Department of Computer Science, University of Calgary

SENG 300 Introduction to Software Engineering

March 21, 2025

Letter to the Reviewed Team

Dear Bravo Team,

We appreciate the effort put into your design and documentation. Our review highlighted strengths as well as areas for improvement, detailed below.

General Critique of the Design

Your project contains well-structured use case diagrams and descriptions that effectively describe expected system behavior. However, some inconsistencies and gaps in integration need to be addressed:

- **Completeness of Game Diagrams**: Chess and Connect4 include full diagrams, but Go and TicTacToe lack completeness. Go only contains a use case description and diagram, while TicTacToe is missing game logic details.
- **Consistency & Organization**: Diagrams should be compiled together with consistent naming conventions to improve clarity and cross-referencing.
- **Integration Class Diagram Issues**: References to non-existent functions and missing connections make it unclear if different sections are properly linked.
- **Class Diagram Challenges**: Floating, unlinked classes reduce clarity, and overcomplicated diagrams could be simplified where interfaces are used by a single class.
- **System Gaps**: Some components are not designed to interact, such as game logic functions missing in the integration diagram.

Diagram Corrections/Changes

We have annotated several diagrams where necessary changes should be made. The annotated diagrams are attached in :

- **GUI Elements**:
    - Suggested improving the sign-up process by requesting an email for verification. The class diagram appears to be missing update functions for the game display.
- **Authentication Class Diagram**:
    - A class inherits another unnecessarily. Further clarity on function connections would improve readability.
- **Matchmaking Class Diagram**:
    - Lacks specific error handling and a "Cancel Matching" function. It also fails to notify both players when a match is found.
- **Leaderboard and Matchmaking:**
    - Rank attributes need clarification, and dependencies between AbstractGame and MatchmakingLogic require further explanation.
- **Integration Class Diagram**:
    - Lacks connections between key sections such as authentication and game session initiation.
- **Chess Class Diagram**:
    - Needs better handling of special moves and status checks for check and checkmate.
- **Chat Use Case Diagram**:
    - Overlapping arrows reduce readability. Suggested improvements include adding a blocking/reporting system.
- **TicTacToe Use Case Description**:
    - Needs to better address tie handling and player disconnections.
- **Synchronization Use Case Diagram:**
    - Synchronization steps should be expanded to clarify processes and triggers.

Feature Requests

We propose two major improvements to enhance functionality:

1. **Cloud Saving for User Progress and Settings**
    - This feature would involve implementing a cloud-based storage solution so players can sync their game progress, statistics and preferences across devices. Cloud sync prevents data loss, and enables more seamless multi-device usage, enhancing convenience and enabling continuity in gameplay, promoting higher user retention, since progress is never restricted to a single device.
2. **Player Actions: Send Friend Requests & Block Users**
    - This feature would allow users to send friend requests, or block disruptive individuals directly from their player profiles. This would create a more social, as well as positive environment across the game network. Players would be able to

form connections for future matches as well as protect themselves against unwanted interactions

3. **Account Recovery System (Forgot Username/Password)**
    - This feature would involve implementing a simple mechanism that allows users to regain access if they lose or forget their credentials. This feature would be extremely useful as it would lower abandonment rates and ensure a smoother experience for returning players. This would also follow the standard set by most online services.
4. **Accessibility Options for Visually Impaired Users**
    - By implementing accessibility options, a wider array of users would be accounted for. This could include things such as high-contrast themes, colorblind friendly palette, and adjustable text sizes in the settings. Having these features would enhance the platform's reputation, and allow individuals with impairments to enjoy all aspects of the game, such as the rest of the users.

Constructive Feedback

- **Improve Diagram Readability**: Reduce overlapping arrows and use grouping for clarity.
- **Expand User Interactions**: Ensure that all user actions beyond dashboards and menus are detailed, particularly for in-game interactions.
- **Clarify Integration Between Diagrams**: Ensure referenced functions exist across all related diagrams for consistency.
- **Ensure Proper Use of Interfaces**: Avoid redundant interfaces that serve single functions.

Your project demonstrates a strong foundation, and with these adjustments, it will achieve better clarity, usability, and maintainability. Thank you for your hard work! Below are your annotated diagrams for more information on improvements.

