M3 - Requirements and Design

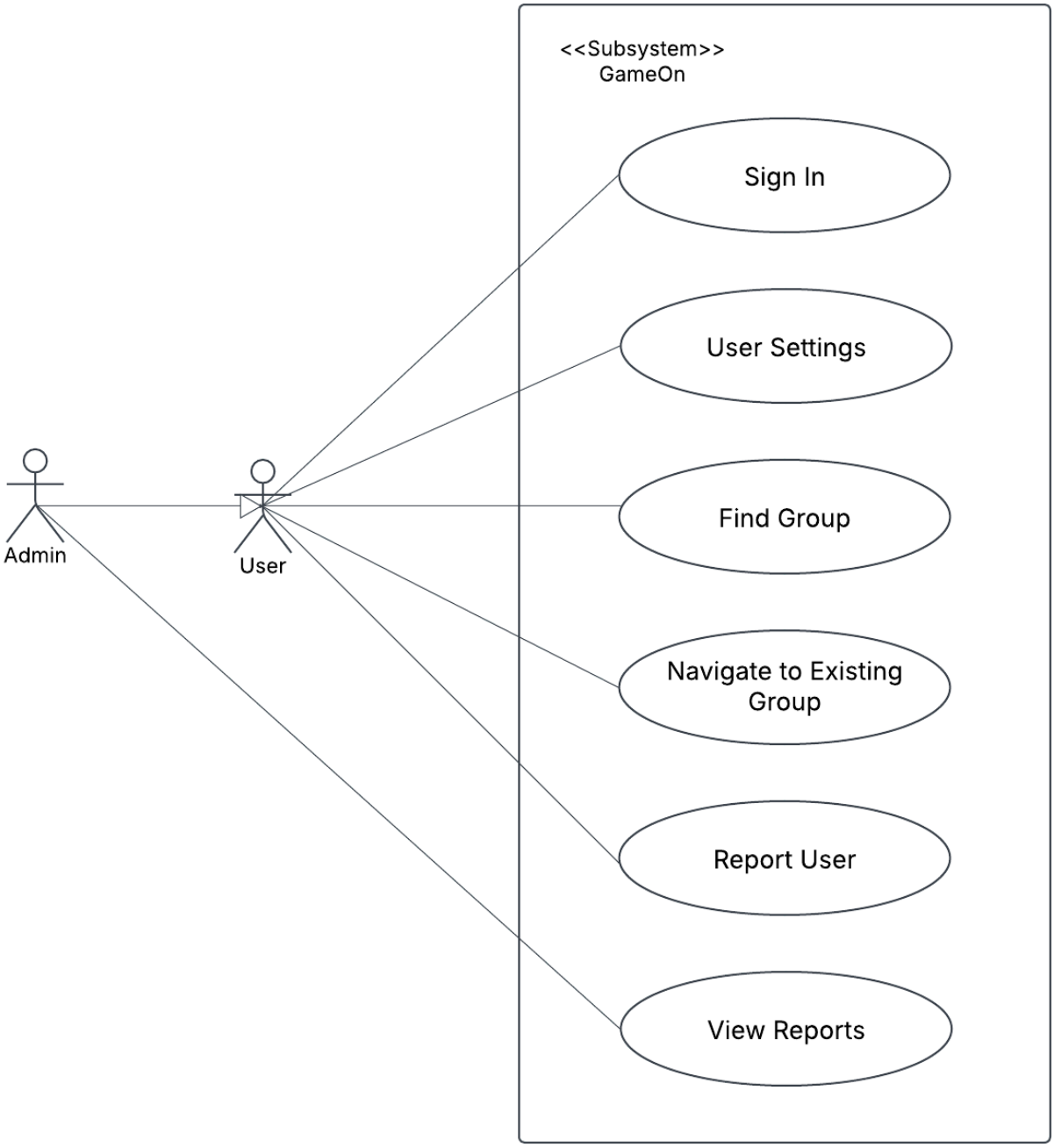
1. Change History

2. Project Description

GameOn is a social matchmaking platform designed for gamers to find ideal teammates and build lasting connections. By authenticating through Discord, players create personalized profiles, sharing details like preferred games, skill levels, communication styles, and playstyles. The app intelligently matches players based on their preferences, instantly creating a dedicated Discord group for seamless in-game coordination and ongoing communication. With integrated feedback systems, including reviews and ratings, GameOn fosters a supportive and positive gaming community.

3. Requirements Specification

3.1. Use-Case Diagram



3.2. Actors Description

1. **User**: A player who uses Discord authentication to access the app. They can set preferences, join groups via group matching, interact with other users, and report users if needed.
2. **Admin**: A system administrator who monitors user reports and has the authority to ban users if necessary, ensuring a safe and fair community environment.

3.3. Functional Requirements

1. **Sign In**
    - **Overview**:
        1. Login
        2. Register

- **Detailed Flow for Each Independent Scenario**:

    1. **Login**:

        - **Description**: Users are prompted to sign in with Discord. If they have already signed in with Discord they are taken to the home screen.
        - **Primary actor(s)**: User
        - **Main success scenario**:
            1. Successful sign-in via Discord authentication.
        - **Failure scenario(s)**:
            - 1a. Unsuccessful sign-in via Discord authentication.
                - 1a1. Incorrect credentials are provided.

