Chapter 1: Introduction

1.1 Introduction

Chess, which is one of the oldest games in history, is a widely played strategic game with millions of players all around the world playing it everyday. With the advancement in technology the way chess is played has also changed and now there are a lot of digital versions of the game freely available everywhere on the internet. Today, with the help of the internet millions of players are involved in multiplayer online chess games, playing with other people from all over the world and interacting with different communities and even conducting large scale tournaments.

These Online platforms have many advantages including real time match making, interactive lessons and chat and forums. However, there are some problems that are associated with the current online chess platforms. Some of the challenges include; poor matchmaking, buggy ui, and cheating bots, have made the gaming experience not so great for some casual users just trying to have fun playing the game.

This report is about the development of a multiplayer online chess game to solve these challenges. The project is based on the idea of making the interface easy to use, so that users can just invite their friends, and increasing the player's interaction by using some simple features. The next sections will discuss the problem statement, objectives, scope, and limitations of the project, as well as the development methodology and the organizational structure of the report.

1.2 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:

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

1.3 Objectives

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

1.4 Scope and Limitation

Scope

1. **Core Functionality**: The platform is designed to provide a simple and easy to use online chess game, focusing on essential chess gameplay without additional distractions or advanced features.
2. **Real-Time Gameplay**: Utilizing WebSocket protocol, the platform ensures smooth and responsive bidirectional communication for real-time player interactions even at heavy traffic.
3. **User-Friendly Interface**: A distraction-free, straightforward interface is provided to make the platform accessible and appealing to casual players.
4. **Matchmaking**: The platform supports basic matchmaking to connect players (especially friends) for quick games.
5. **Cross-Device Compatibility**: Developed with modern web technologies, the platform is optimized for use across various devices, including desktops, tablets, and mobile phones with proper responsiveness.

Limitations

1. **Lack of Cheat Detection**: The platform does not include cheat detection mechanisms. Players are expected to follow fair play practices voluntarily.
2. **No Advanced Features**: Features such as in-game chat, move analysis, or AI-driven insights are intentionally excluded to maintain simplicity and focus on core gameplay.
3. **Basic Matchmaking Only**: Matchmaking is limited to connecting available players without advanced ranking systems or skill-based pairing.

4. **No Offline Play**: The platform requires an active internet connection, as all interactions are conducted in real-time through WebSocket protocol.
5. **Limited Community Features**: Community engagement is almost none and lacks forums, leaderboards, or extensive social features.
6. **Focus on Casual Players**: The platform is not designed for advanced players seeking features like tournaments, chess puzzles, or professional-grade analysis tools, rankings etc.

By defining the scope and limitations, the project aims to deliver a straightforward and efficient platform that meets the needs of casual chess players while maintaining simplicity and accessibility.

1.5 Development Methodology

The methodology followed in this project is **Incremental Development**, where the application is built and tested in stages:

- **Implementation of Core Features**: Starting with essential features such as user authentication and gameplay mechanics.
- **Integration of WebSocket Synchronization**: Adding real-time communication using websocket Protocol to enable multiplayer functionality. [1]
- **Validation and Testing**: Testing game logic, including move validation and rule enforcement using **python-chess**. [2]
- **Scalability Enhancements**: Optimizing backend storage and synchronization using Redis and Django Channels . [3, 4, 5]
- **Frontend Enhancements**: Refining the user interface for a responsive and clean design.

This iterative approach allows incremental testing and improvement, ensuring a robust and user-friendly application.

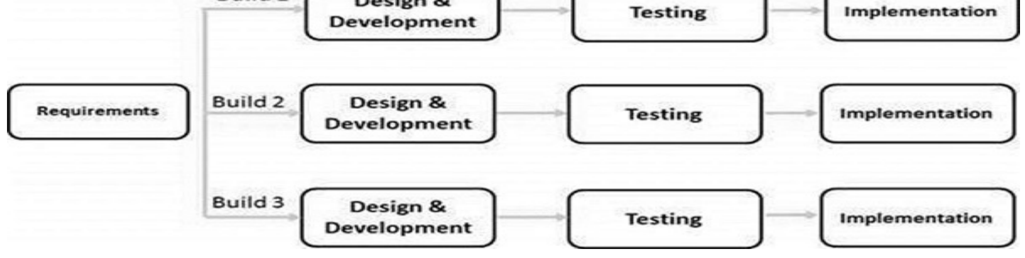


Fig: Incremental Development Model

1.6 Report Organization

Our report is divided into the following chapters:

Chapter 1: Introduction

This chapter introduces our project briefly, explains the problem statement, and outlines the scope and objectives that we are trying to achieve for our platform.