Sincerely,
Bravo Review Team

Appendix 1: Annotated GUI Diagrams

Figure 1.1: Dashboard

Figure 1.2: Landing Page

Figure 1.3: Login Page

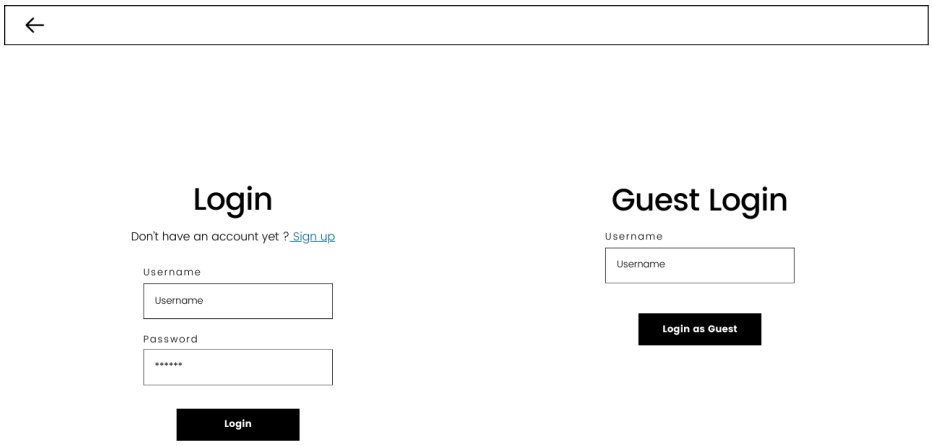


Figure 1.4: Sign Up Page

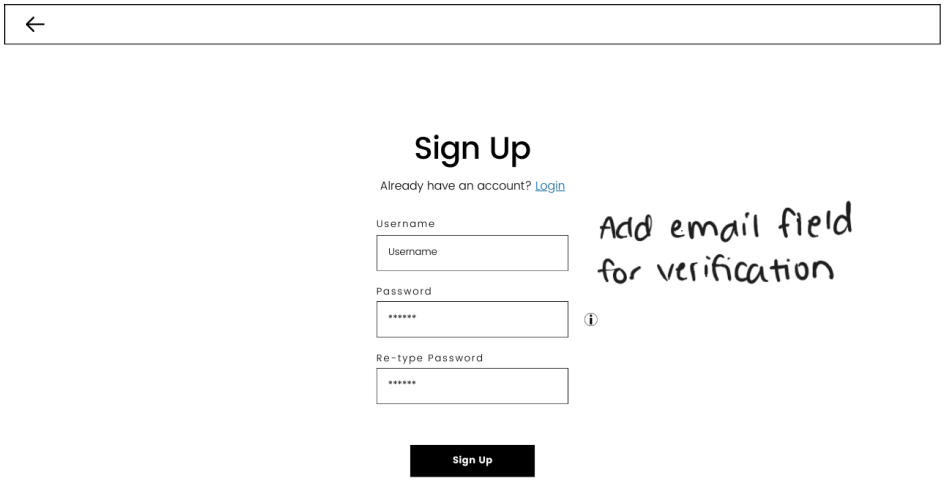


Figure 1.5: Start Page

Figure 1.6: Friends Page

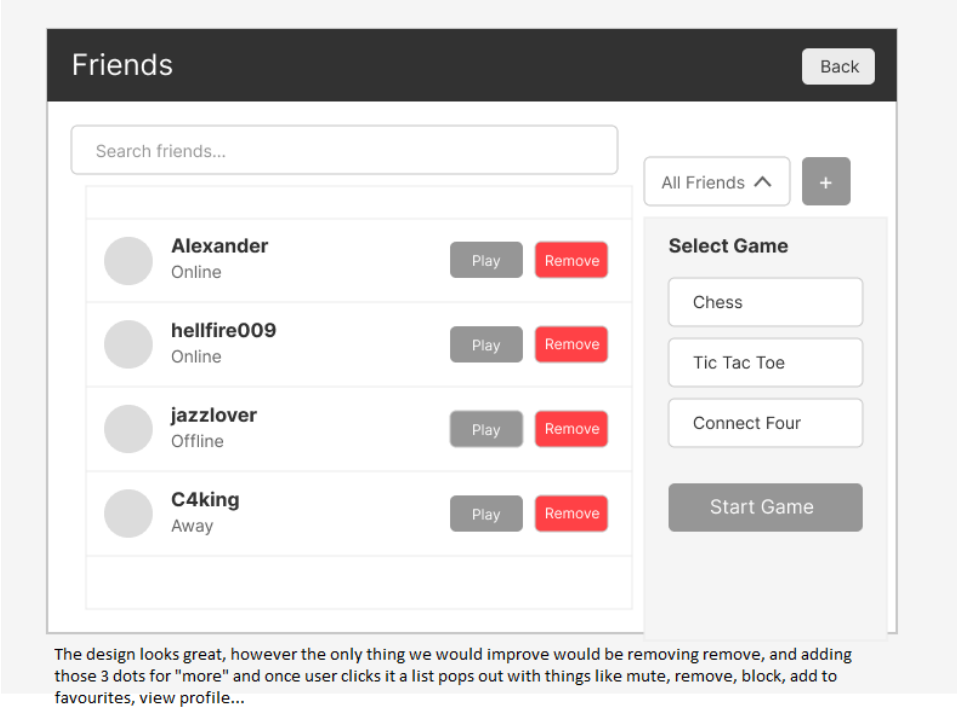


