Faculty of Engineering
Department of Electronics and Electrical Communications
ELC2080 Project

Advanced Tic-Tac-Toe

Software Design Specification Document

Name	ID
Ziad Mostafa	9231055
Ziad Emad	9230401
Ziad Abdel Hafeez	9230399
Sandra Atef	9230428
Robir Tamer	9230378

Under the supervision of:

Dr. Omar Ahmed Nasr

Table of Contents

1 Document Release History ... 4

2 Authors, Reviewers and Stakeholders' ... 4

Client Stakeholders Register ... 4

3 Introduction ... 5

3.1 Purpose ... 5

3.2 Scope ... 5

3.3 Document Conventions ... 5

3.4 Intended Audience ... 5

4 System Overview ... 6

4.1 System Context ... 6

4.2 System Architecture ... 6

5 Architectural Design ... 6

5.1 High-Level Architecture ... 6

5.2 Modules and Components ... 6

5.2.1 Game Logic Module ... 6

5.2.2 AI Opponent Module ... 6

5.2.3 Graphical User Interface (GUI) Module ... 7

5.2.4 User Authentication and Management Module ... 7

5.2.5 Personalized Game History Module ... 7

5.2.6 Database Module ... 7

5.2.7 Testing and Quality Assurance Module ... 7

5.2.8 CI/CD Integration Module ... 8

6 Detailed Design ... 8

6.1 Class Diagrams ... 8

6.2 Sequence Diagrams ... 9

6.2.1 User Login Sequence ... 9

6.2.2 Player Makes a Move Sequence ... 9

6.2.3 Saving Game History Sequence ... 9

6.3 Data Structures ... 10

6.4 Algorithms ... 10

6.5 Security Practices ... 10

6.6 Game Logic Implementation ... 10

6.7 AI Opponent Implementation ... 11

7 User Interface Design ... 11

7.1 Screen Layouts ... 11

7.1.1 Mode Selection Screen (m_modeSelectionScreen) ... 11

7.1.2 Player Authentication Screen (m_playerAuthScreen) ... 11

7.1.3 Game Screen (m_gameScreen) ... 12

7.1.4 Statistics View (m_statisticsView) ... 12

7.2 User Interaction Flows 12
8 Data Design 12
8.1 Data Entities 12
8.2 Data Storage 13
8.3 Data Persistence 13
9 Testing Strategy 13
9.1 Unit Testing 13
9.2 Integration Testing 14
9.3 Test Automation 14
10 Deployment Strategy 14
10.1 CI/CD Pipeline 14
10.2 Version Control 14
11 Future Enhancements 14

1 Document Release History

#	Version No.	Release Date	Prepared By	Reviewed By	Preparation Comments
1	V1.0	17/6/2024	Robir Tamer	Ziad Abdel Hafeez	Initial Publication

2 Authors, Reviewers and Stakeholders’

Client Stakeholders Register

#	Name	Role in the project	Influence	Contact	Email
1	Dr. Omar Ahmed Nasr	Program Manager	SPOC	01143941832	omaranasr.phone@gmail.com

3 Introduction

3.1 Purpose

The purpose of this Software Design Specification (SDS) is to provide a detailed description of the software architecture and design for the Advanced Tic-Tac-Toe Game.

This document outlines the system's structure, components, interfaces, and data, serving as a blueprint for implementation and a reference for future development and maintenance. It aims to ensure a well-structured implementation by detailing the design decisions, architectural patterns, and technical specifications of the game.

3.2 Scope

This SDS covers the design aspects of an advanced Tic-Tac-Toe game that incorporates user authentication, personalized game history, and an intelligent AI opponent. The game supports both player-vs-player and player-vs-AI modes, allowing users to log in, track their game history, and analyze past games. The design emphasizes secure user management, rigorous testing, and professional version control workflows.

3.3 Document Conventions

- **Terms and Definitions:** Key terms used throughout this document are defined to ensure clarity and consistency.
- **Acronyms:** Standard acronyms are used for brevity (e.g., GUI for Graphical User Interface, AI for Artificial Intelligence, CI/CD for Continuous Integration/Continuous Deployment).
- **Formatting:** Headings, bullet points, and code snippets are used to enhance• readability.