Chapter 2: Background Study and Literature Review

This section includes the research we conducted for this project, covering background information and Literature Review where we researched for similar projects and their shortcomings and strengths.

Chapter 3: System Analysis

This chapter discusses the research on feasibility analysis and system requirements for making the project successful.

Chapter 4: System Design

This section describes the design and development process of our project.

Chapter 5: Implementation and Testing

This chapter explains the implementation of the project along with different methods and tools used in implementation.Testing is also included in this section which checks whether the implementation is working properly or not.

Chapter 6: Conclusion and Future Recommendations

This section presents the final outcomes of the project based on the problem statement and objectives, along with future suggestions for the project.

Chapter 2: Background Study and Literature Review

2.1 Background Study

The study of chess combines both theoretical concepts and practical implementations. This section presents a fundamental overview of the key theories, general concepts, and terminologies used in the development of a multiplayer online chess game.

2.1.1 Fundamental Theories

1. **Game Theory**: Game theory is the study of strategic interactions among multiple decision-makers. In chess, players must anticipate their opponent's moves and develop strategies accordingly.
2. **Artificial Intelligence (AI) in Games:** The application of AI techniques in chess has a long history. Algorithms like Minimax and Alpha-Beta pruning are crucial for developing AI opponents that can compete with human players. More advanced techniques involve machine learning, particularly neural networks, which have been used in projects like AlphaZero.
3. **Network Theory:** Multiplayer online chess requires a solid understanding of network protocols and client-server architecture. Concepts such as latency, bandwidth, and data integrity are critical for ensuring a smooth gaming experience because if chess pieces don't move quickly it will decrease user experience.
4. **User Interface Design:** A well-designed user interface (UI) enhances player experience. UI principles such as usability, accessibility, and aesthetics must be considered when developing the chess game interface.

2.1.2 General Concepts

1. **Chess Rules:** Understanding the basic rules of chess, including piece movements, check, checkmate, stalemate, and special moves like castling, is essential for both game design and player engagement.
2. **Multiplayer Mechanics:** Implementing multiplayer functionality involves complex topics like managing sessions, player authentication, and real-time game updates. Concepts such as turn-based play, matchmaking, and leaderboard systems are important to enhance the multiplayer experience.

3. **Game State Management:** The game state represents the current position of all pieces on the chessboard, player turns, and other game variables. Efficient management of the game state is crucial for real-time updates and ensuring consistency such that it doesn't feel buggy for users.

2.2 Literature Review

This section reviews relevant literature, projects, and findings from other researchers that contribute to the development of multiplayer online chess games.

2.2.1 Similar Projects

a) Lichess

Lichess is a widely recognized open-source chess platform that provides real-time gameplay using WebSocket technology. It offers an extensive range of features such as various chess modes, puzzles, tournaments, and in-depth analysis tools. However, this many features can overwhelm users who are looking for a straightforward and distraction-free playing experience. Additionally, while the platform is generally efficient, the high volume of users, especially during peak times, can lead to occasional performance slowdowns and latency issues.[6]

b) Chess.com

Chess.com is one of the most popular platforms for online chess, boasting a vast array of features including multiplayer games, tutorials, puzzles, AI opponents, and tournaments. While these features make the platform versatile, the interface can feel cluttered and intimidating for casual users. Moreover, many advanced tools, such as game analysis and personalized coaching, are locked behind a premium subscription, which limits accessibility for users seeking a fully free and straightforward chess solution. This can make it less appealing to users who prefer simplicity without paying money.[7]

c) Playchess

Playchess is a more minimalistic platform aimed at connecting chess players online for direct matches. It focuses on basic functionality over advanced features, making it appealing to those seeking simplicity. However, the platform's reliance on manual server connections can be a significant hurdle for users who are not technically good or unfamiliar with such systems. It also lacks modern matchmaking features, which can make finding an opponent less efficient compared to other platforms.[8]

d) Chess24

Chess24 offers a comprehensive chess experience with features including live streaming of professional games, puzzles, tutorials, and interactive lessons. The platform also allows users to play online matches against others or AI opponents. While it is popular among serious chess enthusiasts, its advanced features and focus on professional chess can be overwhelming for casual players. Additionally, many of its premium features, such as in-depth game analysis and exclusive video content, are only accessible through a subscription, which might disappoint users looking for a free, no-frills platform.[9]