    2. **Register**:

        - **Description**: Users are prompted to sign in with Discord. If they have not they are taken to the account creation screen and are prompted to enter a screen name, upload an avatar, provide spoken language preference(s), time zone, and date of birth.
        - **Primary actor(s)**: User
        - **Main success scenario(s)**:
            1. Successful sign-in via Discord authentication.
            2. Actor successfully fills out form submission and account is registered.
        - **Failure scenario(s)**:
            - 1a. Unsuccessful sign-in via Discord authentication.
                - 1a1. Incorrect credentials are provided.
            - 2a. User fails to provide all required fields.
                - 2a1. User must be prompted to fill in missing fields.
            - 2b. User uploads an incorrect image format for their avatar.
                - 2b1. User must be prompted to upload an image of all supported filetypes.
            - 2c: User leaves and closes application before all imformation is filled. Registration does not occur.

2. **User Settings**

- **Overview**:

    1. View and Change Settings

- **Detailed Flow for Each Independent Scenario**:

    1. **View and Change Settings**:
        - **Description**: When clicking their avatar in the top right of the screen, users are taken to their account settings page, where they can change their avatar, spoken language preferences, and time zone.
        - **Primary actor(s)**: User
        - **Main success scenario**:
            1. Actor presses submit on user settings and data is updated in the User database.

        - **Failure scenario(s)**:
            - 1a. Actor leaves this screen without submitting, no settings are updated.

3. **Find Group**

    - **Overview**:

        1. Looking for Group

    - **Detailed Flow for Each Independent Scenario**:

        1. **Looking for Group**:
            - **Description**: When clicking the “Find Group” button in the app, you will be prompted to select a game you wish to play. Indicate how important matching language preference, chattiness, time zone, and age range is to you. Once you have entered the data, you will click a confirm dialogue and be transferred to a queue/loading screen.
            - **Primary actor(s)**: User
            - **Main success scenario**:
                1. Once a group is found, we create a Discord server for the group and it is added to the Group's database. Actor is transferred to the "Existing Group" screen.
            - **Failure scenario(s)**:
                - 1a. If there are not enough people matching your preferences in the queue, after a defined timeout users will be kicked from the queue and returned to the “Find Group” screen.

4. **Navigate to Existing Group**

    - **Overview**:

        1. Navigate to Existing Group Page

    - **Detailed Flow for Each Independent Scenario**:

        1. Navigate to Existing Group Page:
            - **Description**: When a user clicks on an existing group from the “My Groups” banner, they are taken to the existing group page.
            - **Primary actor(s)**: User
            - **Main success scenario**:
                1. Actor presses on an existing group and navigates successfully to "Existing Group" screen.
            - **Failure scenario(s)**:
                - 1a. Group not found in database, user is not routed to existing group page.

5. **Report User**

    - **Overview**:

        1. Submit Report

- **Detailed Flow for Each Independent Scenario**:

    1. **Submit Report**:
        - **Description**: When a user long-presses a user within the "Existing Group" screen, an option appears to “Report User”. Upon clicking “Report User”, a pop-up appears where a user can write a summary on why the user is being reported. Upon clicking “Submit”, the report is submitted to the administrators.
        - **Primary actor(s)**: User, Admin
        - **Main success scenario**:
            1. Actor successfully accesses the option to report the user, clicks the “Report User” button, and fills out a summary of reporting.
            2. Clicking “Submit” successfully sends the summary.
        - **Failure scenario(s)**:
            - 2a. User does not click “Submit” and the report is not submitted to the admin.

6. **View Reports**

- **Overview**:

    1. Ban User
    2. Acquit User

- **Detailed Flow for Each Independent Scenario**:

    1. **Ban User**:

        - **Description**: Admins view and action user reports from the view reports screen. Admin has the option to ban a user based on the summary they've been provided.
        - **Primary actor(s)**: Admin
        - **Main success scenario**:
            1. Actor reviews the report, and bans the offending user.
            2. A user is deleted from the Users database. Their existing groups remain.
        - **Failure scenario(s)**:
            - 1a. No action is taken, the report remains in the admin's report queue.

    2. **Acquit User**:

        - **Description**: Admins view and action user reports from the view reports screen. Admin has the option to *not* ban the user based on the summmary they've been provided.
        - **Primary actor(s)**: Admin
        - **Main success scenario**:
            1. Actor reviews the report, and does not ban the offending user. User's access to application and all groups remains the same.
        - **Failure scenario(s)**:
            - 1a. No action is taken, the report remains in the admin's report queue.

3.4. Screen Mockups

3.5. Non-Functional Requirements

1. **Matchmaking Time**
    - **Description**: Users looking for a group should be matched within 10 minutes of initiating the matchmaking process.
    - **Justification**: Reduces user frustration and ensures a smooth experience when matchmaking. Long wait times could make users not want to use our app.

2. **Arbitrary Group Limit**
    - **Description**: Users should not be limited to an arbitrary number of groups. They should always have the option to find a new group and the technology should support this.
    - **Justification**: Many gamers play a variety of games, and arbitrary group limitations could frustrate users.

4. Designs Specification

4.1. Main Components

1. **Authentication**

    - **Purpose**: Provides secure authentication for users accessing the app. Creates a session token and signs in through the discord API. Calls the SignIn interface with this information.
    - **Interfaces**:
        1. `public String discordAuthenticate(DiscordParams discordParams);`
            - **Purpose**: Returns discord token if the user has logged in with discord

2. **Sign In**

    - **Purpose**: Determines whether a user has an existing account in the database, allowing them to log in, or if they need to register a new account.
    - **Interfaces**:
        1. `public signIn(String discordToken, String sessionToken);`
            - **Purpose**: Uses the UserManagement interface to determine if the user exists in our database or not, and routes to the home page or registration page.