3.4 Intended Audience

This document is intended for software developers, quality assurance engineers, project managers, and other stakeholders involved in the development, testing, and maintenance of the Advanced Tic-Tac-Toe Game.

4 System Overview

4.1 System Context

The Advanced Tic-Tac-Toe Game is a software application designed to provide an interactive and engaging experience for players. It operates as a standalone application, interacting with users through a graphical interface and managing data locally or through a lightweight database. The system's primary interactions are with the end-user, who can play the game, manage their profile, and review game history.

4.2 System Architecture

The system architecture of the Advanced Tic-Tac-Toe Game is designed to be modular and scalable, comprising several interconnected components. These components work together to provide the core functionalities of the game, user management, and AI capabilities. The architecture follows a layered approach, separating concerns to enhance maintainability and flexibility.

5 Architectural Design

5.1 High-Level Architecture

The high-level architecture of the Advanced Tic-Tac-Toe Game can be conceptualized into several main layers:

- **Presentation Layer (GUI):** Responsible for user interaction, displaying the game board, and managing user input. This layer includes all graphical elements and forms.
- **Application Layer (Game Logic & AI):** Contains the core game rules, state management, and the AI opponent's decision-making logic. This layer processes user actions and determines game outcomes.
- **Data Management Layer:** Handles the storage and retrieval of user data, game histories, and other persistent information. This layer abstracts the underlying database or file storage mechanism.
- **Utility/Service Layer:** Provides common services such as authentication, data hashing, and other supporting functionalities used across different layers.

This layered approach ensures that changes in one layer have minimal impact on others, promoting a robust and maintainable system.

5.2 Modules and Components

The system is composed of several key modules, each responsible for a specific set of functionalities:

5.2.1 Game Logic Module

This module encapsulates the fundamental rules and mechanics of the Tic-Tac-Toe game. It manages the game board state, validates moves, and determines win or tie conditions. Key components include:

- **GameBoard:** Manages the 3x3 grid, storing the current state of each cell (empty, 'X', or 'O').
- **GameRules:** Implements the logic to check for winning combinations (rows, columns, diagonals) and tie conditions.
- **MoveValidator:** Ensures that player moves are valid (e.g., selecting an empty cell within the board boundaries).

5.2.2 AI Opponent Module

This module provides the intelligent opponent for single-player mode. It employs strategic algorithms to make optimal moves, offering a challenging experience to the player. The primary algorithm used is the Minimax algorithm with Alpha-Beta Pruning.

- **Minimax Algorithm:** A recursive algorithm used for decision-making in game theory. It explores all possible moves to a certain depth, evaluating the board state to determine the best possible move for the AI.
- **Alpha-Beta Pruning:** An optimization technique for the Minimax algorithm that reduces the number of nodes evaluated in the search tree, significantly improving performance without affecting the outcome.

5.2.3 Graphical User Interface (GUI) Module

This module is responsible for presenting the game to the user and handling all user interactions. It is built using the Qt framework, providing a rich and interactive user experience. Key GUI components include:

- **Game Screen:** Displays the Tic-Tac-Toe board, allowing players to make moves by clicking on cells. It also shows game status (e.g., current player, win/loss/tie messages).
- **Login/Registration Forms:** Enables users to create new accounts and log in to existing ones securely.
- **Game History View:** Allows users to browse and replay their past games, showing• details such as opponent, outcome, and date.
- **Main Window:** Serves as the central hub, providing navigation to different sections of the application (e.g., single-player, multiplayer, history, settings).

5.2.4 User Authentication and Management Module

This module handles all aspects of user accounts, including registration, login, and profile management. Security is a paramount concern, with measures taken to protect user credentials.

- **User Registration:** Allows new users to create accounts by providing a username and password.
- **User Login:** Authenticates existing users against stored credentials.
- **Password Hashing:** Uses secure hashing algorithms (e.g., SHA-256) to store passwords, preventing plain-text storage and enhancing security.
- **Session Management:** Manages user sessions to keep users logged in and maintain their state across different interactions.

5.2.5 Personalized Game History Module

This module is responsible for recording, storing, and retrieving game sessions for each user. Players can review their past performance and analyze game strategies.