After researching all these platforms our chess platform takes good ideas from all these platforms while keeping things simple. From Lichess, we use WebSockets for real-time gameplay but make the interface easier for casual players. Chess.com has many features, but we focus on a free and simple experience without extra distractions or paid tools. Playchess shows the value of easy matchmaking, so we make connecting players quick and automatic without manual setup. Chess24 offers advanced content, but instead of professional tools, we focus on a basic and fun chess experience. By learning from these platforms, we create a smooth, easy-to-use chess game for everyone.

CHAPTER 3: SYSTEM ANALYSIS

3.1 System Analysis

For the development of the chess game, the **Incremental Model** was chosen due to its advantages in delivering the game in manageable parts while allowing flexibility and timely feedback. This approach allows us to build the game incrementally, with each iteration or increment adding new functionality and improving the game. The following sections describe the methodology and the rationale for using the Incremental Model in the development of the chess game.

3.1.1. Requirement Analysis

i. Functional Requirement

I) Real-Time Gameplay:

- The system must provide smooth, real-time synchronization of the chessboard between two players using WebSocket technology.
- Ensure all moves are reflected immediately for both players, with validations applied by the chess engine to ensure the legality of moves.

II) Multiplayer Match Initialization:

- Allow players to create or join matches via a game ID internally managed by the platform itself.
- Simplify the process of connecting two players, ensuring accessibility for non-technical users.

III) Basic Chess Functionality:

- Include essential chess rules and mechanics (e.g., check, checkmate, stalemate) using the python-chess library.
- Provide clear feedback to players when a move is invalid or when the game ends.

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.
- Store completed game logs for review or reference.

V) Clean and Simple Interface:

- Design a minimalistic interface that displays only the chessboard, player names, and game status.
- Avoid unnecessary features or distractions.

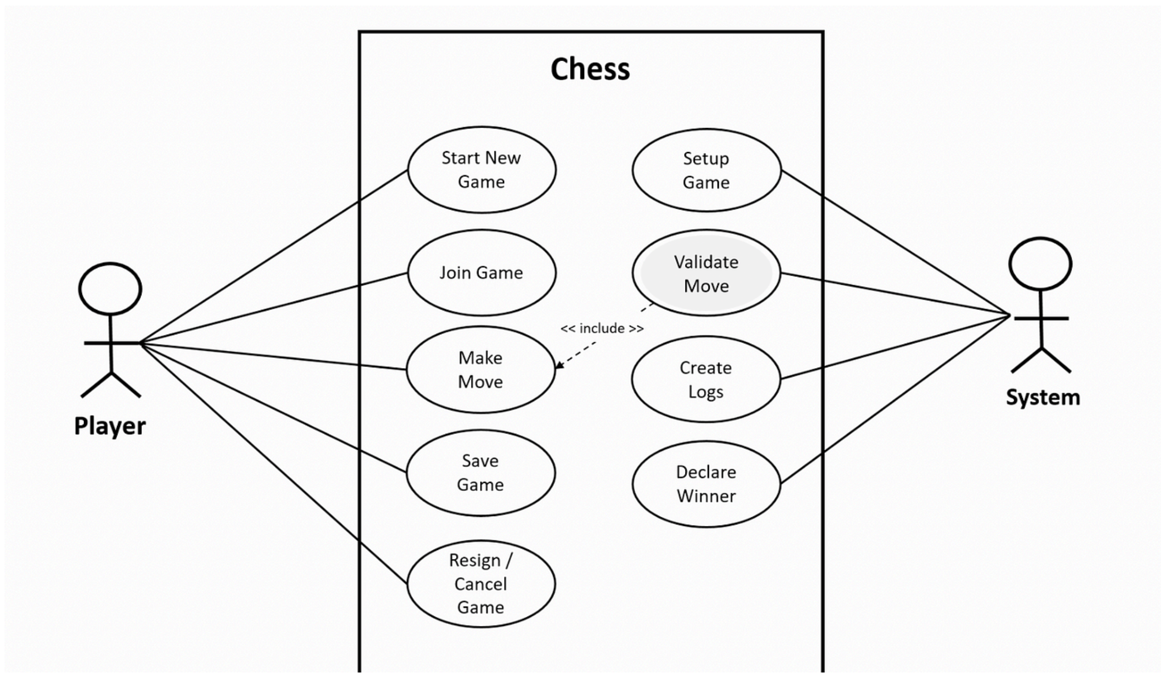


Figure 3. 1: Use Case Diagram for Functional Requirement

ii. 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.
- The chessboard and game interface must load within 2 seconds(average time) to maintain a smooth user experience.

