1. Software requirements specifications

Introduction

Purpose of this Document:

The purpose of this Software Requirements Specification (SRS) document is to provide a comprehensive outline of the requirements for the development of the Advanced Tic Tac Toe Game. It serves as a detailed guide for all collaborators involved in the project, including developers, testers, and project managers. By documenting the functional and non-functional requirements, system behavior, and performance expectations, this document ensures clarity and alignment throughout the development lifecycle.

Scope of this Document:

This document outlines the scope of the Advanced Tic Tac Toe Game project, detailing its objectives, features, and the value it aims to deliver to users. It encompasses the implementation of user authentication, personalized game history tracking, and an intelligent AI opponent. The project also includes enhancements such as gameplay options with selectable difficulty levels (easy and hard) and integrated sound effects for an immersive gaming experience.

Overview:

The Advanced Tic Tac Toe Game project aims to elevate the traditional Tic Tac Toe experience with advanced features designed for enhanced user engagement. Developed in C++ with a focus on robust functionality and intuitive user interaction, this application integrates sophisticated elements such as user authentication, personalized game history tracking, and an intelligent AI opponent. Players can choose between two difficulty levels easy and hard tailoring their gameplay experience to their skill level and challenge preference.

In addition to strategic AI gameplay, the project introduces immersive sound effects that enhance the gaming atmosphere, providing auditory feedback during moves and game outcomes. These enhancements contribute to a dynamic and interactive user experience, fostering deeper engagement and enjoyment.

General Description

This advanced Tic Tac Toe game caters to players of all experience levels, offering a thrilling twist on the classic game. Built with C++, it boasts a user-friendly interface and exciting features that will keep you coming back for more.

Intuitive Gameplay: The familiar 3x3 grid and turn-based gameplay remain, making it easy to jump right in, regardless of your Tic Tac Toe experience.

AI Opponent with Adjustable Difficulty: Challenge yourself with a strategic AI opponent that adapts to your skill level. Choose from easy or hard difficulty to find the perfect match for your gaming mood.

Personalized Game History: Track your progress and relive past victories (or learn from defeats) with the personalized game history feature.

Immersive Sound Effects: Enhance your gameplay experience with captivating sound effects that bring every move and victory to life.

This project is perfect for anyone who:

- Enjoys the strategic challenge of Tic Tac Toe.
- Wants to test their skills against a formidable AI opponent.
- Appreciates a user-friendly interface and engaging sound effects.
- Desires to track their game history and analyze past matches.

Functional Requirements

Core Game Logic and Rules:

The core functionalities manage the game state, validate moves, detect win/tie conditions, and facilitate gameplay for both human players and the AI opponent.

Game flow:

1. User Registration

- **Initial Entry**:
  - When the user initially enters the game, the system checks if the user is registered.
  - If the user is not registered, the system prompts the user to register.
- **Registration Process**:
  - **Username**: The user must create a username that ends with @gmail.com.
  - **Password**: The user must create a password that is exactly 6 characters long.
  - **Database Update**: Upon clicking the register button, the username and password are stored in the users table of the database, which consists of:
    - username
    - password
    - history (initially empty)

2. Main Menu Options

After registration or login, the user is presented with the main menu options:

- **Play New Game**
- **View Game History**
- **Replay Past Game**

Since the account is new, there will be no history or past games initially.

3. Playing a New Game

- **Game Mode Selection**:
    - If the user chooses to play a new game, they are asked to select the game mode:
        - **Player vs Player**
        - **Player vs AI**
- **AI Difficulty Level**:
    - If the user chooses to play against AI, the system further prompts the user to choose the AI difficulty level:
        - **Easy**
        - **Hard**

4. Gameplay Logic

- **Player and AI Moves**:
    - The player always plays as X and the AI plays as O.
    - The game uses a variable called moves to track the sequence of moves. The moves variable has three components for each move:
        - Player (X or O)
        - Row position
        - Column position
- **Turn Sequence**:
    - The player makes the first move, which updates the moves variable:
        - Example: (X, row_position, column_position)
    - The AI then makes its move, updating the moves variable:
        - Example: (O, row_position, column_position)

    - This sequence continues with each move being appended to the moves variable.