3. **User Management**

    - **Purpose**: Enables users to manage their profiles, update preferences, and modify personal settings.
    - **Interfaces**:
        1. `public boolean createProfile(UserProfile userArgs);`
            - **Purpose**: Creates a user profile.
        2. `public boolean updateProfile(String discordToken, UserProfile profile);`
            - **Purpose**: Updates the user's profile information.
        3. `public UserProfile getProfile(String discordToken);`
            - **Purpose**: Retrieves the user's profile details from DB.
        4. `public boolean deleteProfile(String discordToken);`
            - **Purpose**: Permanently deletes a user's account.

4. **Matchmaking**

    - **Purpose**: Connects users with similar preferences to form gaming groups.

- **Interfaces**:
    1. `public String matchmaking (MatchmakingRequest matchmakingRequest);`
        - **Purpose**: Based on matchmaking preferences will create a group in discord with other users. Returns group id.

5. **Group Management**

    - **Purpose**: Facilitates the creation and management of groups within the app and on Discord.
    - **Interfaces**:
        1. `public boolean createGroup(GroupArgs groupArgs);`
            - **Purpose**: Creates a group.
        2. `public boolean updateGroup(String groupId, Group group);`
            - **Purpose**: Updates the Group's information.
        3. `public Group getGroup(String groupId);`
            - **Purpose**: Retrieves the group details from DB.
        4. `public boolean deleteGroup(String groupId);`
            - **Purpose**: Permanently deletes a group.

6. **Report Management**

    - **Purpose**: Allows users to report other users for rule violations.
    - **Interfaces**:
        1. `public boolean submitReport(String discordTokenReporter, String discordTokenReportee, String reason);`
            - **Purpose**: Submits a report against a user with a given reason.

7. **Admin Management**

    - **Purpose**: Allows admins to monitor reports, ban rule violators, ensure a safe community, promote other users to admin, and add supported games.
    - **Interfaces**:
        1. `public boolean banUser(String discordTokenAdmin, String discordTokenReportee);`
            - **Purpose**: Bans a reported user from our app.
        2. `public boolean resolveReport(String reportId, String discordTokenAdmin, String discordTokenReportee, String reasoning);`
            - **Purpose**: Resolves a report by providing a resolution reason.
        3. `public List<Reports> getReports();`
            - **Purpose**: Retrieves a list of all submitted reports.
        4. `public boolean promoteToAdmin(String discordTokenAdmin, String discordTokenAdminToPromote);`
            - **Purpose**: Promotes a user to admin privileges.
        5. `public boolean addGame(GameInformation gameInformation);`
            - **Purpose**: Adds a new game to the system.

4.2. Databases

1. **GameOnDB (MySQL)**

- **Purpose**: It will store four main tables: user profile information, matchmaking/group details, session details and reports on users who have violated application guidelines. We will use SQL since this data is relational and well-suited to our needs.

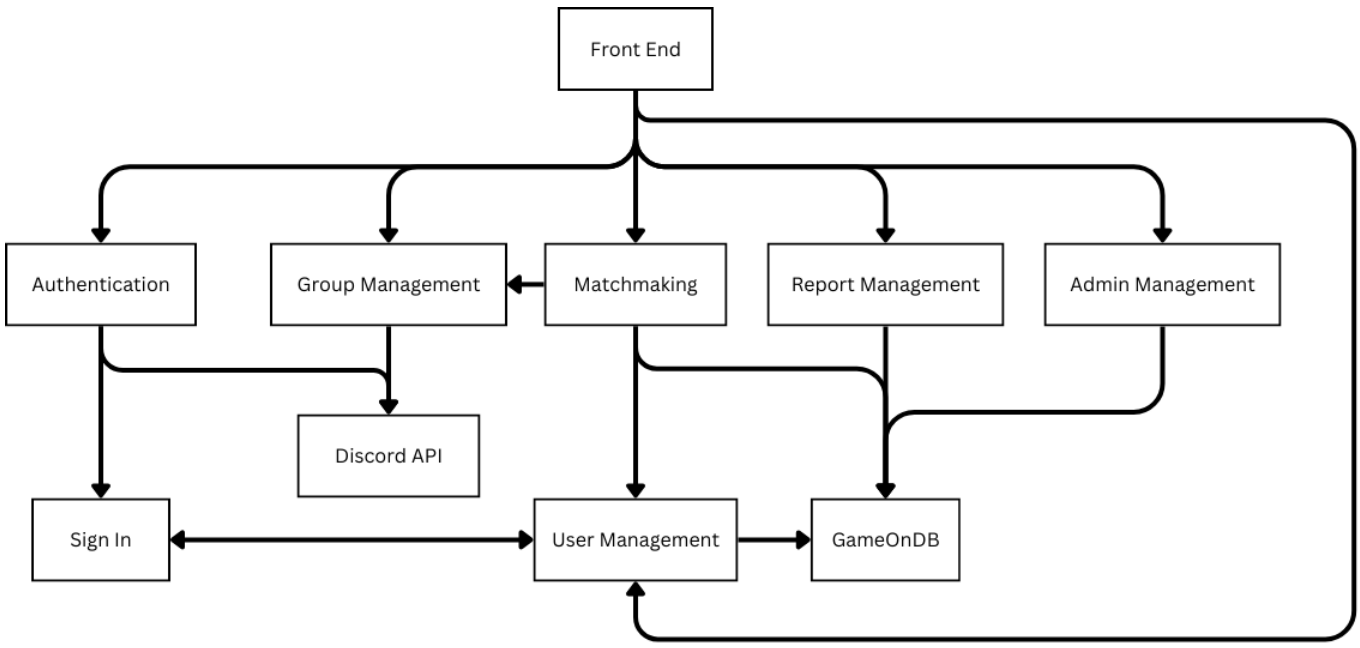
4.3. External Modules

1. **Discord API**
   - **Purpose**: We will use the Discord API for authentication within the app and for creating matchmaking groups. The groups will be created on Discord, as gamers widely use the platform and have a large, active community.