II) Scalability

- The system must handle an increasing number of concurrent games efficiently by managing WebSocket connections effectively.
- It should be designed to support additional features in the future without significant architectural changes.

III) Compatibility

- The application must function seamlessly across all major web browsers, including Chrome, Firefox, Safari, and Edge.
- It should also provide a responsive design that adapts to desktops, tablets, and smartphones, ensuring a consistent user experience.

IV)Usability

- The interface must be simple and intuitive, catering to players of all skill levels.
- It should feature a clean layout focused on the chessboard and essential game information, avoiding unnecessary distractions or complex navigation.

V) Security

- The system must ensure secure WebSocket communication, encrypting all transmitted data to protect game states and user information.
- It should implement secure authentication methods to prevent unauthorized access to game sessions.

3.1.2 Feasibility Analysis

i. Technical Feasibility

The Multiplayer Chess Game uses popular tools like HTML, CSS, JavaScript, andPython, which is widely known and easy to learn. Django, used for the backend, is highly accessible, with many tutorials, books, and online videos available for guidance. For real-time communication, WebSockets are implemented using DjangoChannels, which are well-documented and simple to integrate. Redis is chosen for its fast performance and ease of use, making it ideal for storing game states. This combination of tools ensures the project is technically feasible and easy to maintain.

ii. Operational Feasibility

The application is built to work on both computers and mobile devices, ensuring players can access the game from any platform. Its clean and simple interface makes it easy to use for players of all skill levels, including those who are new to online chess. Hosting the game on a cloud platform ensures it is always available, and regular updates will be planned to keep the application running smoothly. This design ensures the project meets the needs of users without requiring complex navigation or technical knowledge.

iii. Economic Feasibility

The project uses free tools and technologies, making it affordable to develop and deploy. Costs will primarily involve hosting the application, which is minimal due to its lightweight nature. By focusing on core features without unnecessary complexity, the project offers a cost-effective solution that provides value to users. This approach ensures the project remains budget-friendly while delivering a simple and reliable chess platform.

iv. Schedule Feasibility

The game can be developed in stages, with each increment contributing to a working version of the game. Each increment can be completed in a set period, allowing for a predictable schedule of releases and updates.