- **Game Storage:** Stores details of each completed game, including moves made, outcome, date, and opponent.
- **History Retrieval:** Allows users to query and view their game history based on various criteria.
- **Game Replay:** Provides functionality to replay past games, step-by-step, to analyze moves and strategies.

5.2.6 Database Module

This module provides an abstraction layer for data persistence, managing user data and game histories. It is designed to be flexible, supporting either SQLite or a custom file storage solution.

- **Database Interface:** Defines a consistent API for interacting with the underlying data storage.
- **User Data Management:** Stores and retrieves user profiles, including usernames and hashed passwords.
- **Game Record Management:** Stores and retrieves detailed records of each game played.

5.2.7 Testing and Quality Assurance Module

This module encompasses the strategies and tools used to ensure the quality, reliability, and performance of the software. It includes both unit and integration testing.

- **Unit Testing (Google Test):** Develops and executes tests for individual components and functions to verify their correctness in isolation.
- **Integration Testing:** Tests the interactions between different modules and components to ensure they work together seamlessly as a complete system.

5.2.8 CI/CD Integration Module

This module integrates the development workflow with Continuous Integration and Continuous Deployment practices using GitHub Actions. This automates the build, test, and deployment processes.

- **Automated Builds:** Configures workflows to automatically build the project upon code changes.
- **Automated Testing:** Runs unit and integration tests automatically to catch regressions early.
- **Deployment Automation:** Sets up scripts for automated deployment of the application.

6 Detailed Design

6.1 Class Diagrams

- Placeholder for Class Diagrams - These will be generated based on code analysis and typical Tic-Tac-Toe game structures. The sample SDS provided UML diagrams, which will be referenced for structure.
- While a visual UML Class Diagram would typically be included here, this section provides a textual description of the key classes and their relationships within the Advanced Tic-Tac-Toe Game. The design follows an object-oriented approach, encapsulating functionalities within distinct classes to promote modularity and maintainability.
    - MainWindow Class:
        - ❖ Purpose: The central class managing the application's GUI, user interactions, and overall flow. It acts as the main orchestrator, connecting various modules.
        - ❖ Relationships:
            - ➢ Composition: Owns instances of Authentication, GameLogic,GameHistory , AIOpponent, and Database classes. This indicates that Main Window is responsible for their creation and lifetime.
            - ➢ Aggregation: Interacts with User objects (e.g., m_currentUser in Authentication) to display user-specific information and manage game sessions.
            - ➢ Dependency: Depends on all other core classes for its functionality, as it coordinates their operations
- Authentication Class:
    - Purpose: Manages user registration, login, and authentication processes, including secure password handling.
    - Relationships:
        - ❖ Composition: Manages a QVector<User*> of registered users.
        - ❖ Dependency: Depends on User for user data representation and QCryptographicHash for password hashing.
- User Class
    - Purpose: Represents a single user profile, storing username, hashed password, and game statistics (wins, losses, draws, streaks) and history.
    - Relationships:
        - ❖ Composition: Contains a QVector<GameRecord> to store individual game history entries.
        - ❖ Dependency: Used by Authentication and Database classes.
- GameLogic Class:
    - Purpose: Implements the core rules and mechanics of the Tic-Tac-Toe game, managing the board state, player turns, and win/draw conditions.
    - Relationships:
        - ❖ Dependency: Used by MainWindow to control game flow and by AIOpponent to interact with the game board.
- AIOpponent Class:
    - Purpose: Implements the intelligent AI opponent using the Minimax algorithm with Alpha-Beta Pruning.
    - Relationships:
        - ❖ Dependency: Depends on GameLogic to access and manipulate the game board state and determine difficulty.
- Database Class:
    - Purpose: Handles data persistence for user profiles and game history, using a JSON file-based storage mechanism.
    - Relationships:
        - ❖ Dependency: Interacts with User and GameRecord objects for data storage and retrieval.
- GameHistory Class:

  - Purpose: (Based on the presence in MainWindow 's includes this class likely manages the overall game history functionality, possibly aggregating records from multiple users or providing specific history-related queries. Its detailed implementation was not provided in the src folder, but its inclusion suggests a dedicated role.)
  - Relationships: Likely depends on User and GameRecord classes

6.2 Sequence Diagrams

- Placeholder for Sequence Diagrams - These will illustrate the flow of interactions between different components for key use cases like user login, making a move, and saving game history. The sample SDS provided UML diagrams, which will be referenced for structure.
- While visual UML Sequence Diagrams would typically be included here, this section provides textual descriptions of key interaction flows within the Advanced Tic-Tac-Toe Game. These descriptions illustrate the sequence of messages exchanged between objects to accomplish specific use cases.