4.4. Frameworks

1. **Android Studio**
   - **Purpose**: Developing the Android front-end UI of the application.
   - **Reason**: Course requirement.
2. **Node.js**
   - **Purpose**: Developing the back-end server of the application.
   - **Reason**: Course requirement.
3. **Azure Virtual Machine**
   - **Purpose**: We are using a VM for hosting the back-end server.
   - **Reason**: Azure is the most financially viable option with a student account.
4. **Azure Database for MySQL**
   - **Purpose**: We will use a MySQL server to store our data.
   - **Reason**: We will use a SQL database since this data is relational and well-suited to our needs. It remains in the same cloud service as our VM.

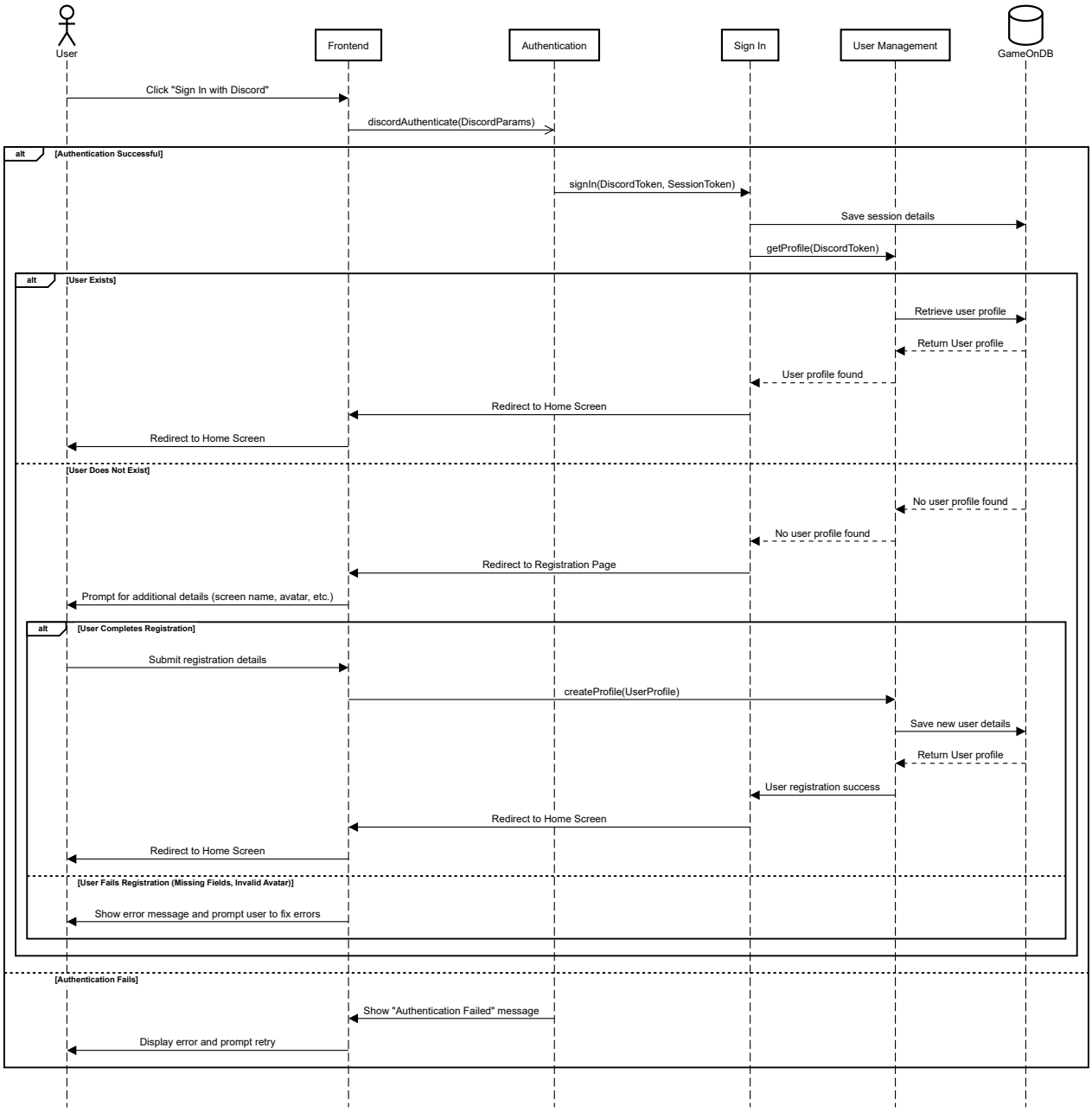
4.5. Dependencies Diagram



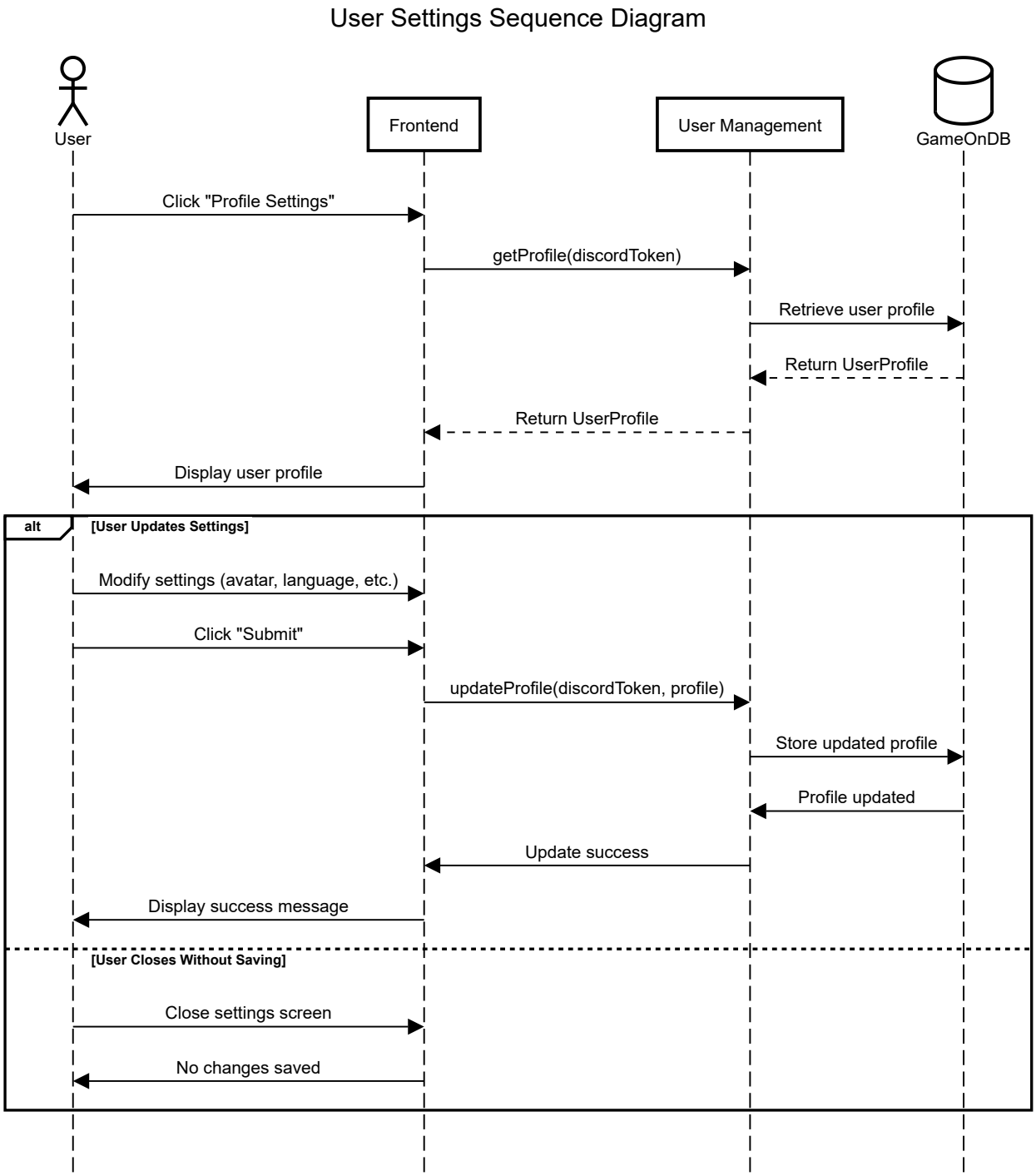
4.6. Functional Requirements Sequence Diagram