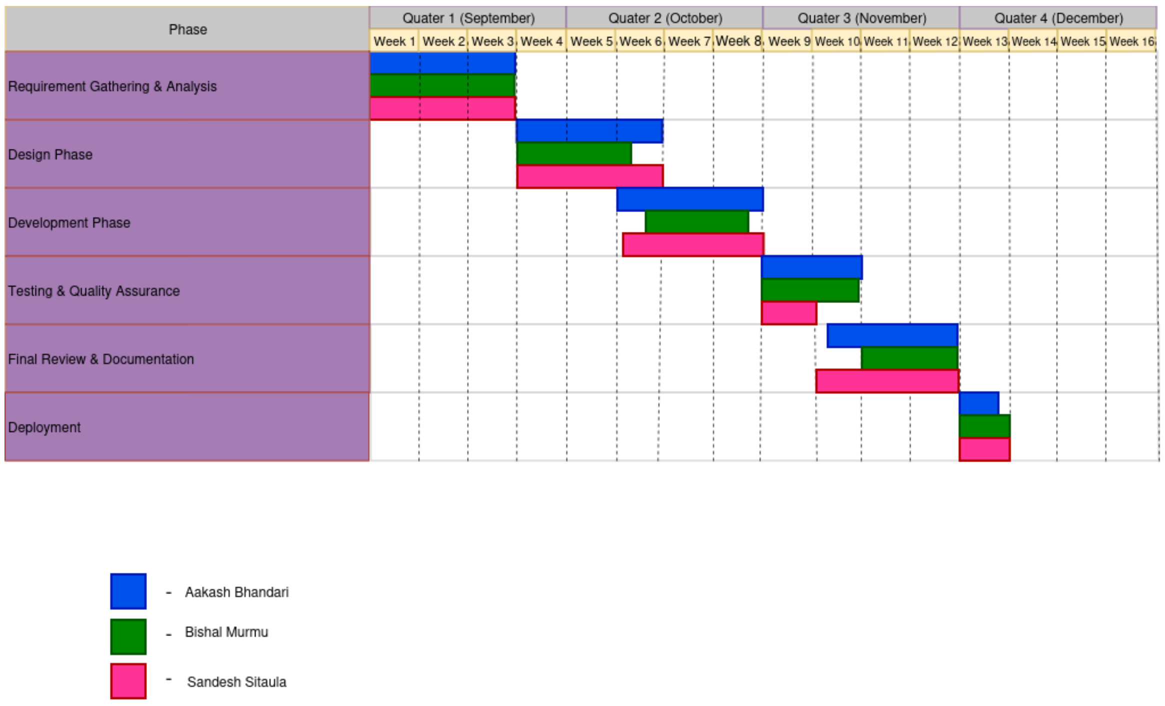


Figure 3. 2: Gantt Chart

3.1.3 Analysis

i) Data modeling using E-R Diagram

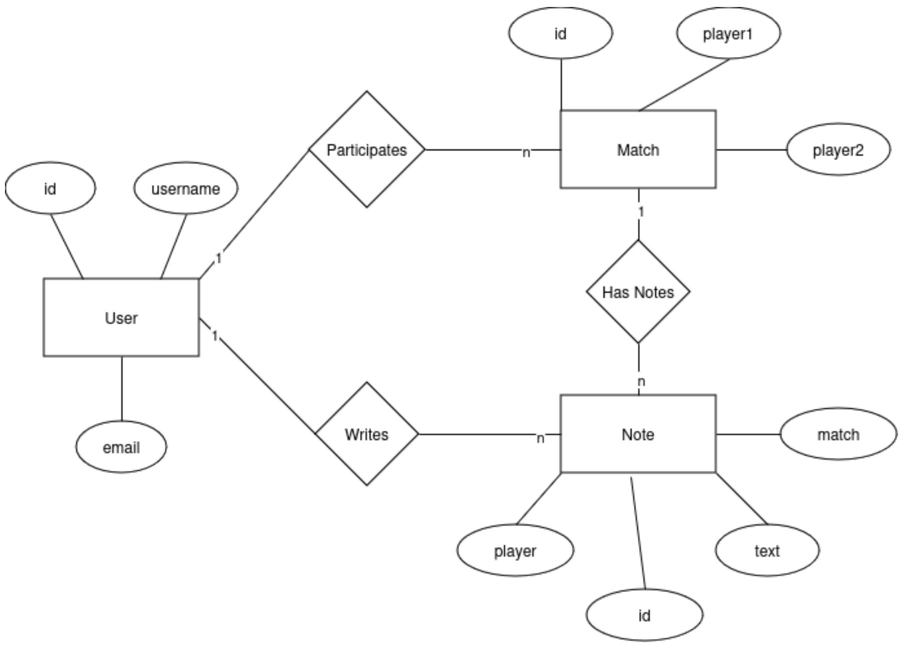


Figure 3. 3: E-R diagram

ii) Level 0 DFD

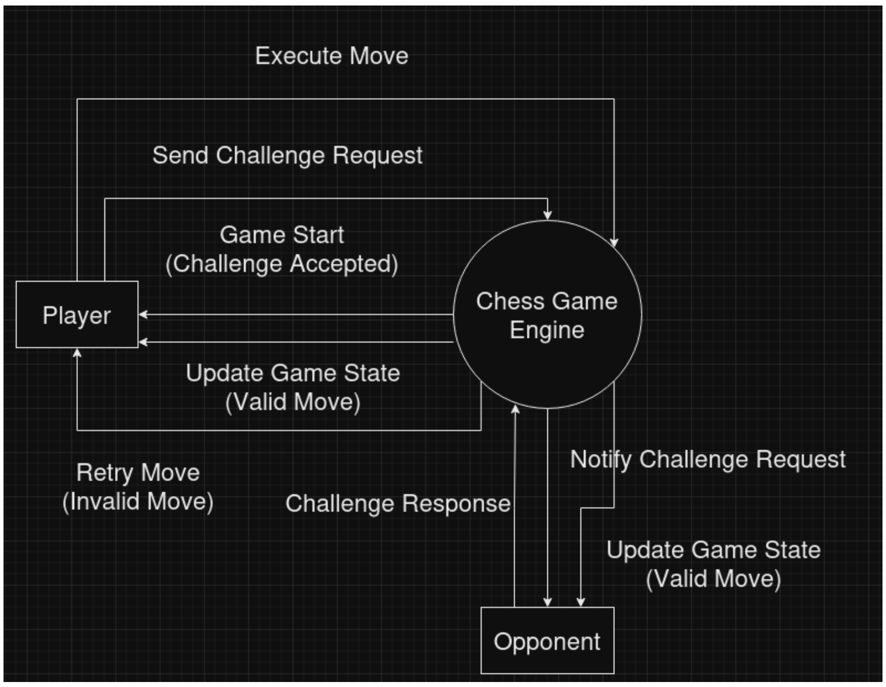