- **Game Conclusion**:
    - The game continues until there is either a win or a tie.
    - After the game concludes, the sequence of moves is recorded in the game table in the database. The game table consists of:
        - ID (auto-incrementing identifier)
        - username
        - moves (the complete sequence of moves for the game)

5. Minimax Algorithm for AI

The AI opponents utilize the Minimax algorithm, enhanced by Alpha-Beta Pruning, to determine optimal moves. The Minimax algorithm evaluates all possible moves and their outcomes to select the move that maximizes the AI's chances of winning while minimizing the player's chances. Alpha-Beta Pruning optimizes this process by eliminating branches of the game tree that do not need to be explored, thus improving efficiency. The difficulty levels of the AI are distinguished by the depth to which these algorithms explore. In the "Easy" mode, the AI searches to a shallower depth, making it quicker but potentially less optimal in its decisions. In contrast, the "Hard" mode involves deeper exploration of the game tree, allowing the AI to make more strategic and challenging moves, thereby providing a tougher opponent. This variable depth in search significantly affects the AI's performance and difficulty level, ensuring a dynamic and scalable gaming experience. Here’s how the move is calculated.

- **AI Move Calculation**:
    - The AI uses the Minimax algorithm to determine the best move to make in order to win or at least tie the game.
    - For the **Easy** level, the depth of the Minimax algorithm is relatively small.
    - For the **Hard** level, the depth of the Minimax algorithm is high.

6. Database Tables

- **users**:
    - username (Primary Key)
    - password
    - history (initially empty)
- **game**:
    - ID (Primary Key, auto-incrementing)

- username
- moves (sequence of moves in the game)

Example Game Flow

1. **User Registration**:
    - User enters username: example@gmail.com
    - User enters password: secret
2. **Main Menu**:
    - User selects "Play New Game"
3. **Game Mode**:
    - User selects "Player vs AI"
    - User selects "Hard" difficulty
4. **Gameplay**:
    - Player (X) moves: (X, 1, 1)
    - AI (O) moves: (O, 0, 2)
    - Player (X) moves: (X, 2, 0)
    - ... (continues until game ends)
5. **Game Conclusion**:
    - Moves recorded in game table:
        - ID: 1
        - username: example@gmail.com
        - moves: X(1,1);O(0,2);X(2,0);...

Sound Effects:

The game can play sound effects for events like moves, wins, and ties, enhancing the user experience.

Overall System Function:

These functionalities work together to create a well-functioning Tic Tac Toe game. The system manages the game state, enforces game rules, validates moves, detects win/tie conditions, and facilitates intelligent (Hard difficulty) or random (Easy difficulty) AI gameplay.

Interface Requirements

Visual Elements:

- The game board will be visually represented as a 3x3 grid, with clear distinctions between empty cells and cells occupied by X or O symbols.
- Players will be indicated by X and O symbols placed in their respective turns.
- Additional UI elements include:
    - Menu bar for selecting between playing a new game (Player vs. Player, Player vs. AI) and difficulty level (Easy, Hard) for AI mode, view game history or replay.
    - Clear indication of win or tie conditions, such as a win message.

User Interaction Methods:

- Players will interact with the game by clicking on empty cells within the grid to place their move (X or O).
- Mouse clicks will be used to navigate and interact with menu options and buttons, ensuring intuitive gameplay.

Game Logic and AI Interface:

- Data Exchange:
    - The game logic module will communicate with the AI opponent module by providing:
        - The current state of the game board, detailing the positions of X, O, and empty cells.
    - The AI module will respond with its chosen move, specifying the cell index (position on the 3x3 grid) where it intends to place its symbol (X or O).

Optional External Interface Requirements:

- If the game includes features like persistent game history, the following additional interface requirements will apply:
    - Data Format:
        - Game history data, including dates, player names, winners, and difficulty levels, will be stored in a structured format.

