Chess Project
Introduction

Online Chess Game is a web-based platform designed to provide users with an interactive and seamless environment to play casual chess. It allows both multiplayer mode where players can challenge each other in real-time or single player mode compete against a computer opponent.

The platform features a simple, intuitive interface that ensures an enjoyable experience for all users. The game is built using the WebSocket protocol to
enable real-time synchronization, ensuring smooth communication between players.

The project aims to deliver a responsive and stable chess-playing experience through the effective use of these technologies, providing both competitive and casual players with a platform to enjoy the classic game of chess.

Problem Statement

Online chess platforms have made the game more accessible to everyone but players still face several challenges that make their experience bad. Key issues include:

1) Lack of Simple Multiplayer Chess Options: Many existing chess platforms are either very complex or most of them lack the simplicity required for casual users.

2) Real-Time Gameplay Issues: Achieving smooth, real-time gameplay is challenging, which leads to less enjoyable user experience.

3) Accessibility for Casual Players: There is a need for a straightforward chess platform that focuses on core gameplay only without overwhelming users with additional features.

Objective

1) To create a simple and accessible multiplayer chess game that allows users to connect and play against each other in real-time using WebSockets protocol, ensuring smooth and responsive gameplay.

2) To provide a distraction-free platform focused on basic chess functionality, catering to casual players who prefer a straightforward and easy-to-use interface.

3) To deliver a reliable and efficient application by using modern web technologies, ensuring that the game is accessible to a wide range of users across different devices.

Literature Review

1) Lichess

Lichess is a popular open-source chess platform offering real-time gameplay, puzzles, tournaments, and analysis tools. While feature-rich, it can overwhelm users seeking a simple experience. During peak times, occasional slowdowns and latency may occur.

2) Chess.com

Chess.com is known for its extensive features like multiplayer games, tutorials, and AI opponents. However, its interface can feel cluttered for casual users, and many advanced tools are locked behind a premium subscription, limiting access for those preferring a fully free experience.

3) Playchess

Playchess focuses on simplicity and direct online matches but lacks modern matchmaking features. Its reliance on manual server connections can be challenging for non-technical users.

4) Chess24

Chess24 offers a comprehensive chess experience with live streaming, puzzles, and tutorials. Popular among serious players, it can be overwhelming for casual users, with many advanced features available only through a subscription.

Requirement Analysis

Functional Requirement

I) Real-Time Gameplay
● The system must provide smooth, real-time synchronization of the chessboard between two players using WebSocket technology.

II) Multiplayer Match Initialization:
● Allow players to create or join matches via a game ID internally managed by the platform itself.

III) Basic Chess Functionality:
● Include essential chess rules and mechanics (e.g., check, checkmate, stalemate) using the python-chess library.

IV) Game State Persistence:
● Ensure that the game state is saved during active matches to allow players to reconnect in case of disconnections or other work.

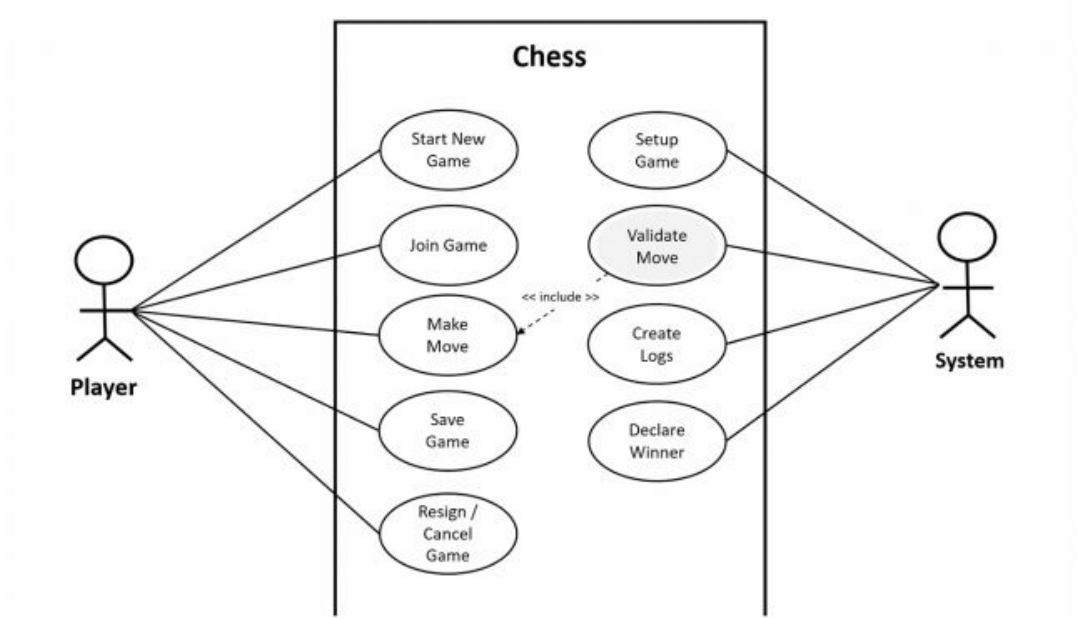


Figure 3. 1: Use Case Diagram for Functional Requirement

Non-functional requirement

I) Performance

● The application should ensure fast data processing and low latency for real-time gameplay, with WebSocket updates occurring within 100 milliseconds.