6.2.1 User Login Sequence

This sequence describes the steps involved when a user attempts to log in to the application.

1. **User** interacts with **MainWindow** (via m_loginBtn click).
2. **MainWindow** calls onLoginClicked().
3. **MainWindow** retrieves username and password from m_usernameInput and m_passwordInput .
4. **MainWindow** calls authentication.login(username, password).
5. **Authentication** checks credentials against its m_users collection.
6. **Authentication** calls user.checkPassword(password) for the matching User object.
7. **User** verifies the provided password against its stored hashed password.
8. **Authentication** returns true (success) or false (failure) to MainWindow .
9. **MainWindow** updates m_loginStatus label based on the result.
10. If successful, **MainWindow** transitions to the appropriate screen (e.g., game mode selection).

6.2.2 Player Makes a Move Sequence

This sequence describes the steps involved when a player clicks on a cell on the game board.

1. **User** interacts with **MainWindow** (via m_cells QPushButton click).
2. **MainWindow** calls onCellClicked().
3. **MainWindow** determines the index of the clicked cell.
4. **MainWindow** calls gameLogic.makeMove(index).
5. **GameLogic** updates its m_board with the current player's mark.
6. **GameLogic** checks for win or draw conditions (checkWinner()).
7. **GameLogic** emits boardChanged() signal.
8. **MainWindow** receives boardChanged() signal and calls onBoardChanged().
9. **MainWindow** updates the visual state of the m_cells on the GUI
10. If the game is not over, **GameLogic** switches m_currentPlayer and emits playerChanged() signal.
11. **MainWindow** receives playerChanged() signal and calls onPlayerChanged().
12. **MainWindow** updates m_statusMessage to reflect the new current player.
13. If the game is in AI mode and it's AI's turn, **MainWindow** aiOpponent.makeMove().
14. **AIOpponent** calculates the best move using minimax and evaluateBoard.
15. **AIOpponent** calls gameLogic.makeMove(bestMove).
16. The sequence from step 5 repeats for the AI's move.

6.2.3 Saving Game History Sequence

This sequence describes how game results are recorded and stored.

1. **GameLogic** determines game over (win, loss, or draw) and emits gameOver(result) signal.
2. **MainWindow** receives gameOver(result) signal and calls onGameOver(result).
3. **MainWindow** calls addGameToHistory(result).
4. **MainWindow** prepares game record data (player names, result, date).
5. **MainWindow** calls database.saveGame(playerX, playerO, result) (though the provided database. shows this is a placeholder).
6. **MainWindow** updates the current user's game statistics and history (e.g., user.addGameWithDate()).
7. **MainWindow** calls database.saveUsers(m_auth->getUsers()) to persist updated user data.
8. **Database** serializes user data and game history to tictactoe.json **.**

6.3 Data Structures

- **Game Board Representation:** A 2D array (e.g., char gameData[3][3]) is used to represent the Tic-Tac-Toe board, storing the state of each cell.
- **Minimax Tree:** For the AI opponent, a tree data structure is implicitly or explicitly used to explore possible game states and evaluate moves. Each node in the tree represents a game state, and edges represent possible moves.
- **User Data Storage:** For user authentication and game history, a lightweight database like SQLite or a custom file storage mechanism is used. This would involve data structures suitable for efficient storage and retrieval of user profiles and game records (e.g., hash tables for user lookup, linked lists or arrays for game history).

6.4 Algorithms

- **Minimax Algorithm with Alpha-Beta Pruning:** As detailed in the AI Opponent Module, this algorithm is central to the AI's decision-making process. It recursively evaluates game states, assigning scores to determine the optimal move.
- **Hashing Algorithm:** For secure password storage, a cryptographic hashing algorithm (e.g., SHA-256) is used to transform plain-text passwords into fixed-size hash values. This ensures that passwords are not stored in a readable format.• **Game State Evaluation:** Functions to evaluate the current state of the game board (e.g., checkWinner() , checkDraw())are crucial for determining game outcomes and informing the AI's decisions.