Figure 3. 4: Level 0 Data Flow Diagram

iii) Level 1 DFD

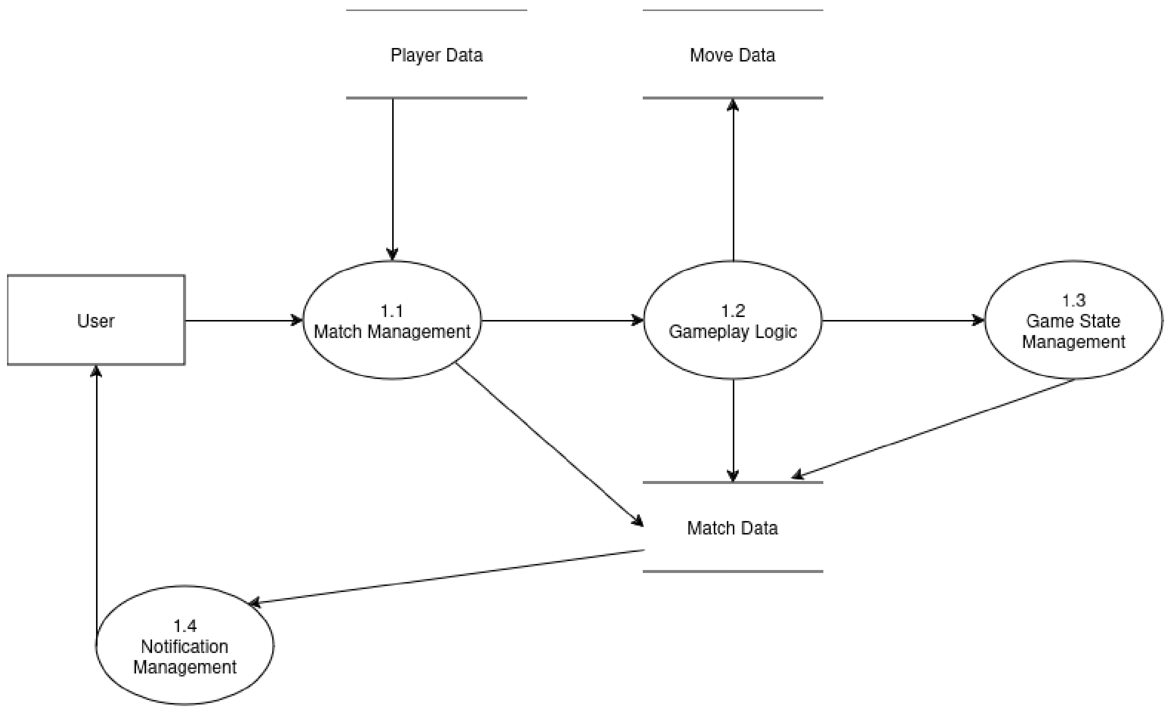


Figure 3. 5: Level 1 Data Flow Diagram

CHAPTER 4: SYSTEM DESIGN

4.1.1) Database Design (Transformation of ER to Relation Diagram)