1. **Sign In**

Sign In Sequence Diagram

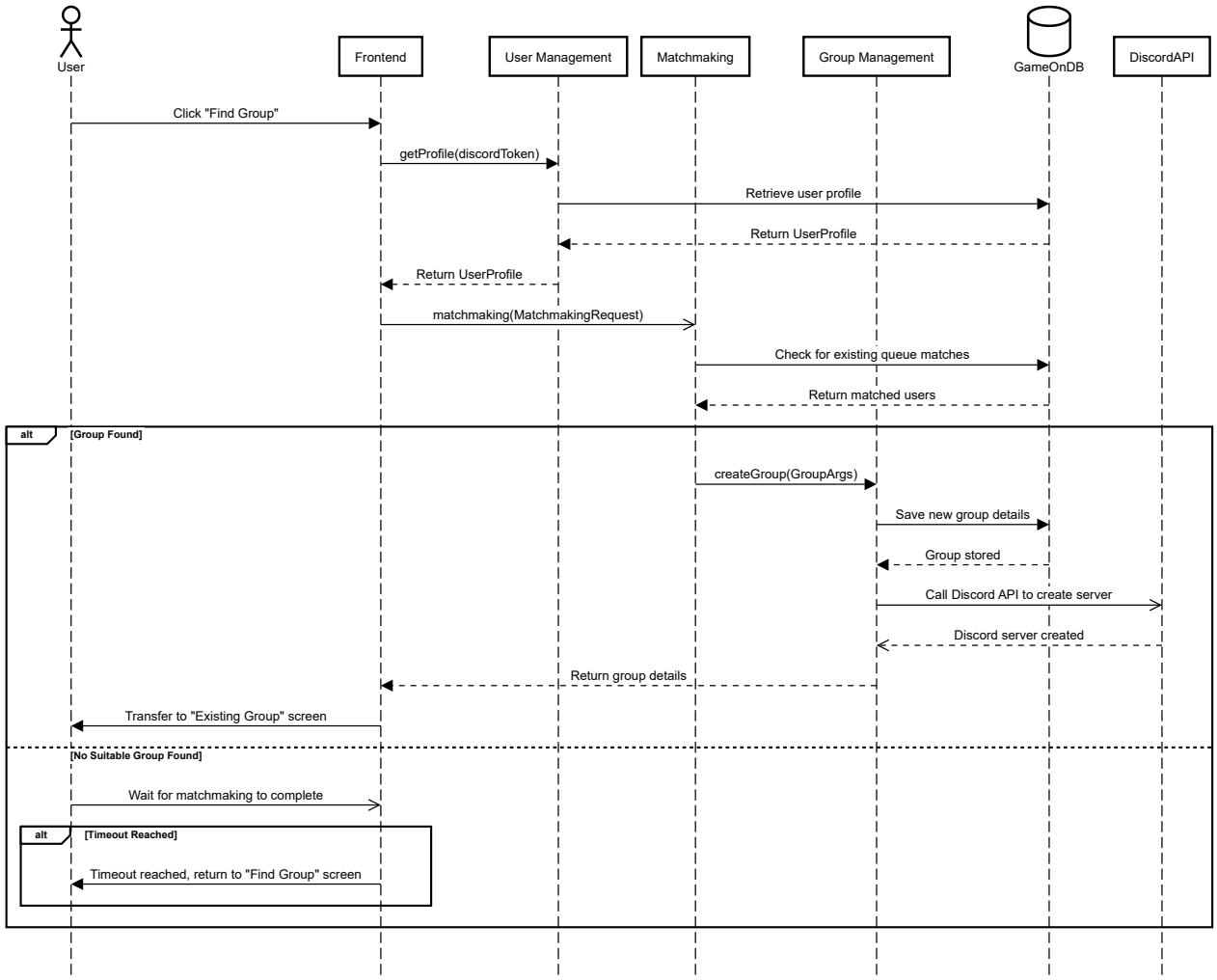


2. **User Settings**

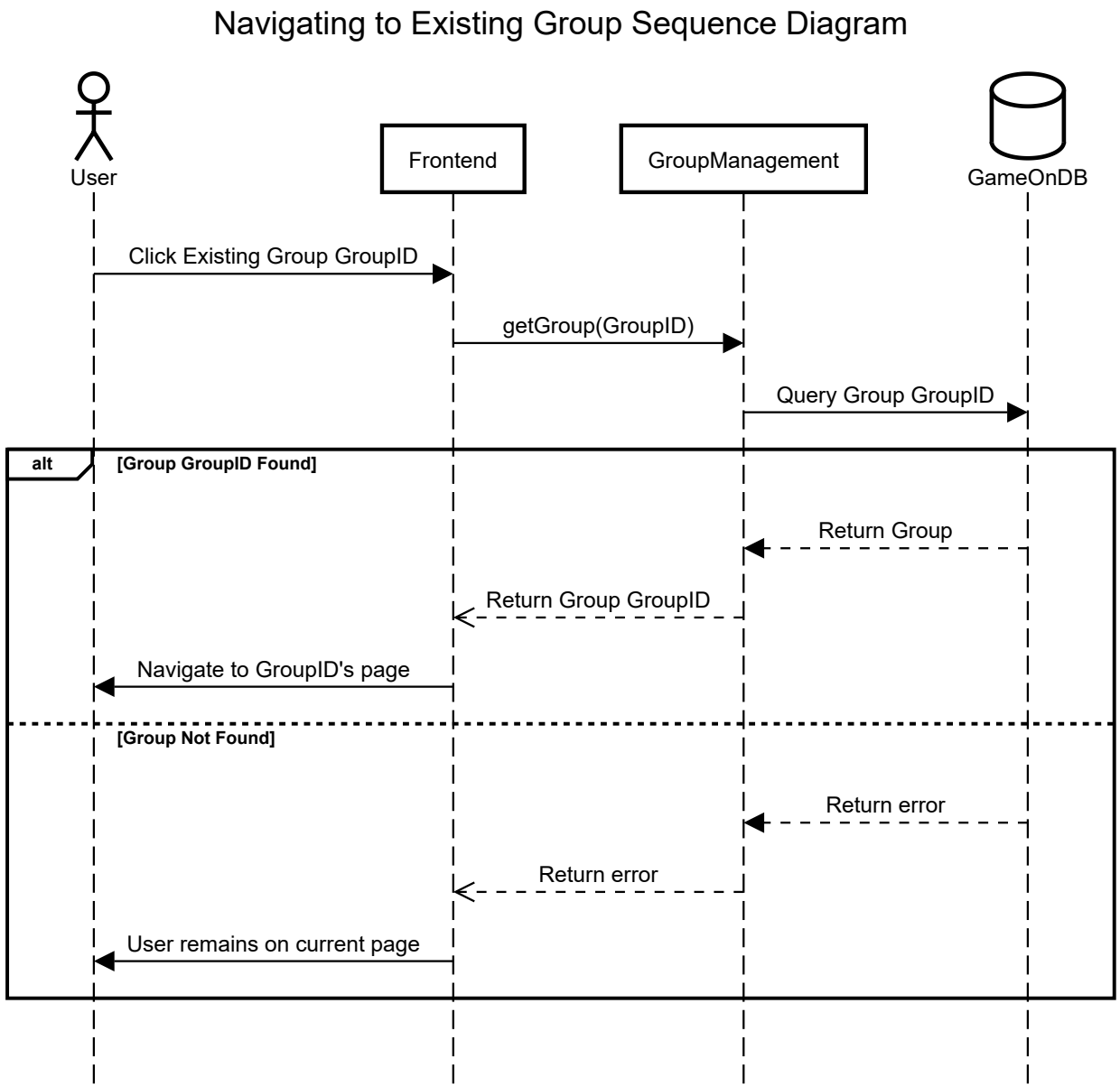


3. **Find Group**

Find Group Sequence Diagram

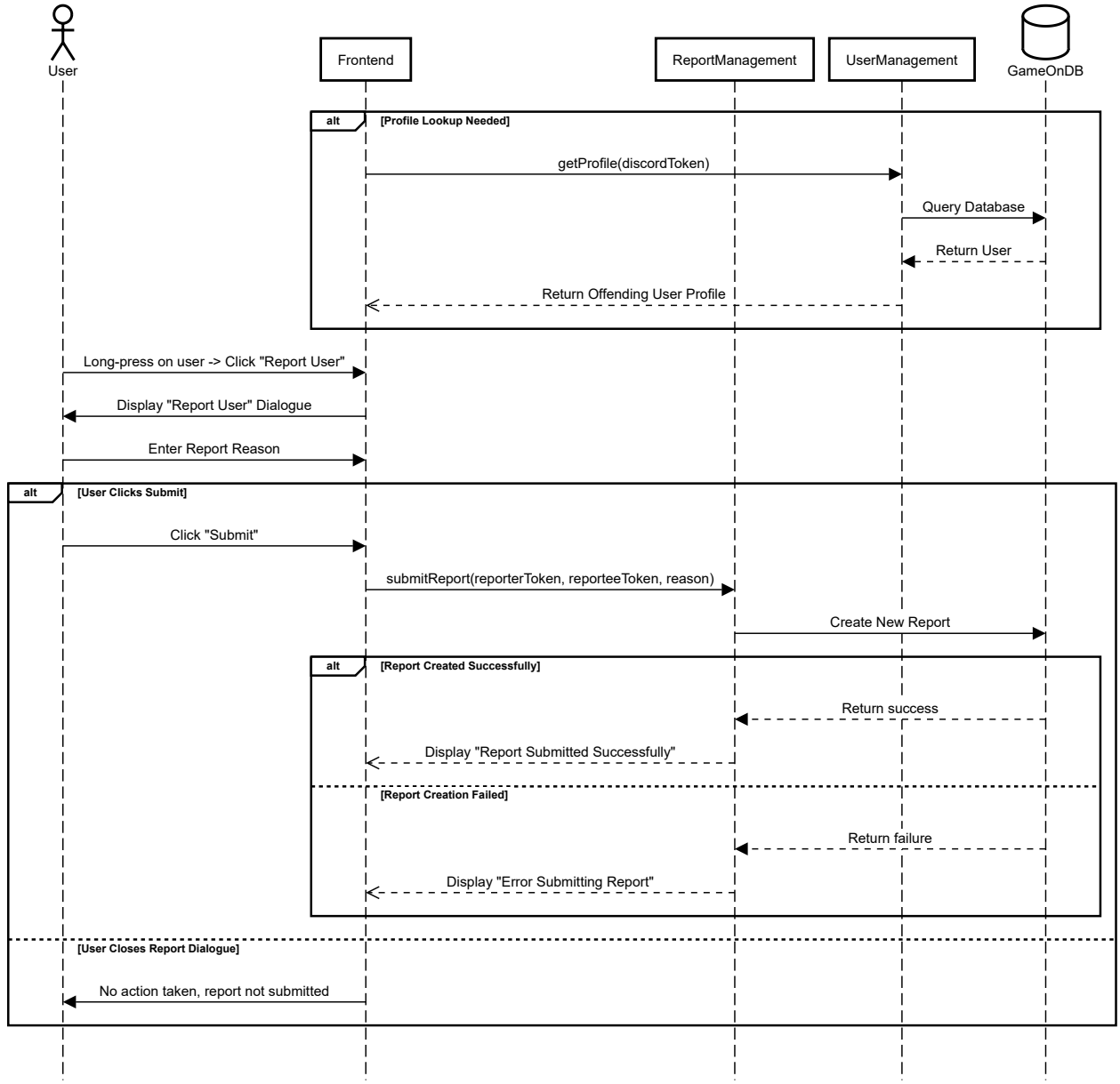


4. **Navigate to Existing Group**

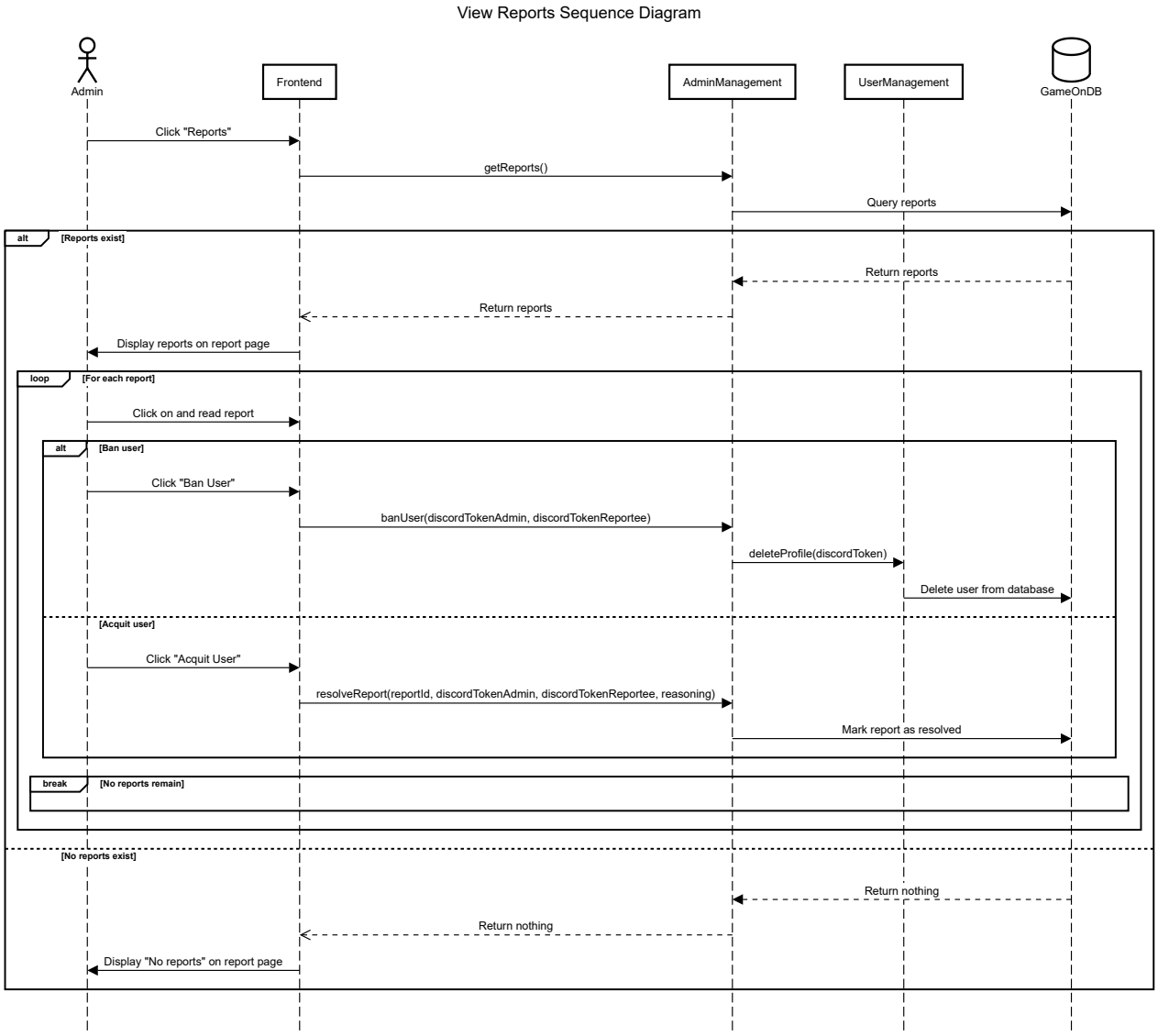


5. **Report User**

Report User Sequence Diagram



6. **View Reports**



4.7. Non-Functional Requirements Design

1. **Matchmaking Time**
   - **Validation**: We will ensure matchmaking times stay under 10 minutes by monitoring real-time user activity and dynamically adjusting matching criteria if needed. If a group is not found within 10 minutes, the system will automatically time out and notify the user to try again later.
2. **Arbitrary Group Limit**
   - **Validation**: Our MySQL database on Azure will auto-scale, optimizing read/write operations to support unlimited group participation.

4.8. Main Project Complexity Design

Group-Based Gaming Matchmaking Using Gale-Shapley (No Ranked User Preferences)

- **Description**: This system matches users into stable gaming groups based on preferences such as **language**, **time zone**, and **game choice**. The **Gale-Shapley algorithm** is adapted to form groups **without ranking individual users**. Instead, users propose to groups that match their preferences, and groups accept users until they reach capacity.