6.5 Security Practices

• **Password Hashing with Salt:** The Authentication class implements secure password storage by hashing passwords using SHA-256. A unique, randomly generated salt is prepended to the password before hashing. This prevents rainbow table attacks and makes brute-force attacks more difficult, even if the database is compromised. The hashPassword function generates a salt and combines it with the password before hashing, and verifyPassword uses the stored salt to verify the provided password [1].

• **Secure User Authentication:** The login and registerUser methods in the Authentication class handle user authentication and registration. They validate input, check for existing usernames, and use the hashPassword and verifyPassword functions to ensure secure credential handling [1].

6.6 Game Logic Implementation

The GameLogic module is central to the Tic-Tac-Toe game, managing the game state, player turns, and determining game outcomes. It provides a clear interface for making moves, resetting the board, and querying the current state of the game.

- **Board Representation:** The game board is represented by a QVector<Player> m_board of size 9, where each element corresponds to a cell on the 3x3 grid. The Player enum (None , X , O) indicates the state of each cell [2].
- **Making a Move:** The makeMove(int index) function allows a player to place their mark on the board. It validates the move (ensuring the cell is empty and the game is active), updates the board, and then checks for a winner or a draw. After a valid move, it switches the m_currentPlayer [2].
- **Win and Draw Conditions:** The checkWinner() function iterates through predefined winPatterns (rows, columns, and diagonals) to determine if the current player has achieved three in a row. If no winner is found and the board is full, the game is declared a draw [2].

6.7 AI Opponent Implementation

The AIOpponent module is responsible for the intelligent decision-making of the computer opponent in single-player mode. It leverages the Minimax algorithm with Alpha-Beta Pruning to determine the optimal move based on the current game state and selected difficulty level.

- **Minimax Algorithm with Alpha-Beta Pruning:** The core of the AI is the minimax function, which recursively explores possible game states. It evaluates board configurations, assigning scores to maximize the AI's chances of winning and minimize the player's chances. Alpha-beta pruning is implemented to optimize the search by eliminating branches that cannot possibly influence the final decision [3].
- **Board Evaluation:** The evaluateBoard function assigns a numerical score to a given board state. This score guides the Minimax algorithm, favoring positions that lead to a win for the AI and penalizing those that lead to a win for the player. It considers factors like two-in-a-row opportunities and control of the center position [3].
- **AI Difficulty Levels:** The AI's behavior is adjusted based on the AIDifficulty setting from the GameLogic module. This is primarily achieved by varying the maxDepth parameter in the minimax function and introducing strategic variations for easier difficulties [3].

  ◦ **Easy:** maxDepth = 1. The AI performs a very shallow search, often making random moves or immediate winning moves.

  ◦ **Medium:** maxDepth = 2. The AI performs a slightly deeper search and will block immediate player wins.

  ◦ **Hard:** maxDepth = 3. The AI performs a deeper search, making more strategic moves.

  ◦ **Expert:** maxDepth = 9. The AI performs a full search of the game tree, aiming for the optimal move (assuming a perfect game from both sides).

  Additionally, for Easy and Medium difficulties, there's a chance for the AI to make a random move, simulating less-than-perfect play [3].

  - **Move Ordering:** The findBestMove function incorporates moveOrder (center, corners, edges) to prioritize certain moves, which can improve the efficiency of alpha-beta pruning by finding good moves earlier in the search [3].

7 User Interface Design

The Graphical User Interface (GUI) of the Advanced Tic-Tac-Toe Game is implemented using the Qt framework, providing an interactive and visually appealing experience. The MainWindow class orchestrates the various screens and manages user interactions, connecting UI elements to the underlying game logic and authentication systems [4].

7.1 Screen Layouts

Placeholder for Screen Layouts - Descriptions of the main screens: Login, Registration, Main Menu, Game Screen, History View, and Replay Screen. This section will detail the arrangement of UI elements on each screen.