II) Scalability

● The system must handle an increasing number of concurrent games efficiently by managing WebSocket connections effectively.

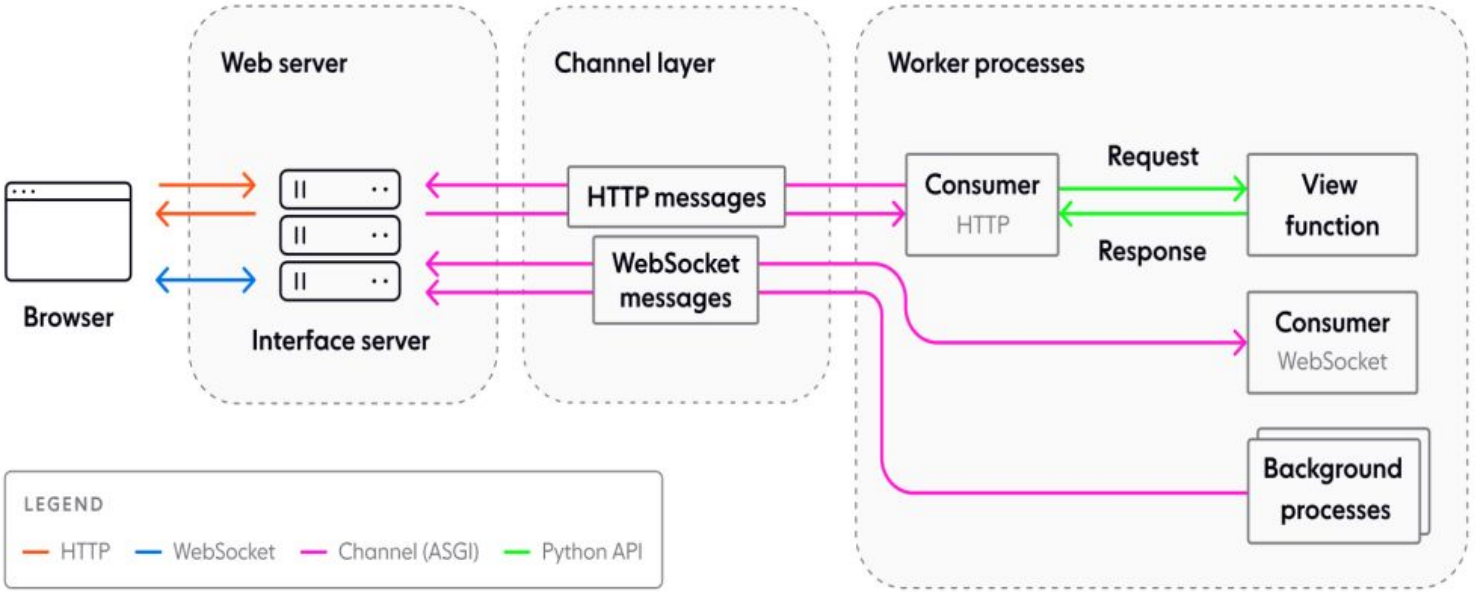
III) Compatibility

● It should also provide a responsive design that adapts to desktops, tablets, and smartphones, ensuring a consistent user experience.

IV) Security

● It should implement secure authentication methods to prevent unauthorized access to game sessions.

System Design (General Program Flow)



Algorithm Implementation

1) Websocket Synchronization Algorithm

The synchronization algorithm ensures that the chessboard state remains consistent for both players in real time, handles simultaneous move attempts, and manages player Reconnections.

Steps in the Algorithm:

i) Player Move Submission:

- Player A makes a move and sends it to the server over a WebSocket connection.

ii) Validation of Move:

- The server validates the move using the python-chess library.
- If the move is illegal, the server sends an error message back to Player A, and the state remains unchanged.

iii) Game State Update:

- If the move is valid:
  - The server retrieves the current game state from Redis.
  - Applies the move to the game state.
  - Saves the updated state back to Redis using atomic operations to prevent conflicts.

iv) Broadcast Update:

- The updated game state is sent to both Player A and Player B over WebSocket connections.

v) Reconnection Handling:

- If a player disconnects, the server keeps the game state in Redis.
- Upon reconnection, the server retrieves the latest game state and sends it to the player to resume the match.

