, stakeholders, potential users)? For those that were not, why, and where did they come from?

   As a team, many of our design choices originated from discussions with peers and user proxies. The idea to create a McMaster-only ridesharing platform emerged from user concerns around safety and trust. Our matching approach, including the swipe-based interface, was influenced by peer feedback indicating that a familiar interaction model would improve adoption. Other decisions—such as using tRPC, BetterAuth, and Drizzle ORM—came from internal technical reasoning. These choices focused on maintainability, type safety, and long-term scalability rather than direct stakeholder input.

4. While creating the design doc, what parts of your other documents (e.g. requirements, hazard analysis, etc), it any, needed to be changed, and why?

   During the MIS development, the team identified minor inconsistencies between the Module Guide, SRS, and earlier architectural descriptions. Some module names were standardized to ensure consistent referencing, and inter-module relationships were clarified, particularly for Matching, Scheduling, and Notification. A few requirement identifiers were refined to improve traceability. No core requirements changed, but descriptions were tightened to align with the finalized MIS structure.

5. What are the limitations of your solution? Put another way, given unlimited resources, what could you do to make the project better? (LO_ProbSolutions)

   The current solution, while well-structured, is limited by capstone scope and development time. Our matching algorithm is deterministic and rule-based; with more resources, the team would implement machine-learning-driven adaptive matching. Additional improvements could include:

   - fully implemented CI/CD pipelines for automated testing and deployment,
   - real-time route optimization using live traffic data,
   - enhanced accessibility and UX refinement,
   - automated moderation tools and trust scoring systems,
   - end-to-end encryption and advanced safety features.

   These expansions would elevate Hitchly from a functional prototype to a production-ready system.

6. Give a brief overview of other design solutions you considered. What are the benefits and tradeoffs of those other designs compared with the chosen design? From all the potential options, why did you select the documented design?

   The team considered multiple architectural and implementation approaches. One option was to use a REST-based backend instead of tRPC, but this offered weaker type safety across the mobile app and API. Another option was building a native iOS/Android app, but Expo provided faster development and easier integration within the monorepo. The team also explored using a monolithic architecture versus a modular service structure. Ultimately, the chosen modular design offered the best balance between clarity, extensibility, and meeting course expectations. Alternative matching designs, such as a manually curated list instead of a scoring algorithm, were rejected because they reduced flexibility and diminished the quality of the user matching experience.

   (LO_Explores)

References

Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, Upper Saddle River, NJ, USA, 2nd edition, 2003.

Daniel M. Hoffman and Paul A. Strooper. *Software Design, Automated Testing, and Maintenance: A Practical Approach*. International Thomson Computer Press, New York, NY, USA, 1995. URL http://citeseer.ist.psu.edu/428727.html.

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.