Figure 1.7: Player Lookup Page

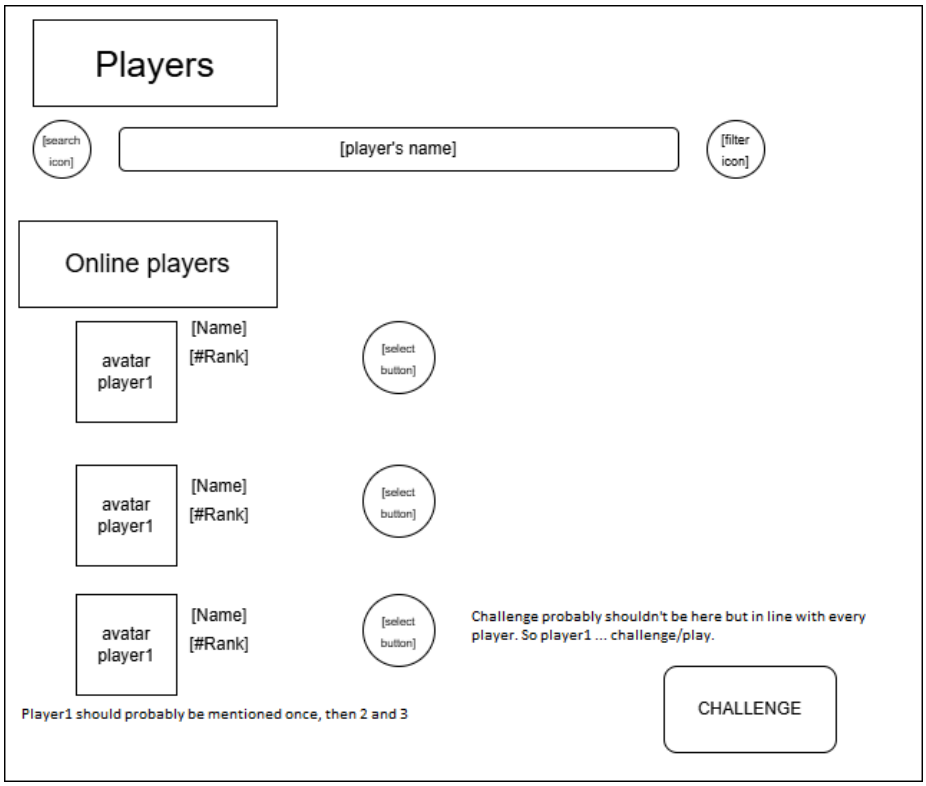


Figure 1.8: Settings Display Page

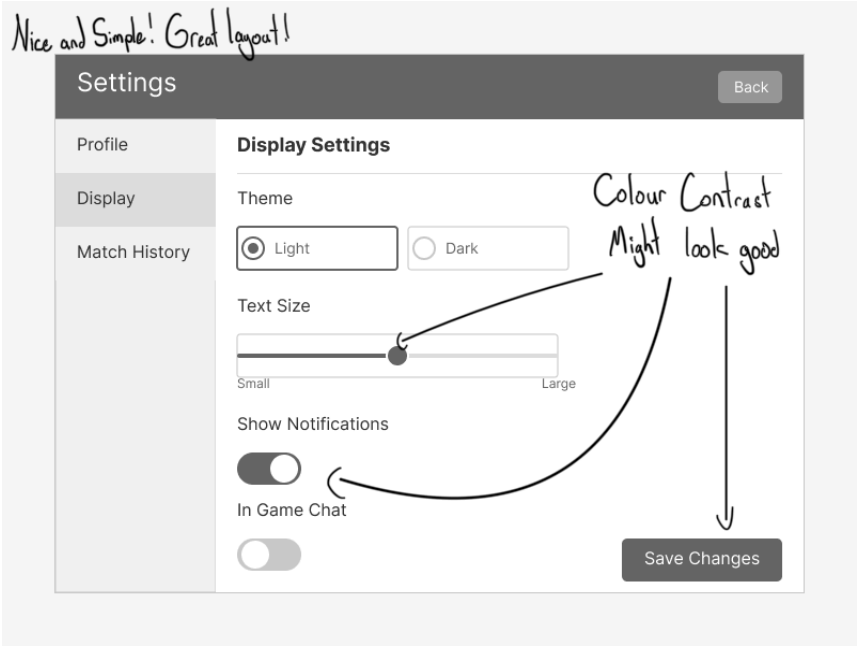


Figure 1.9: Player Profile

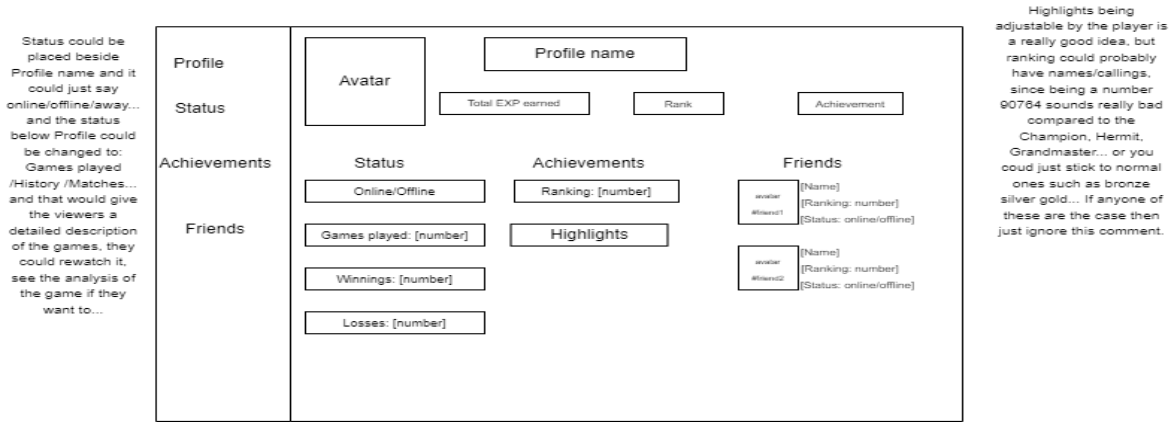


Figure 1.10: Match History Tab

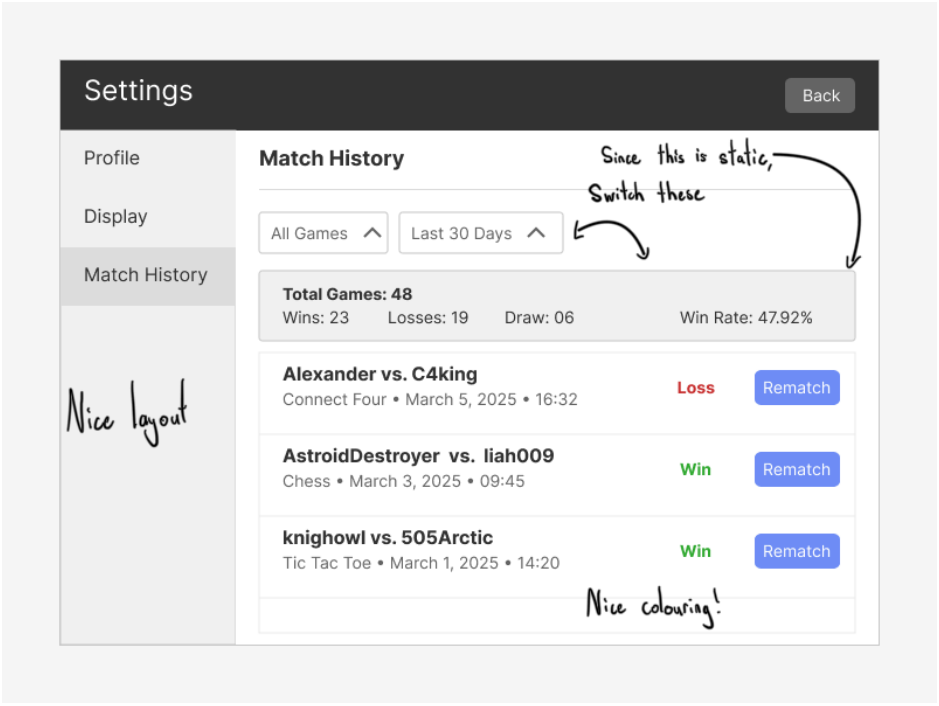
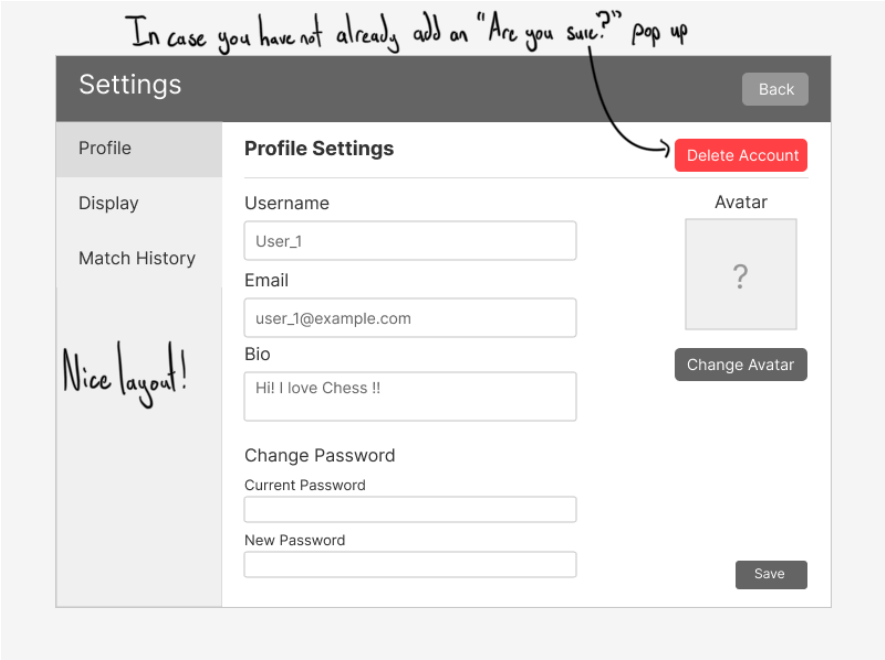