- Communication Method:
    - The game logic will interact with a database (SQLite) using designated methods (e.g., API calls) to store and retrieve game history information secure.

Design Constraints

1. Algorithmic Constraints:
    - The AI opponent shall utilize the minimax algorithm with alpha-beta pruning for strategic decision-making.
    - Game state evaluation for AI shall be optimized to ensure moves are computed within acceptable time limits.
2. Platform and Environment Constraints:
    - The application shall be developed in C++ to ensure compatibility with existing libraries and frameworks used for GUI development.
    - GUI components and interactions shall be implemented using Qt framework to leverage its capabilities for graphical rendering and user interaction.
3. Security Requirements:
    - User authentication and password management shall adhere to industry-standard practices, including secure encryption algorithms for storing passwords.
    - Access to sensitive user data, such as game history and personal information, shall be protected through encryption during transmission and storage.

Non-functional attributes

- Security:

The application employs robust security measures to protect user data, including secure authentication mechanisms and encrypted data transmission.

- Portability:

Designed with platform independence in mind, the game supports deployment across multiple operating systems (e.g., Windows, macOS, Linux) without compromising functionality or performance.

- Reliability:

The software is engineered to deliver consistent and dependable performance under normal and peak load conditions, minimizing crashes and errors during gameplay.

- Usability:

User interfaces are intuitively designed to enhance usability, featuring clear navigation, informative feedback, and accessible controls that cater to both novice and experienced players.

- Scalability:

As player engagement grows, the game's architecture allows for seamless scalability, accommodating increased user traffic and data volume without degradation in performance.

- Performance Efficiency:

Optimized for efficiency, the application minimizes resource utilization (CPU, memory) while delivering responsive gameplay and quick response times to user interactions.

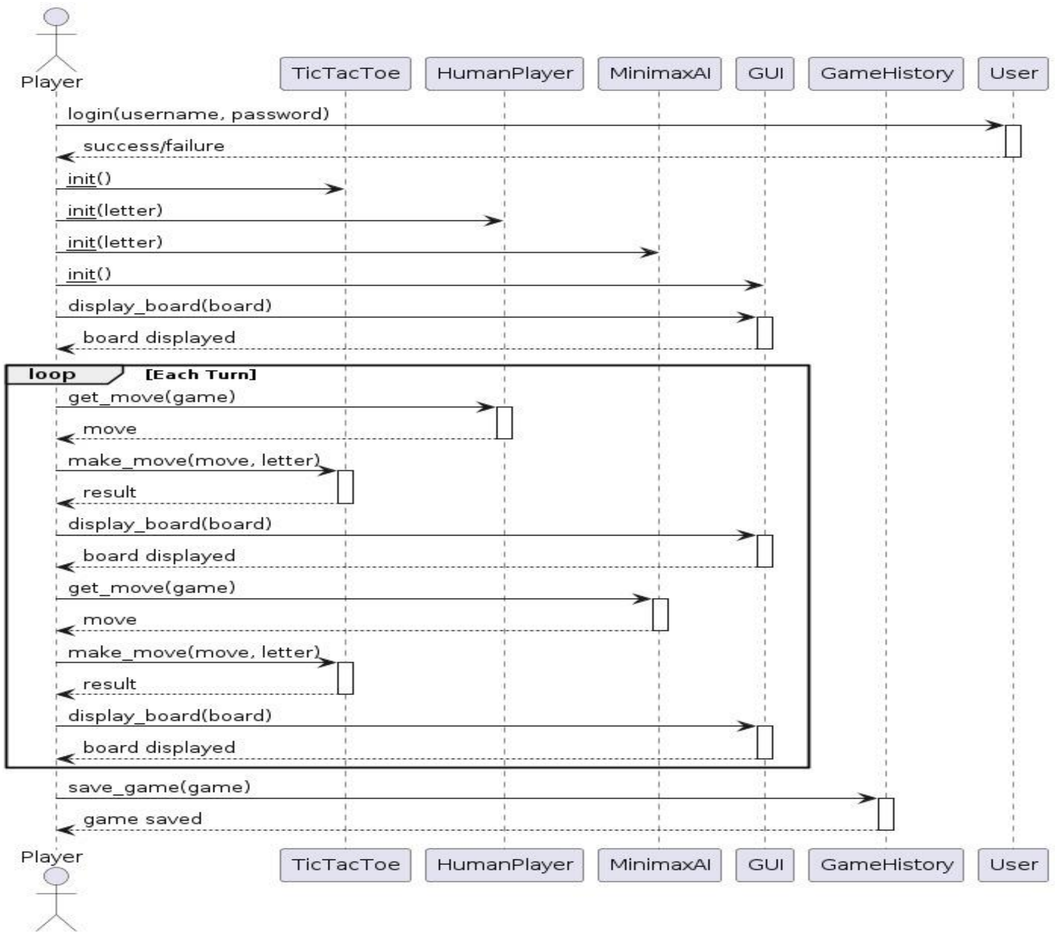
- Maintainability:

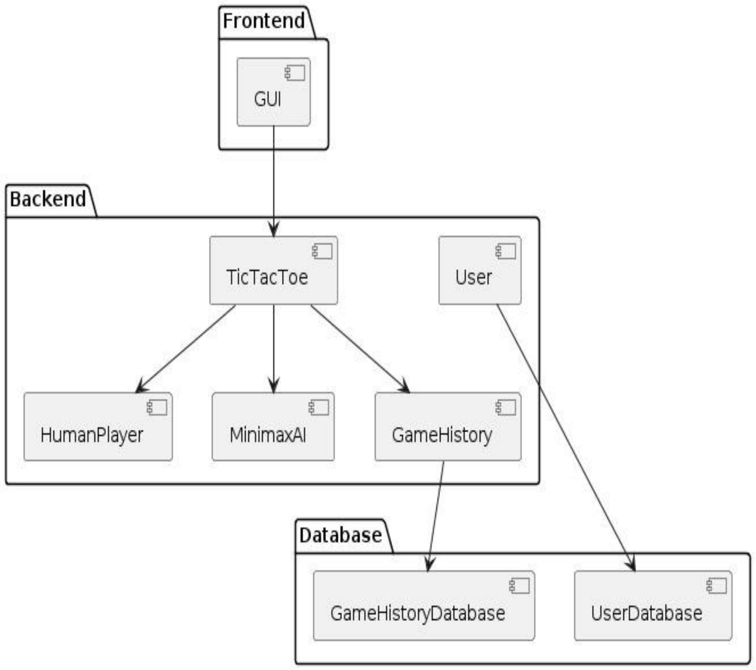
The code base is structured and documented to facilitate ease of maintenance and future updates, ensuring agility in adapting to evolving user needs and technological advancements.

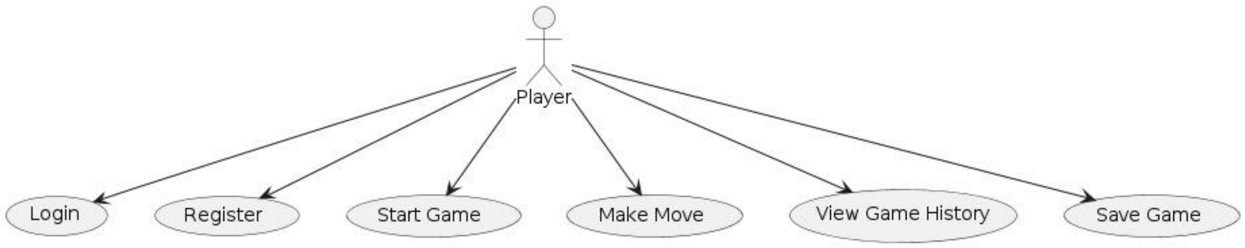
- Data Integrity:

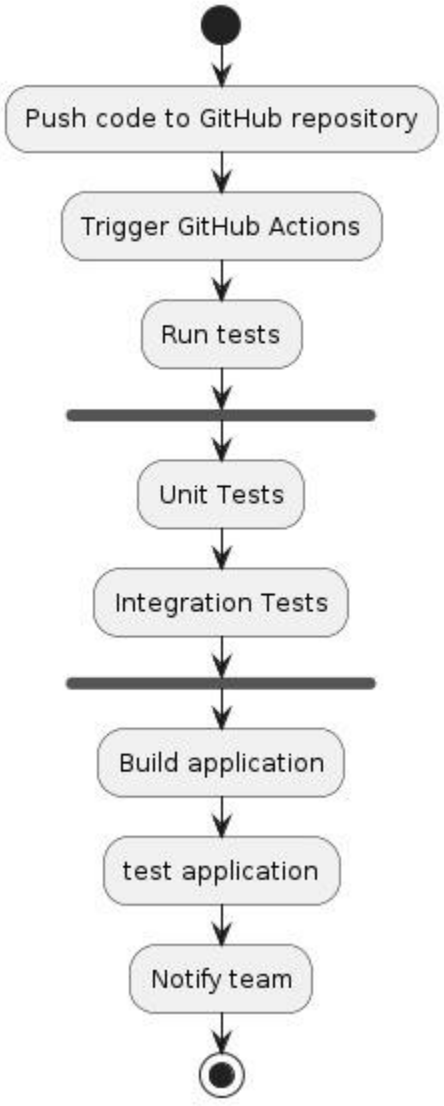
Measures are implemented to ensure the accuracy, consistency, and reliability of stored game data, preventing unauthorized access or corruption.