7.1.1 Mode Selection Screen (m_modeSelectionScreen)

This is the initial screen presented to the user, offering choices for game mode and user authentication. It includes:

- Game Mode Buttons: m_vsAIBtn (Player vs. AI) and m_vsPlayerBtn (Player vs. Player) to select the desired game type.
- Login/Registration Inputs: m_usernameInput, m_passwordInput for single-player login, and m_loginBtn, m_registerBtn to initiate authentication processes.
- Login Status Label: m_loginStatus to display messages related to login attempts.

7.1.2 Player Authentication Screen (m_playerAuthScreen)

Dedicated to multiplayer game setup, this screen allows two players to authenticate before starting a game. It features:

- Player 1 Credentials: m_player1UsernameInput, m_player1PasswordInput.

- Player 2 Credentials: m_player2UsernameInput, m_player2PasswordInput.
- Action Buttons: m_startPvpGameBtn to start the game and m_backToModeBtn to return to the mode selection screen.

7.1.3 Game Screen (m_gameScreen)

This is the main gameplay interface where users interact with the Tic-Tac-Toe board. Key elements include:

- Game Board: m_boardLayout (a QGridLayout) populated with QVector<QPushButton*> m_cells representing the 3x3 grid. Each button corresponds to a cell on the board.
- Game Status: m_statusMessage label to display current player, win/loss/draw messages.
- Game Control Buttons: m_newGameBtn (start a new game), m_saveGameBtn (save current game state), and m_exitGameBtn (exit the game).
- AI Difficulty Selection: m_difficultyContainer with m_easyBtn, m_mediumBtn , m_hardBtn , m_expertBtn to set the AI difficulty in single-player mode.
- Player Information: m_player1Name, m_player2Name labels, and m_player1Box, m_player2Box widgets to display current player details.
- Loading Overlay: m_loadingOverlay and m_loadingSpinner to indicate processing during AI turns or other operations.

7.1.4 Statistics View (m_statisticsView)

This screen provides detailed statistics and game history for the logged-in user. It includes:

- Summary Statistics: Labels such as m_statTotalGames, m_statWins, m_statLosses, m_statDraws, m_statVsAI, m_statVsPlayers, m_statWinRate, m_statBestStreak to display aggregated game data.
- Leaderboard: m_leaderboardList (a QListWidget) to show top players, potentially using LeaderboardItemDelegate for custom rendering.
- Full History: m_fullHistoryList (a QListWidget) to display a detailed list of past games, potentially using HistoryItemDelegate .
- Navigation Buttons: m_toggleStatsViewBtn to switch between game and statistics views, and m_backToG

7.2 User Interaction Flows

- Placeholder for User Interaction Flows - Descriptions of how users navigate through the application and interact with its features, such as the flow from launching the application to playing a game, or from logging in to viewing game history.
- The MainWindow class manages various user interaction flows through its private slots, which are connected to UI element signals. Key interaction flows include [4]:
    - Game Play: Users click on m_cells (buttons on the game board) to make moves. The onCellClicked() slot handles these events, triggering game logic updates (m_gameLogic->makeMove()) and UI refreshes (onBoardChanged()).
    - Game Mode Selection: Clicking m_vsAIBtn or m_vsPlayerBtn transitions the user to the appropriate game setup or authentication screen.
    - User Authentication: onLoginClicked() and onRegisterClicked() handle user input from QLineEdit fields, interacting with the Authentication module to verify credentials or create new accounts.
    - Game Over and Reset: The onGameOver() slot is triggered when a game concludes (win, loss, or draw), updating the status message and potentially highlighting winning cells. onNewGameClicked() resets the board and game state.
    - Statistics and History Navigation: onToggleStatsViewClicked() and onBackToGameClicked() manage the visibility of the statistics view, allowing users to review their performance and history.

8 Data Design

8.1 Data Entities

- **User:**
    - username (string, unique identifier)
    - hashed_password (string)

    - salt (string, if applicable for hashing)
- **Game Session:**
    - game_id (unique identifier)
    - player1_username (string)
    - player2_username (string, or 'AI' for single-player)
    - outcome (string: 'Win', 'Loss', 'Tie')
    - date (timestamp)
    - moves (serialized list of moves)

8.2 Data Storage

The system will utilize either SQLite for structured data storage or a custom file-based storage solution. The choice will depend on the specific requirements for portability,performance, and complexity of data management. If a custom file storage is used, a suitable data structure (e.g., JSON, CSV) will be chosen for efficient serialization and deserialization of data.