Figure 1.11: Player Profile Tab



Appendix 2: Annotated Class Structure Diagrams

Figure 2.1: Networking Diagram

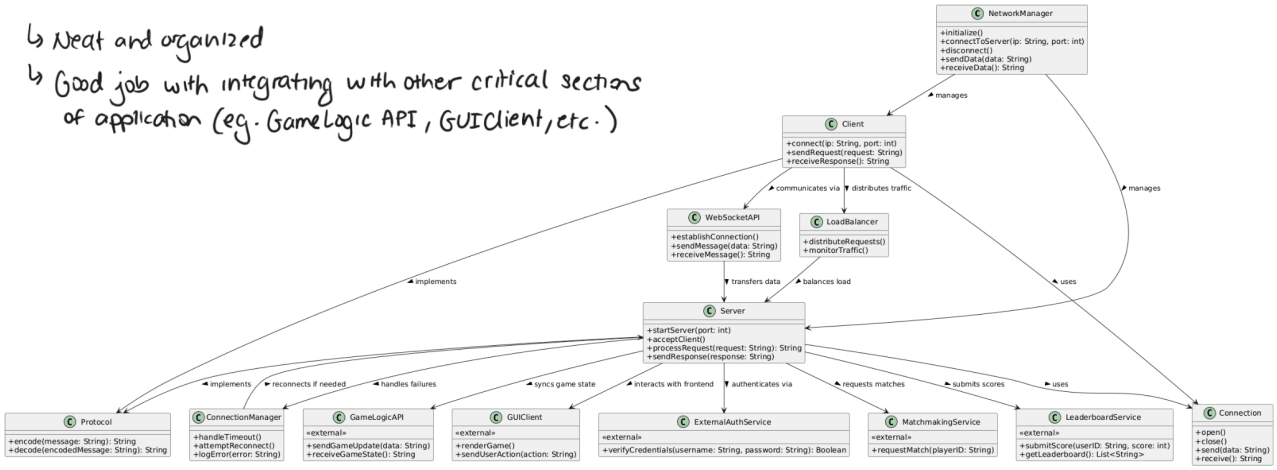


Figure 2.2: TicTacToe Diagram

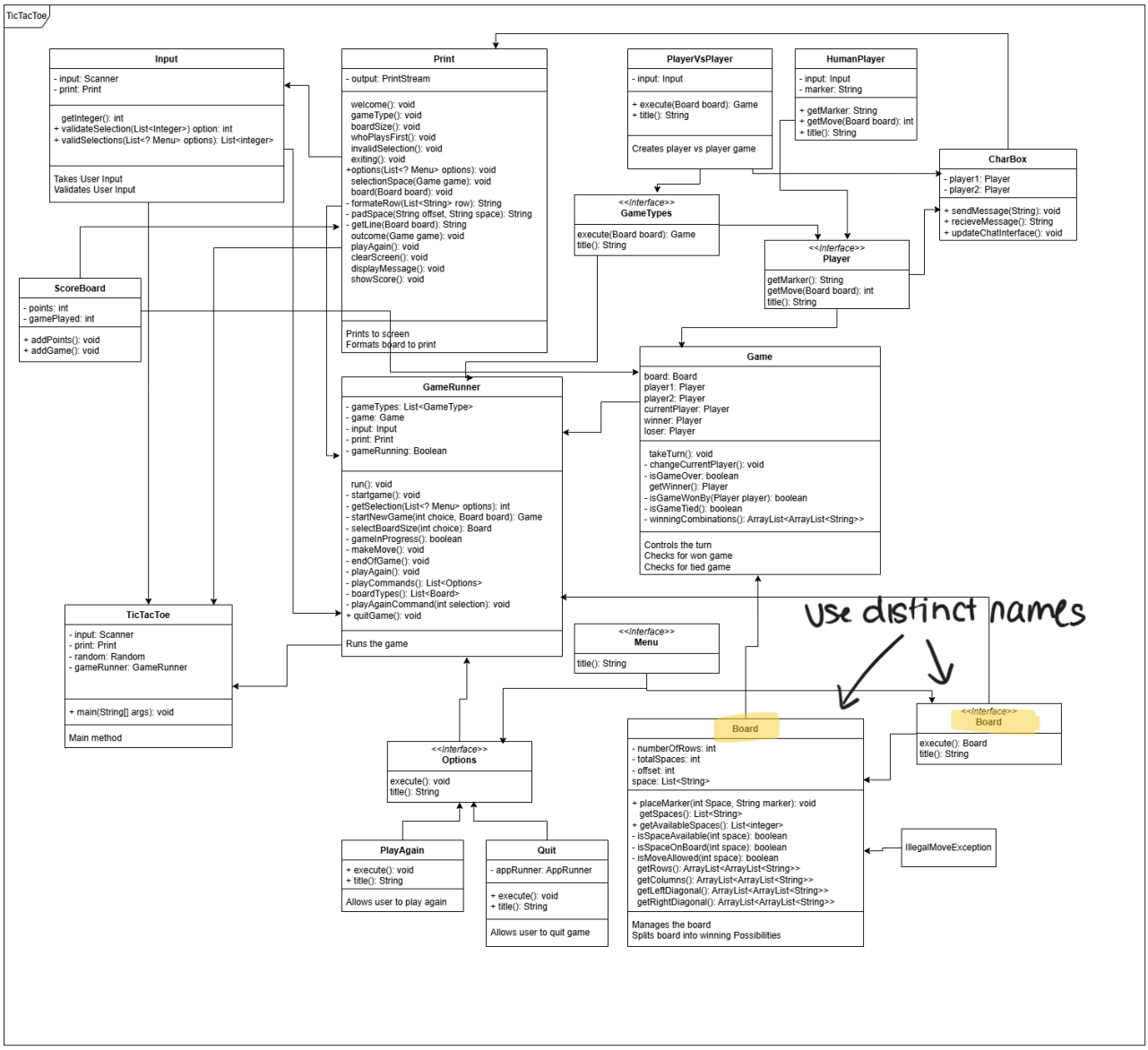


Figure 2.3: Connect4 Diagram

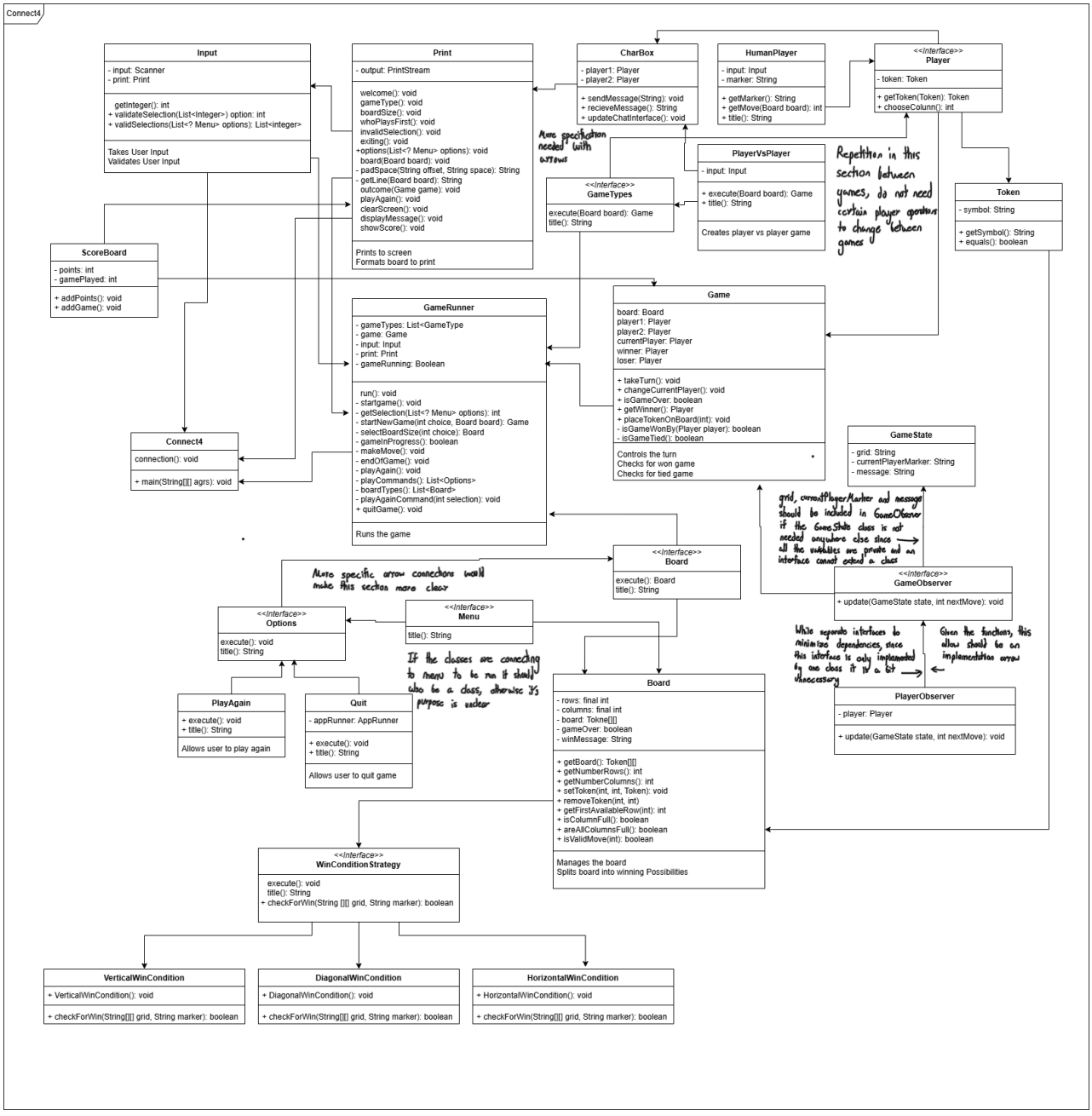

Connect4
Input
- input: Scanner
- print: Print
getInteger(): int
+ validateSelection(List<Integer>) option: int
+ validSelections(List<? Menu> options): List<integer>
Takes User Input
Validates User Input
Print
- output: PrintStream
welcome(): void
gameType(): void
boardSize(): void
whoPlaysFirst(): void
invalidSelection(): void
exiting(): void
+options(List<? Menu> options): void
board(Board board): void
- padSpace(String offset, String space): String
- getLine(Board board): String
outcome(Game game): void
playAgain(): void
clearScreen(): void
displayMessage(): void
showScore(): void
Prints to screen
Formats board to print
CharBox
- player1: Player
- player2: Player
+ sendMessage(String): void
+ recieveMessage(): String
+ updateChatInterface(): void
HumanPlayer
- input: Input
- marker: String
+ getMarker(): String
+ getMove(Board board): int
+ title(): String
<<Interface>>
Player
- token: Token
+ getToken(Token): Token
+ chooseColunn(): int
More specification needed with arrows
PlayerVsPlayer
- input: Input
+ execute(Board board): Game
+ title(): String
Creates player vs player game
Repetition in this section between games, do not need certain player options to change between games
<<Interface>>
GameTypes
execute(Board board): Game
title(): String
Token
- symbol: String
+ getSymbol(): String
+ equals(): boolean
ScoreBoard
- points: int
- gamePlayed: int
+ addPoints(): void
+ addGame(): void
GameRunner
- gameTypes: List<GameType
- game: Game
- input: Input
- print: Print
- gameRunning: Boolean
run(): void
- startgame(): void
- getSelection(List<? Menu> options): int
- startNewGame(int choice, Board board): Game
- selectBoardSize(int choice): Board
- gameInProgress(): boolean
- makeMove(): void
- endOfGame(): void
- playAgain(): void
- playCommands(): List<Options>
- boardTypes(): List<Board>
- playAgainCommand(int selection): void
+ quitGame(): void
Runs the game
Game
board: Board
player1: Player
player2: Player
currentPlayer: Player
winner: Player
loser: Player
+ takeTurn(): void
+ changeCurrentPlayer(): void
+ isGameOver: boolean
+ getWinner(): Player
+ placeTokenOnBoard(int): void
- isGameWonBy(Player player): boolean
- isGameTied(): boolean
Controls the turn
Checks for won game
Checks for tied game
Connect4
connection(): void
+ main(String[][] agrs): void
GameState
- grid: String
- currentPlayerMarker: String
- message: String
grid, currentPlayerMarker and message should be included in GameObserver if the GameState class is not needed anywhere else since all the variables are private and an interface cannot extend a class
<<Interface>>
Board
execute(): Board
title(): String
<<Interface>>
GameObserver
+ update(GameState state, int nextMove): void
More specific arrow connections would make this section more clear
<<Interface>>
Options
execute(): void
title(): String
<<Interface>>
Menu
title(): String
While seperate interfaces do minimize dependencies, since this interface is only implemented by one class it is a bit unnecessary
Given the functions, this allow should be an implementation arrow
If the classes are connecting to menu to be run it should also be a class, otherwise it's purpose is unclear
PlayObserver
- player: Player
+ update(GameState state, int nextMove): void
PlayAgain
+ execute(): void
+ title(): String
Allows user to play again
Quit
- appRunner: AppRunner
+ execute(): void
+ title(): String
Allows user to quit game
Board
- rows: final int
- columns: final int
- board: Tokne[][]
- gameOver: boolean
- winMessage: String
+ getBoard(): Token[][]
+ getNumberRows(): int
+ getNumberColumns(): int
+ setToken(int, int, Token): void
+ removeToken(int, int)
+ getFirstAvailableRow(int): int
+ isColumnFull(): boolean
+ areAllColumnsFull(): boolean
+ isValidMove(int): boolean
Manages the board
Splits board into winning Possibilities
<<Interface>>
WinConditionStrategy
execute(): void
title(): String
+ checkForWin(String [][] grid, String marker): boolean
VerticalWinCondition
+ VerticalWinCondition(): void
+ checkForWin(String[][] grid, String marker): boolean
DiagonalWinCondition
+ DiagonalWinCondition(): void
+ checkForWin(String[][] grid, String marker): boolean
HorizontalWinCondition
+ HorizontalWinCondition(): void
+ checkForWin(String[][] grid, String marker): boolean