- **Why complex?**:

- **Many-to-many matching**: Unlike traditional Gale-Shapley (one-to-one), this involves grouping multiple users.
- **Dynamic group formation**: Instead of ranking users, groups fill up based on availability, requiring a mechanism for reassignment.
- **Fairness and stability**: Users must be placed in the best possible group without needing an explicit ranking system.
- **Real time matchmaking**: The algorithm must match people in real-time as they join or leave the queue.

- **Design**:
  - **Input**:
    1. A list of users with:
       - Preferred **language**
       - Preferred **time zone**
       - Preferred **game**
    2. A list of groups with:
       - Max **group size**
       - List of **current members**
  - **Output**: A list of **stable gaming groups**, ensuring that all users are placed in a group that aligns with their preferences.
  - **Main computational logic**:
    1. **Initialize empty groups** based on unique (language, time zone, game) combinations.
    2. **Users propose** to the first available group matching their preferences.
    3. If a group has space, it **accepts the user**.
    4. If a group is full, the **user searches for the next closest match**.
    5. If no exact match exists, a **fallback mechanism** places users in the closest possible group.
    6. The process continues until **all users are placed**.
  - **Pseudo-code**:

```
function findMatches(users, max_group_size):
    groups = createEmptyGroups(users, max_group_size)  // Initialize
groups based on preferences
    ungrouped_users = users  // Users still searching for a group

    while ungrouped_users is not empty:
        for user in ungrouped_users:
            preferred_group = findFirstAvailableGroup(user, groups)

            if preferred_group is not null:
                addUserToGroup(user, preferred_group)
                ungrouped_users.remove(user)
            else:
                backup_group = findClosestMatchingGroup(user, groups)
// Fallback mechanism
                if backup_group is not null:
```

```
                    addUserToGroup(user, backup_group)
                    ungrouped_users.remove(user)

    return groups
```

5. Contributions

It should be noted that all work for Milestone 3 was completed synchronously, in person in the lab or library so each member has made a strong and equal contribution.