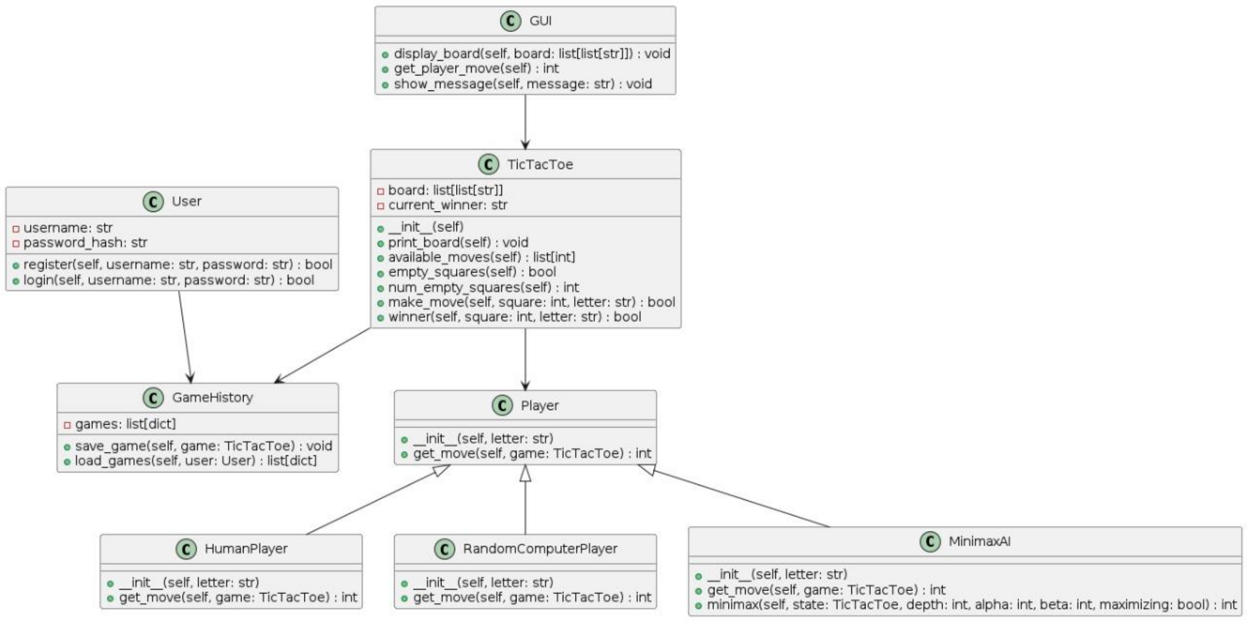
2. Software design specifications

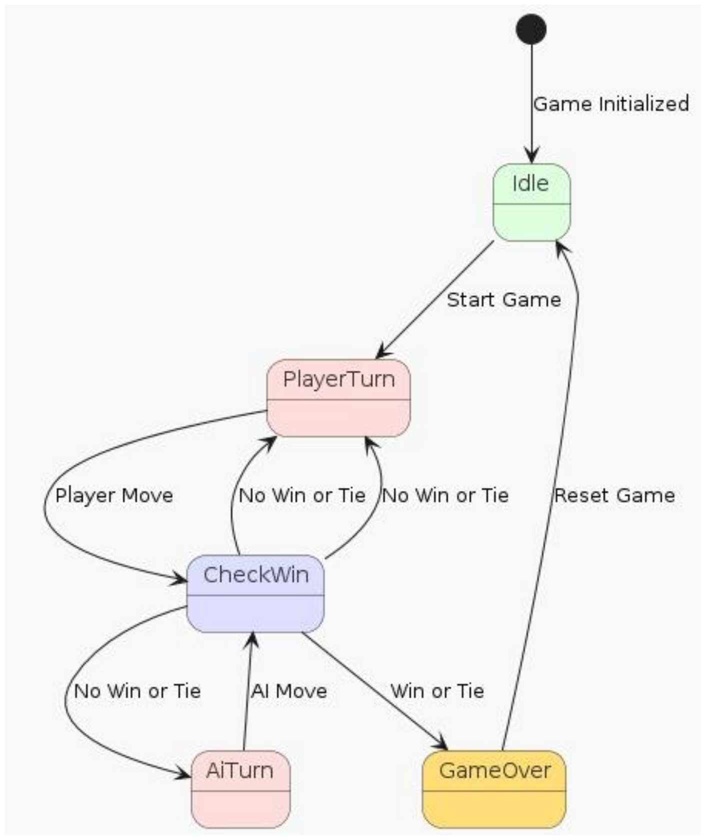





Player
Login
Register
Start Game
Make Move
View Game History
Save Game


Push code to GitHub repository
Trigger GitHub Actions
Run tests
Unit Tests
Integration Tests
Build application
test application
Notify team


GUI
display_board(self, board: list[list[str]]) : void
get_player_move(self) : int
show_message(self, message: str) : void
TicTacToe
board: list[list[str]]
current_winner: str
__init__(self)
print_board(self) : void
available_moves(self) : list[int]
empty_squares(self) : bool
num_empty_squares(self) : int
make_move(self, square: int, letter: str) : bool
winner(self, square: int, letter: str) : bool
User
username: str
password_hash: str
register(self, username: str, password: str) : bool
login(self, username: str, password: str) : bool
GameHistory
games: list[dict]
save_game(self, game: TicTacToe) : void
load_games(self, user: User) : list[dict]
Player
__init__(self, letter: str)
get_move(self, game: TicTacToe) : int
HumanPlayer
__init__(self, letter: str)
get_move(self, game: TicTacToe) : int
RandomComputerPlayer
__init__(self, letter: str)
get_move(self, game: TicTacToe) : int
MinimaxAI
__init__(self, letter: str)
get_move(self, game: TicTacToe) : int
minimax(self, state: TicTacToe, depth: int, alpha: int, beta: int, maximizing: bool) : int


Game Initialized
Idle
Start Game
PlayerTurn
Player Move
No Win or Tie
No Win or Tie
Reset Game
CheckWin
No Win or Tie
AI Move
Win or Tie
AiTurn
GameOver

4. Testing documentation

Game Is Over Tests
Test Scenario:
The scenario aims to verify that the game correctly identifies and handles the end of a game when the grid is completely filled.