Figure 2.4: Authentication Diagram

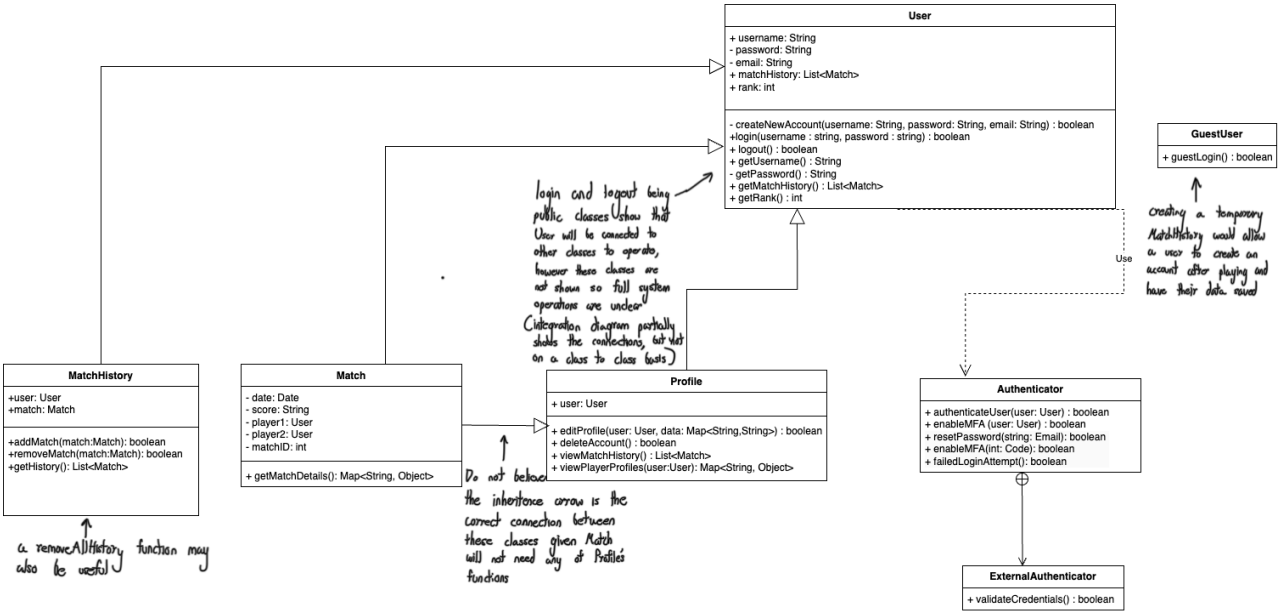


Figure 2.5: Leaderboard and Matchmaking Diagram

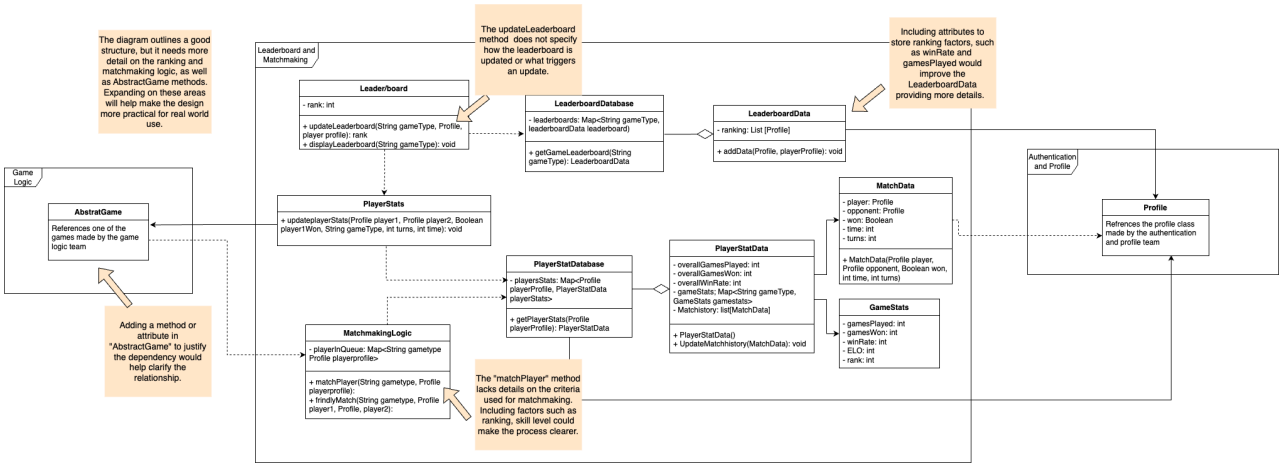
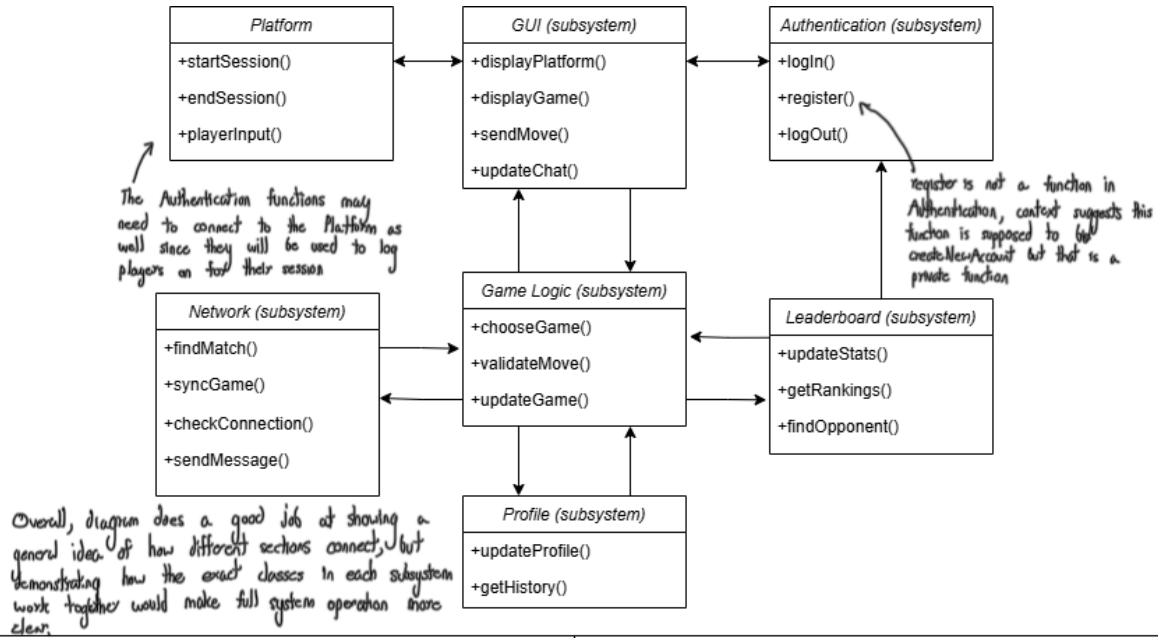


Figure 2.6: Integration Diagram



Platform	Authentication
Methods: startSession(): void Initializes the platform and all subsystems. endSession(): void Closes the session and shuts down subsystems. playerInput(input: string): void Routes player actions to the appropriate subsystem.	**Methods:** logIn(username: string, password: string): boolean Authenticates the player. register(username: string, password: string): boolean Registers a new player. logOut(): void Ends the player's session.
GUI	**Networking**
Methods: displayGame(gameState: GameState): void Renders the game board and interface. updateChat(message: string): void Displays chat messages from the opponent. sendMove(move: string): void Sends the player's move to the GameLogicManager. Functions shown not matching GUI diagram	**Methods:** findMatch(playerID: string): string Searches for an opponent using the LeaderboardManager. sendMessage(message: string): void Sends chat messages to the opponent. syncGame(gameState: GameState): void Synchronizes the game state between players. checkConnection(): boolean Verifies the player's internet connection. Having these functions match the class diagram in terms of function name and also parameter variable type will make it more clear how they connect
• Game Logic	
Methods: validateMove(move: string): boolean Ensures the player's move is valid. updateGame(move: string): GameState Updates the game state after a move. endGame(): Result Determines the winner and sends results to LeaderboardManager. This would be more clear if these exact functions were included in all of the game class diagrams	**Leaderboard and Matchmaking**
Profile	**Methods:** updateStats(playerID: string, result: Result): void Updates player stats after a game. getRankings(): Ranking[] Retrieves the current leaderboard rankings. findOpponent(playerID: string): string Finds a suitable opponent based on skill level.
Methods: updateProfile(playerID: string, stats: Stats): void Updates the player's profile with new stats and game history. getHistory(playerID: string): GameHistory[] Retrieves the player's game history.	