8.3 Data Persistence

The Database module is responsible for managing the persistence of user data and game history. It uses a JSON file (tictactoe.json) located in the application's data directory for storage. This approach provides a lightweight and flexible solution for data management without requiring a full-fledged database server

- User Data Storage: The saveUsers function serializes a QVector of User objects into a JSON array, where each User object is represented as a JSON object containing their username, game statistics (total games, wins, losses, draws, vs AI, vs Players, win rate, best streak), and game history. The loadUsers function performs the reverse operation, deserializing the JSON data back into User objects.
- Game History Storage: Each User object contains a QVector<GameRecord> to store their game history. Each GameRecord includes the date and result of the game. This history is serialized and deserialized along with the user's other statistics.
- Leaderboard Calculation: The getLeaderboard function calculates a composite score for each player based on their wins, total games, win rate, and best streak. This allows for a dynamic ranking of players. The calculatePlayerScore helper function defines the weighting for these factors.

9 Testing Strategy

9.1 Unit Testing

Unit tests will be developed using the Google Test framework to verify the correctness of individual functions, classes, and modules in isolation. This includes testing:

- Game logic (win conditions, tie conditions, valid moves)
- AI algorithm (Minimax logic, optimal move selection)
- Authentication functions (password hashing, login validation)
- Database operations (data storage, retrieval, updates)

9.2 Integration Testing

Integration tests will focus on verifying the interactions between different modules and components. This ensures that the system works cohesively as a whole. Examples include:

• User login flow (GUI interacting with authentication and database modules)• Game play flow (GUI interacting with game logic and AI modules)
• Saving and retrieving game history (Game Logic interacting with Database module)

9.3 Test Automation

Tests will be automated using GitHub Actions as part of the CI/CD pipeline. This ensures that all tests are run automatically upon code changes, providing immediate feedback on the health and stability of the codebase.

10 Deployment Strategy

10.1 CI/CD Pipeline

A Continuous Integration/Continuous Deployment (CI/CD) pipeline will be implemented using GitHub Actions. This pipeline will automate the following stages:

• **Build:** Compiling the C++ source code into an executable application.
• **Test:** Running all unit and integration tests to ensure code quality and functionality.
• **Deployment:** Packaging the application for distribution and potentially deploying it to a target environment (e.g., a release build for users).

10.2 Version Control

Git will be used as the version control system, with GitHub serving as the remote repository. This enables collaborative development, tracks all code changes, and facilitates branching and merging strategies for managing different development lines.

11 Future Enhancements

This section outlines potential future enhancements and features that could be added to the Advanced Tic-Tac-Toe Game to further improve its functionality, user experience, and scalability.

- **Online Multiplayer:** Implement network capabilities to allow players to compete against each other over the internet. This would involve a server-side component to manage game sessions, matchmaking, and real-time communication between clients.
- **More Sophisticated AI:** Explore advanced AI techniques beyond Minimax, such as Monte Carlo Tree Search (MCTS) or neural networks, to create an even more challenging and adaptive AI opponent.
- **Customizable Game Rules:** Allow users to define custom game rules, such as different board sizes (e.g., 4x4, 5x5), different winning conditions (e.g., four in a row), or special game modes.
- **Themed Boards and Pieces:** Provide options for players to customize the visual appearance of the game board and pieces (e.g., different colors, textures, or animated effects).
- **Achievements and Leaderboards:** Expand the current statistics and leaderboard functionality to include a more robust achievement system, rewarding players for specific milestones or accomplishments.
- **Replay Analysis Tools:** Enhance the game replay feature with analytical tools, allowing players to visualize game statistics, identify key turning points, and analyze their decision-making process.
- **Cross-Platform Compatibility:** Extend the application to support other operating systems (e.g., macOS, Linux) or even mobile platforms (iOS, Android) using Qt's cross-platform capabilities.
- **Improved Error Handling and Logging:** Implement more comprehensive error handling mechanisms and detailed logging to facilitate debugging and system monitoring in production environments.
- **Automated UI Testing:** Integrate automated UI testing frameworks to ensure the graphical user interface remains functional and consistent across updates.
- These enhancements would build upon the existing foundation, providing a richer and more engaging experience for players and extending the ga