2) Minimax Algorithm

1. Initialization

    i) Start the WebSocket server.

    ii) Initialize the chessboard using python-chess

    iii) Set the depth for the Minimax algorithm (e.g., 2 or 3).

    iv) Define the evaluation function to scoreboard states based on material advantage (e.g., pawns = 1, queens = 9).

2. Player's Move

    i) Receive Move:

    - Wait for the player to send a move in UCI format (e.g., e2e4) via WebSocket.

I i) Validate Move:

    - Parse the move using python-chess.
    - Check if the move is in the list of legal moves:
        - If invalid, send an error message back to the player.

- If valid, apply the move to the chessboard using board.push().

iii) Check Endgame Conditions:

- Use board.is_game_over() to check for game-ending conditions (checkmate, stalemate, etc.).
- If the game is over, send the result (checkmate, stalemate, etc.) to the player and end the game.

3. AI's Turn

i) Simulate All Possible Moves:

- For each legal move the AI can make:
  - Apply the move to the chessboard using board.push().
  - Call the Minimax function recursively to evaluate the Move.

ii) Evaluate Board State:

- At the base case of the recursion (depth = 0 or game over):
  - Use the evaluation function to assign a score to the current board state.
    - Positive scores favor the AI.
    - Negative scores favor the opponent.

iii) Backpropagate Scores:

- For the maximizing player (AI):
    - Return the maximum score among all possible moves.
- For the minimizing player (human):
    - Return the minimum score among all possible moves.

iv) Prune Irrelevant Branches:

- Use Alpha-Beta Pruning to skip branches of the game tree that cannot influence the final decision.

v) Select the Best Move:

- The move with the highest score from the root node is chosen as the AI’s move.

vi) Apply AI’s Move:

- Apply the selected move to the chessboard using board.push().
- Send the AI’s move to the player via WebSocket.

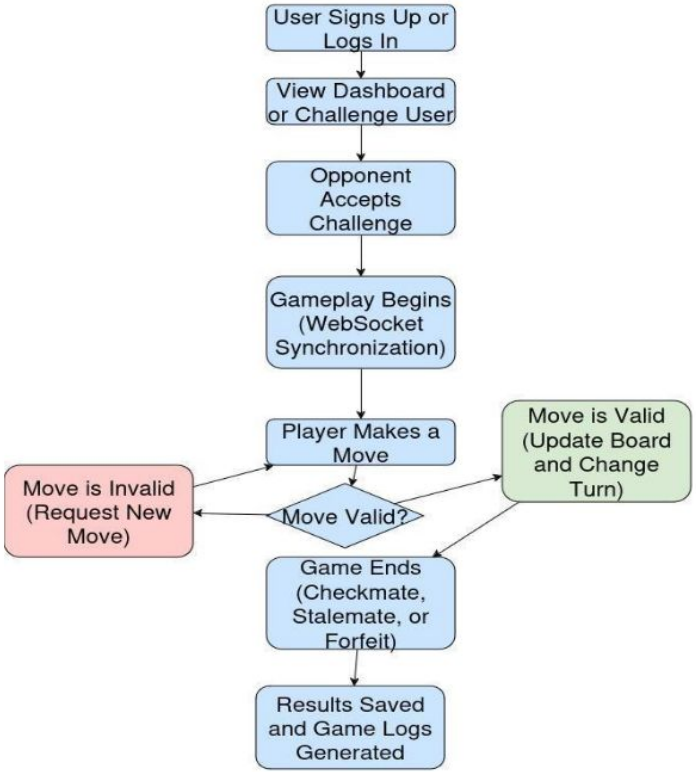
4. Repeat

i) Alternate turns between the player and the AI.

ii) Repeat until the game ends:

- Either due to checkmate, stalemate, or resignation.

iii) Send the final game result to the player.



FlowChart of Overall Gameplay

Conclusion

The Multiplayer Chess Game project has achieved its objectives, delivering a simple and reliable platform for real-time chess matches. Here are the key outcomes:

i) Seamless Gameplay: Real-time synchronization and rule enforcement ensure smooth and fair gameplay for players.

ii) User-Friendly Interface: The application features a clean, responsive, and easy-to-navigate design, providing a positive user experience across all devices.

iii) Reliable Backend: Built with Python and Django Channels, the backend efficiently handles game sessions, player interactions, and data management.

iv) Scalable Architecture: The modular system supports future expansion and integration of additional features, ensuring the platform adapts to user needs.

In summary, the Multiplayer Chess Game project successfully meets its goals, delivering a well-designed and efficient platform for casual and competitive chess players.