Appendix 3: Annotated Use Case Diagrams

Figure 3.1: Matchmaking Diagram

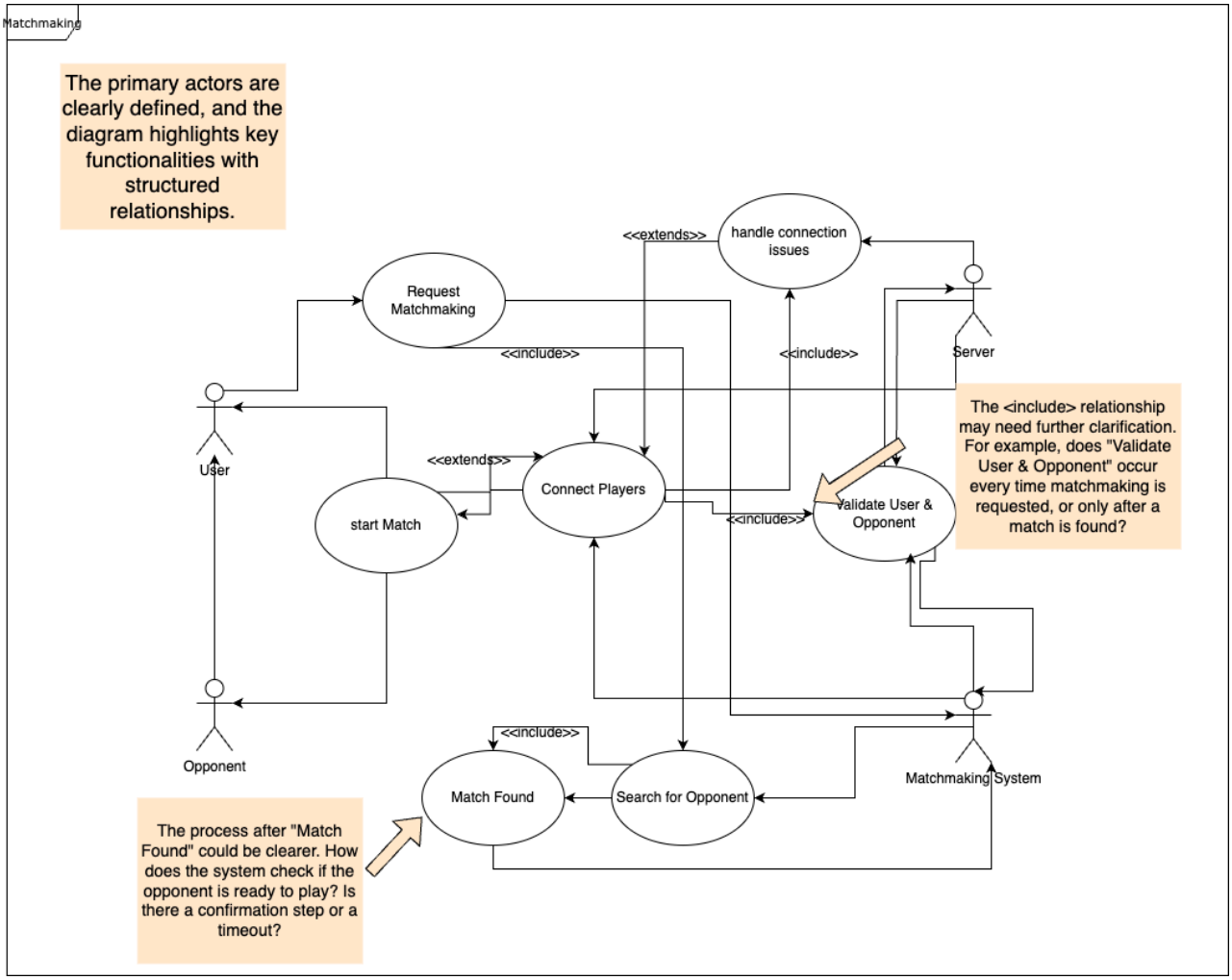


Figure 3.2: Go Diagram

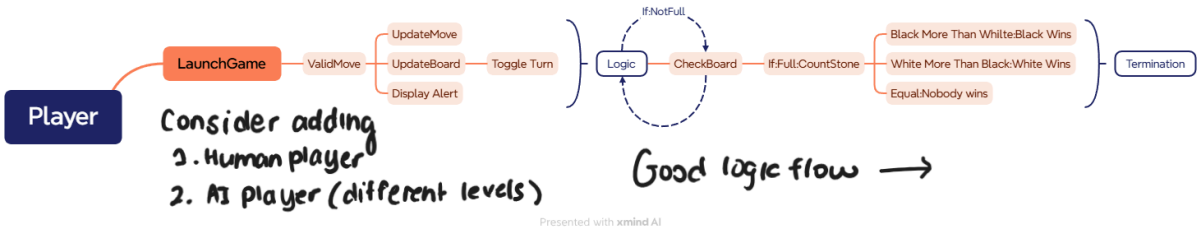


Figure 3.3: Query Diagram

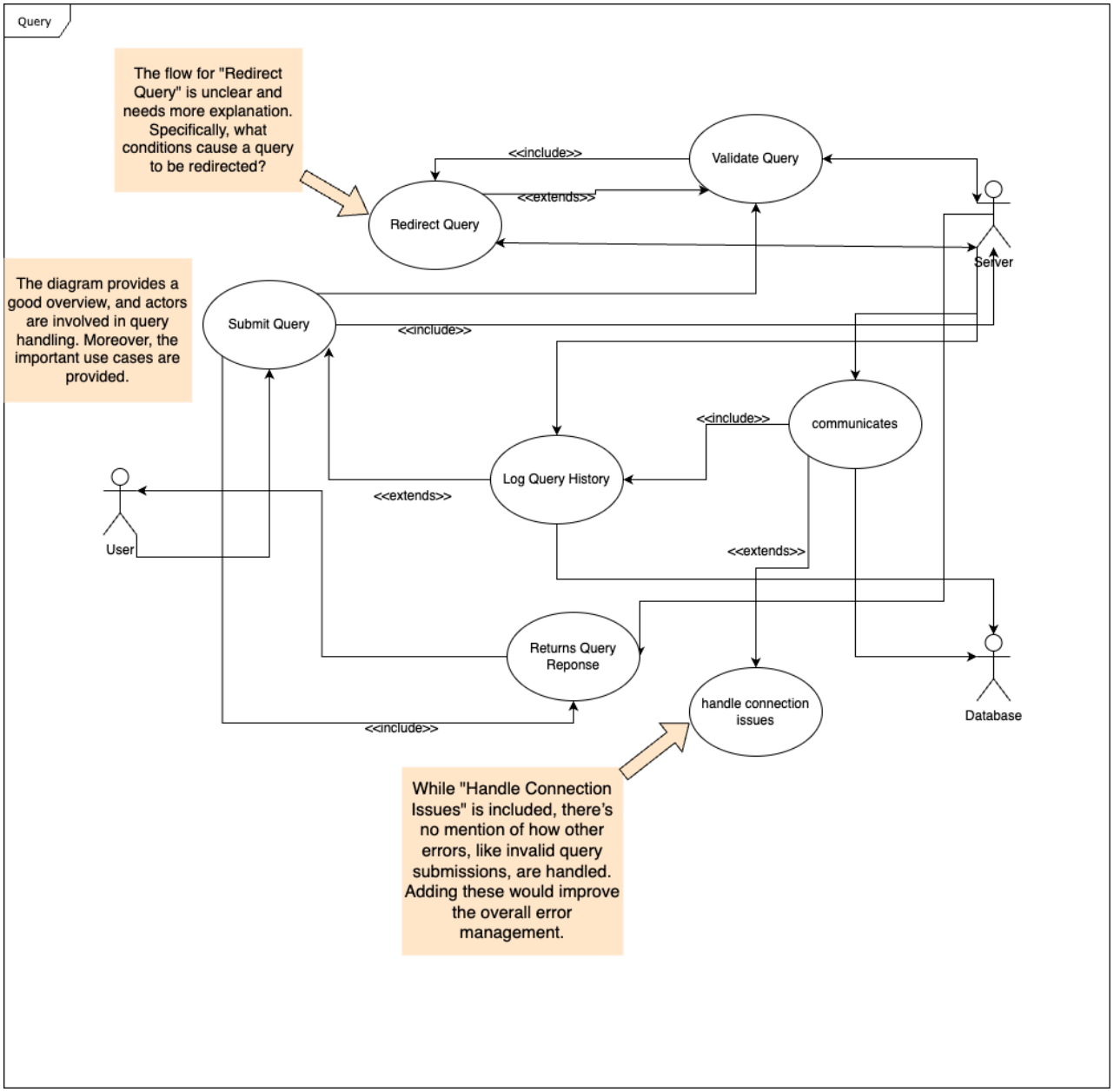


Figure 3.4: Synchronization Diagram

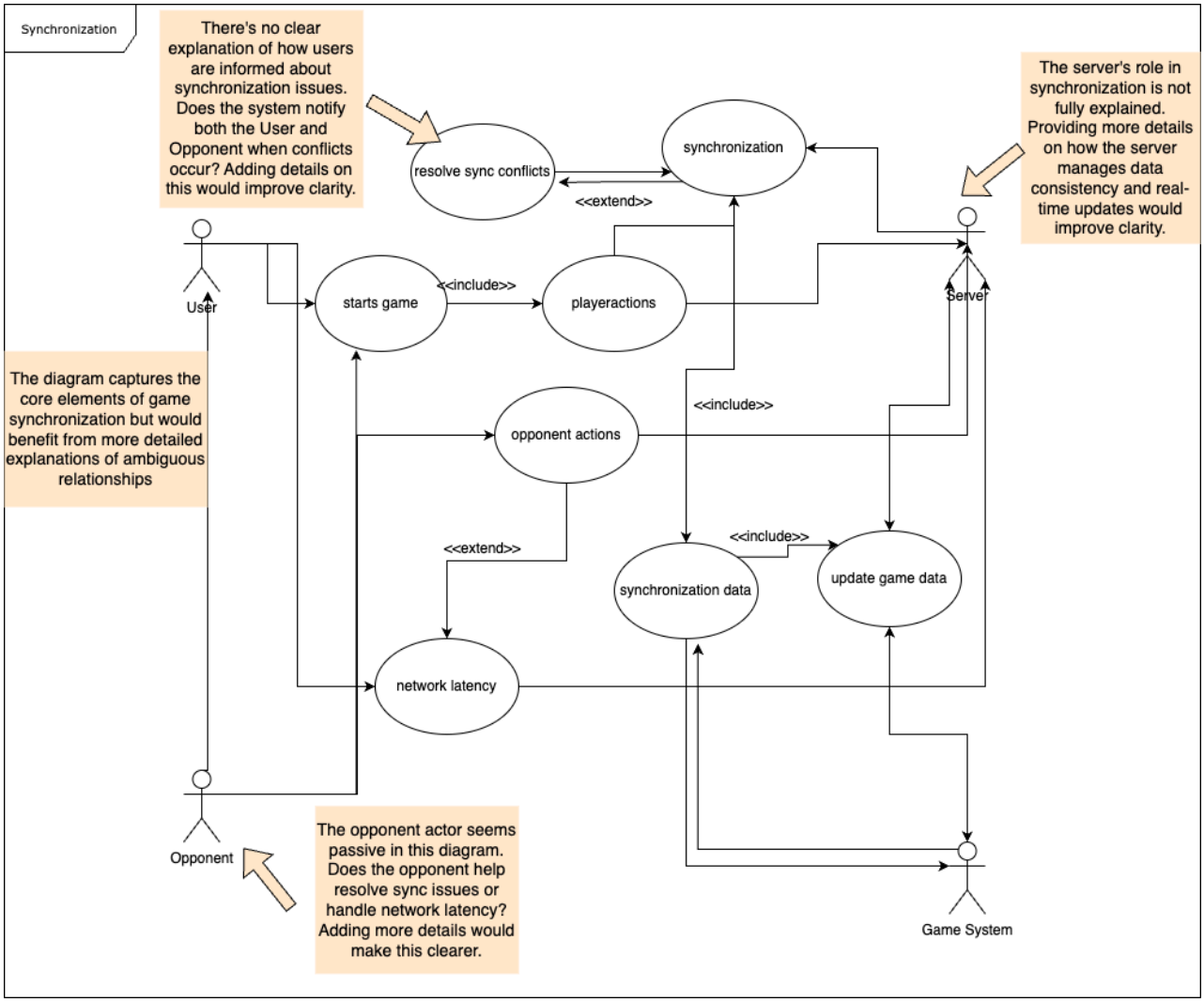


Figure 3.5: Gameplay Diagram

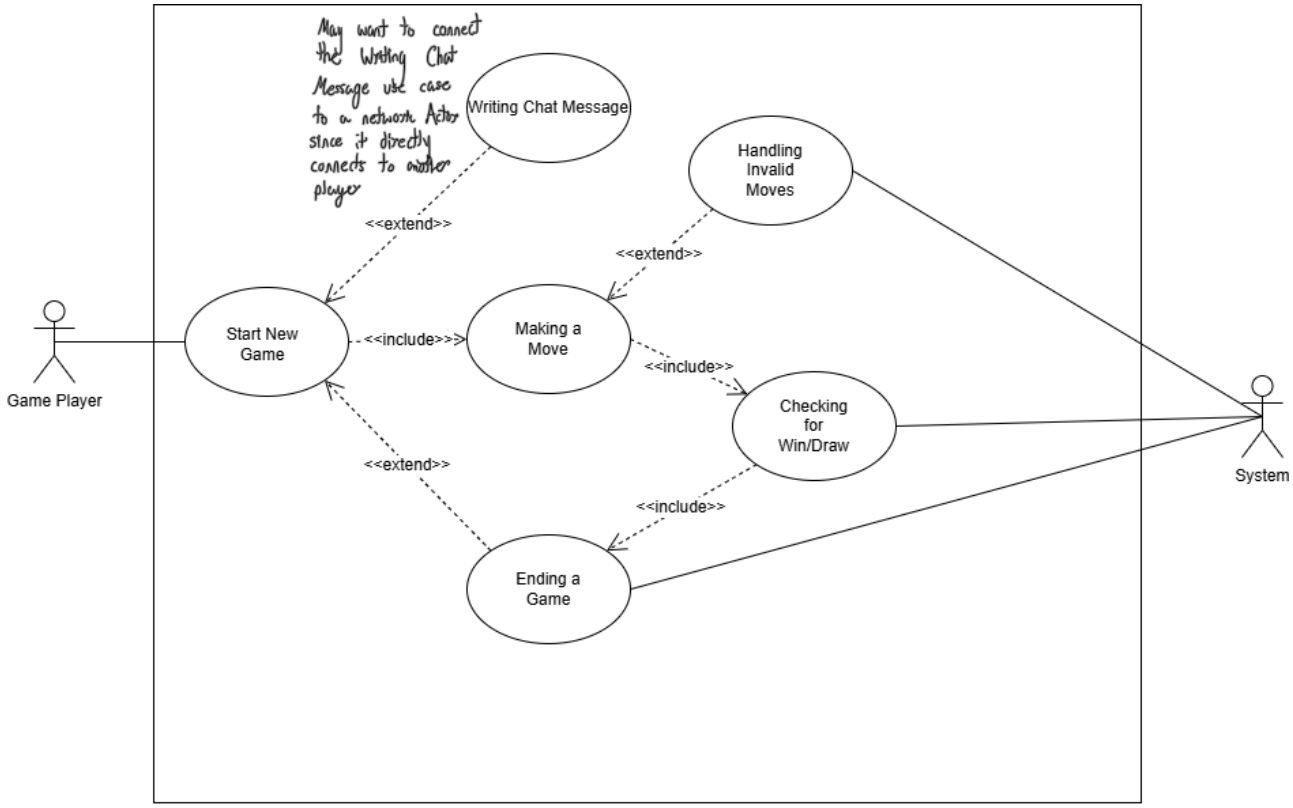


Figure 3.6: Authentication Diagram

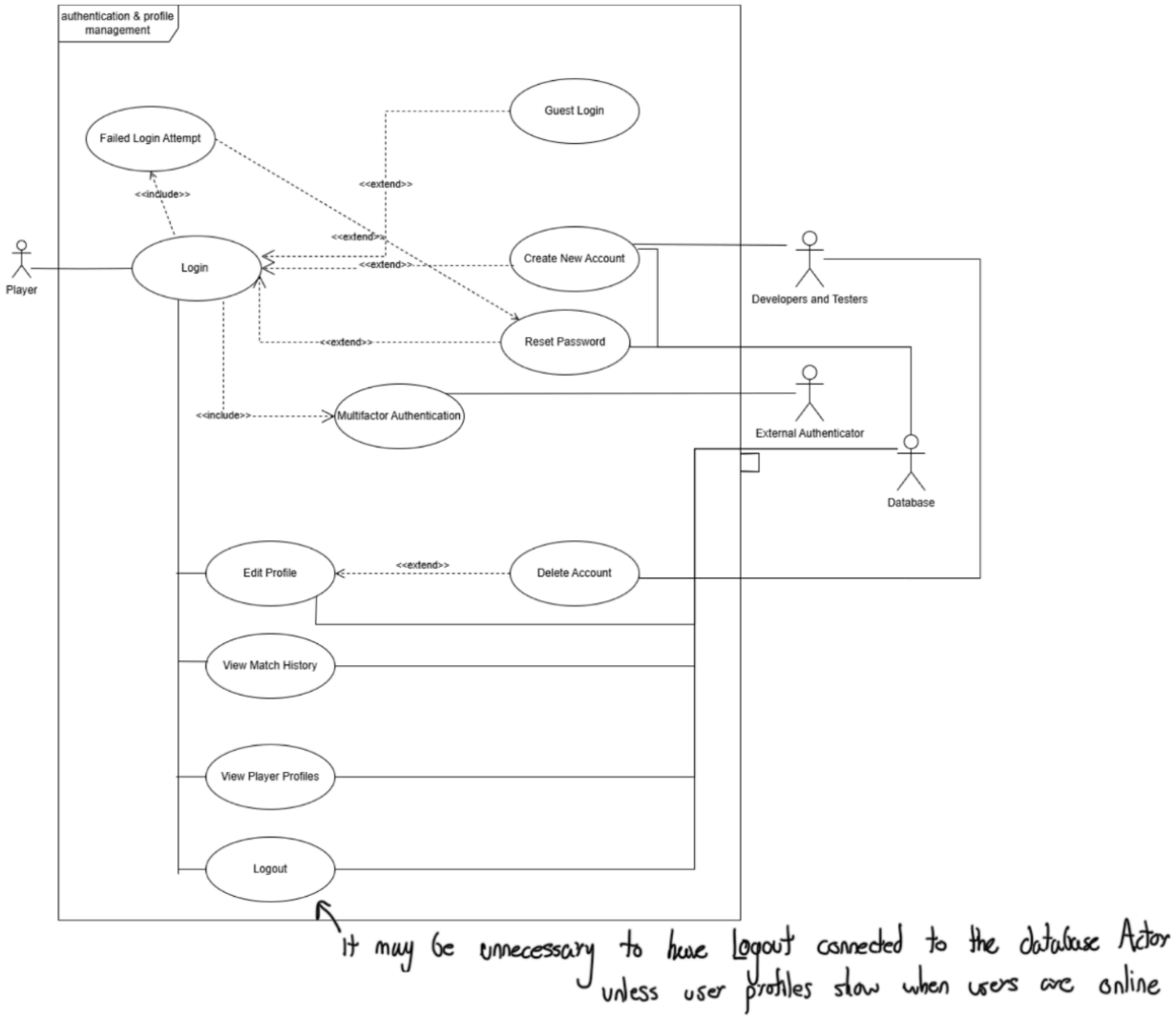