Test Case:
The GameIsOverTests test case begins by loading the main QML file of the game and waits for it to be fully loaded. It then simulates a game by iteratively filling each cell in the grid. Once the grid is completely filled, the test case checks for the presence of the game over label to ensure that the game recognizes the game-over condition. It verifies that the label appears and that it displays the correct message, "Game Over!" This confirms that the user is properly informed when the game ends.

Registration Tests
Test Scenario:
This scenario tests the user registration functionality, ensuring users can successfully register with valid credentials.

Test Case:
The RegistrationTests test case loads the main QML file and identifies the registration components, including the username field, password field, email field, and register button. It verifies the presence of these components and inputs test data for the username, password, and email fields. The test case then simulates a mouse click on the register button. It checks for the appearance of a success label and verifies that it displays the message "Registration successful!" This ensures that the registration process works correctly and that users receive proper feedback upon successful registration.

Switch Tests
Test Scenario:
This scenario verifies that users can switch between the login and registration views within the application.

Test Case:
The SwitchTests test case loads the game and identifies the buttons for switching to the login and registration views. It first simulates a click on the button to switch to the login view and verifies that the login view becomes visible. Then, it simulates a click on the button to switch to the registration view and verifies that the registration view becomes visible. This ensures that the view-switching mechanism functions correctly, allowing users to navigate between different views seamlessly.

User Login Tests
Test Scenario:
This scenario tests both the user interaction with the game grid and the user login process.

Test Case:
The UserLoginTests test case consists of two parts. First, it tests the game grid interaction by simulating a mouse click on a specific grid cell (e.g., "gridCell_0_0") and verifying that the cell is marked with an "X". This checks the basic functionality of user interaction with the game grid. Second, it tests the login process by loading the game, locating the login components (username field, password field, and login button), and verifying their presence. The test case inputs test credentials and simulates a mouse click on the login button.

5. **CI/CD pipelining**

Workflow (.yml file)

```
name: Build Qt Project
on: push
jobs:
  build:
    name: Build Qt Project
    runs-on: ubuntu-latest
    steps:
      - name: Checkout repository
        uses: actions/checkout@v2
      - name: Set up dependencies
        run: |
          sudo apt-get update
          sudo apt-get install -y libxcb-xinerama0 libxcb-cursor0 libxcb-icccm4 libxcb-keysyms1
libxcb-image0 libxcb-render-util0 libxcb-xfixes0 libxkbcommon-x11-0
      - name: Install Qt
        uses: jurplel/install-qt-action@v3
        with:
          version: '6.7.0'
          modules: 'qtmultimedia'
      - name: Add Qt to PATH
        run: echo "$HOME/Qt/6.7.0/gcc_64/bin" >> $GITHUB_PATH
      - name: Create build directory
        run: mkdir build2
      - name: Build project
        run: |
          cd build2
          export PATH="$HOME/Qt/6.7.0/gcc_64/bin:$PATH"
          qmake ../finalqmakeproject.pro  # Replace with your actual .pro file name
          make
```

This GitHub Actions workflow automates the build process for a Qt project whenever changes are pushed to the repository. It runs on an Ubuntu environment and begins by checking out the repository code. It then installs necessary system dependencies required for Qt applications. Using the install-qt-action, it installs Qt version 6.7.0 along with the qtmultimedia module. The workflow subsequently updates the system PATH to include the Qt binaries, ensuring that Qt tools are accessible. It creates a dedicated build directory, and within this directory, it runs qmake on the specified project file to generate a Makefile and then uses make to compile the project. This process ensures a consistent and clean build environment, facilitating early detection of build issues.

6. Performance Measurement and Optimization

- Application initialization took: 51927900 nanoseconds.
- Main Window creation and show took: 179905800 nanoseconds.

	AI hard	**AI easy**	**Player-vs-player**
Player move timing average	**847776066.7 nanoseconds**	**654251025 nanoseconds**	**142888966.7 nanoseconds**
AI move timing average	**847677700 nanoseconds**	**654157375 nanoseconds**	**---------**

Memory usage: 350.7 MB

CPU: 5%