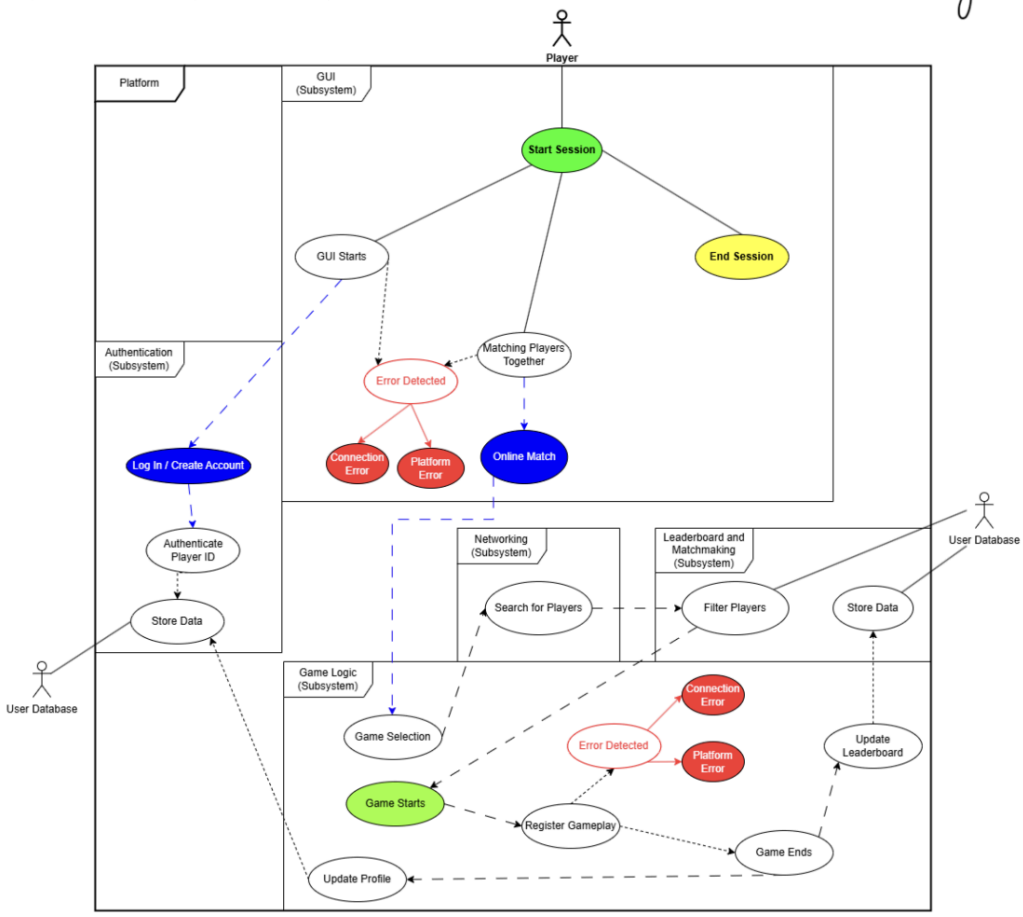


Figure 3.7: Integration Diagram

Clear and well done diagram



Association Legend

	Include	Description: The relationship includes the following use case. (i.e. When the user moves to 'Pay for Order', they'll need to select their 'Payment Method', and finally 'Confirm Payment')
	Associated	Description: There is an association in the relationship that causes an actor or a use case to interact with another.
	Extend	Description: The relationship implies that the case extends under certain conditions. In this case, when the user begins their session they'll immediately have the option to select their language of operation and/or call for assistance
	User Database	Description: This represents the project's external database that stores the players statistics (Wins, losses, points, and ranking). This information can be accessed mainly by the Authentication and Leaderboard and Matchmaking teams