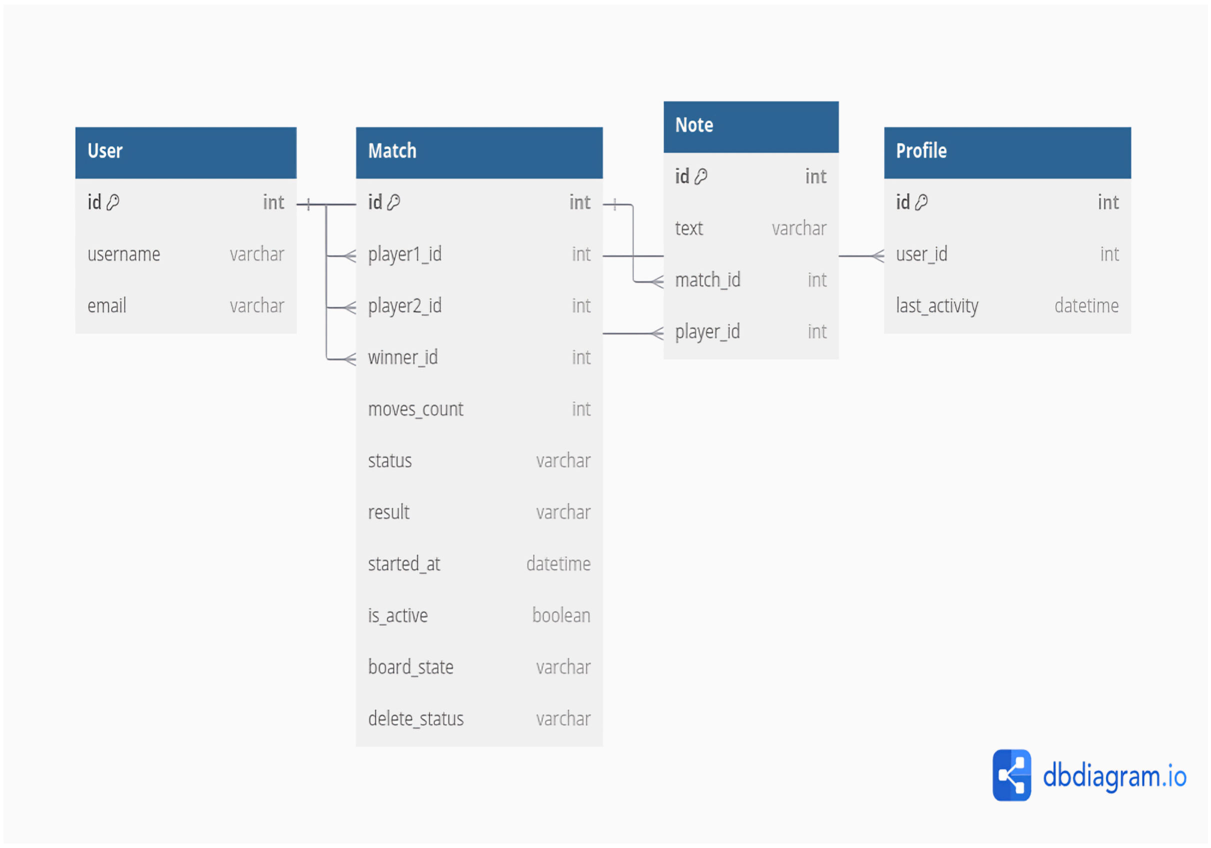


Fig 4.1) Relation Diagram

4.1.2) Illustrative Diagram System Design

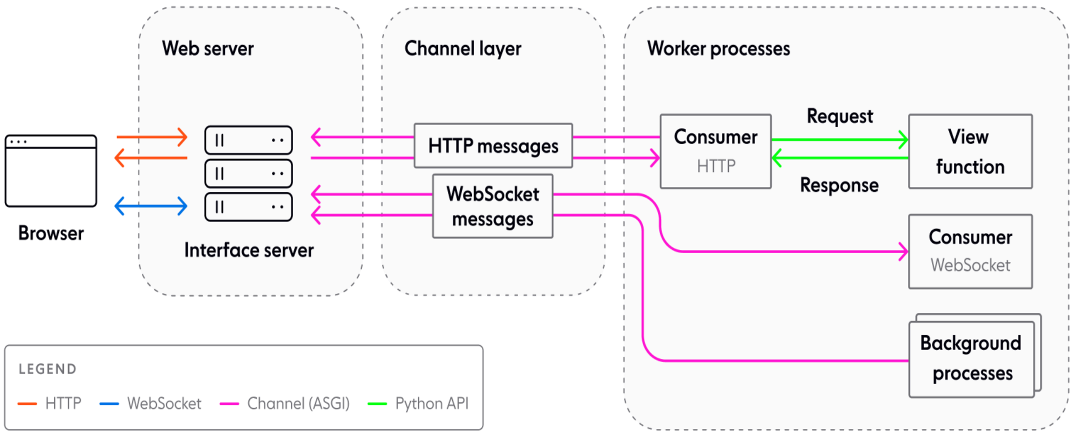


Figure 4.2) Illustrative Diagram1

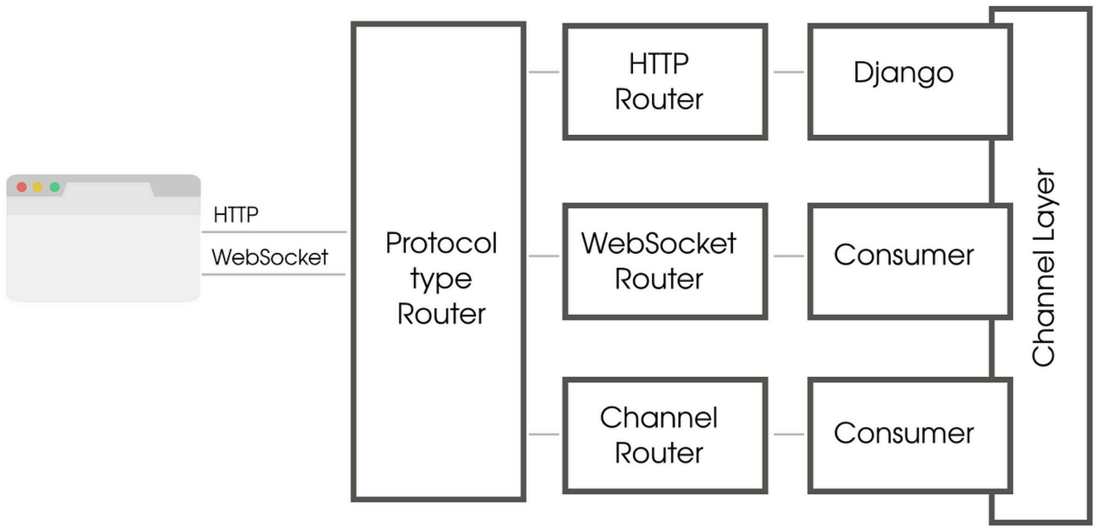


Figure 4.3) Illustrative Diagram2

4.2) Algorithm Details

4.2.1) Websocket Synchronization Algorithm

Overview:

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:

1. **Player Move Submission:**
   - Player A makes a move and sends it to the server over a WebSocket connection.
2. **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.
3. **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.

4. **Broadcast Update:**
    - The updated game state is sent to both Player A and Player B over WebSocket connections.
5. **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.

4.2.2) Checkmate Detection Algorithm

The **Checkmate Detection Algorithm** determines whether a player's move has resulted in a checkmate, signaling the end of the game. This algorithm validates the current board state after each move and declares the appropriate result if a checkmate condition is met.

Steps:

1. **Move Execution**
    - After a player makes a move, the server applies the move to the current game state (chessboard) and evaluates the board.
2. **Check Detection**
    - The algorithm first checks if the opponent's king is in a state of check:
        - A king is in check if it is under immediate threat from any opposing piece.
        - This validation is done using the rules of chess or a library like python-chess.
3. **Legal Moves Evaluation**
    - If the opponent's king is in check, the algorithm evaluates all possible legal moves for the opponent:
        - If no legal moves exist that can remove the king from check, it is a checkmate.
        - If legal moves are available, the game continues as usual.
4. **Checkmate Confirmation**
    - When the algorithm confirms a checkmate:
        - The server declares the player who delivered the checkmate as the winner.
        - The game is marked as completed in the database or game state store.

5. **Game State Update**
    - The server updates the game state to reflect the checkmate:
        - The final board state is stored.
        - The result is recorded as "Checkmate."
        - The winner's details are saved.

6. **Result Broadcasting**
    - The server sends the final game state and the result to both players:
        - This includes the winning player's identity and the board position at the time of checkmate.
7. **Edge Cases**
    - If the game ends in checkmate but both players' timers expire simultaneously, the server determines the result based on timing priority or a pre-defined rule.
    - If a player resigns after delivering a checkmate, the resignation takes precedence.

4.2.3) Minimax Algorithm

Algorithm Steps:

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 score board 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.

    ii) 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.

4.4.5) Flowchart of Gameplay / Working Mechanisms

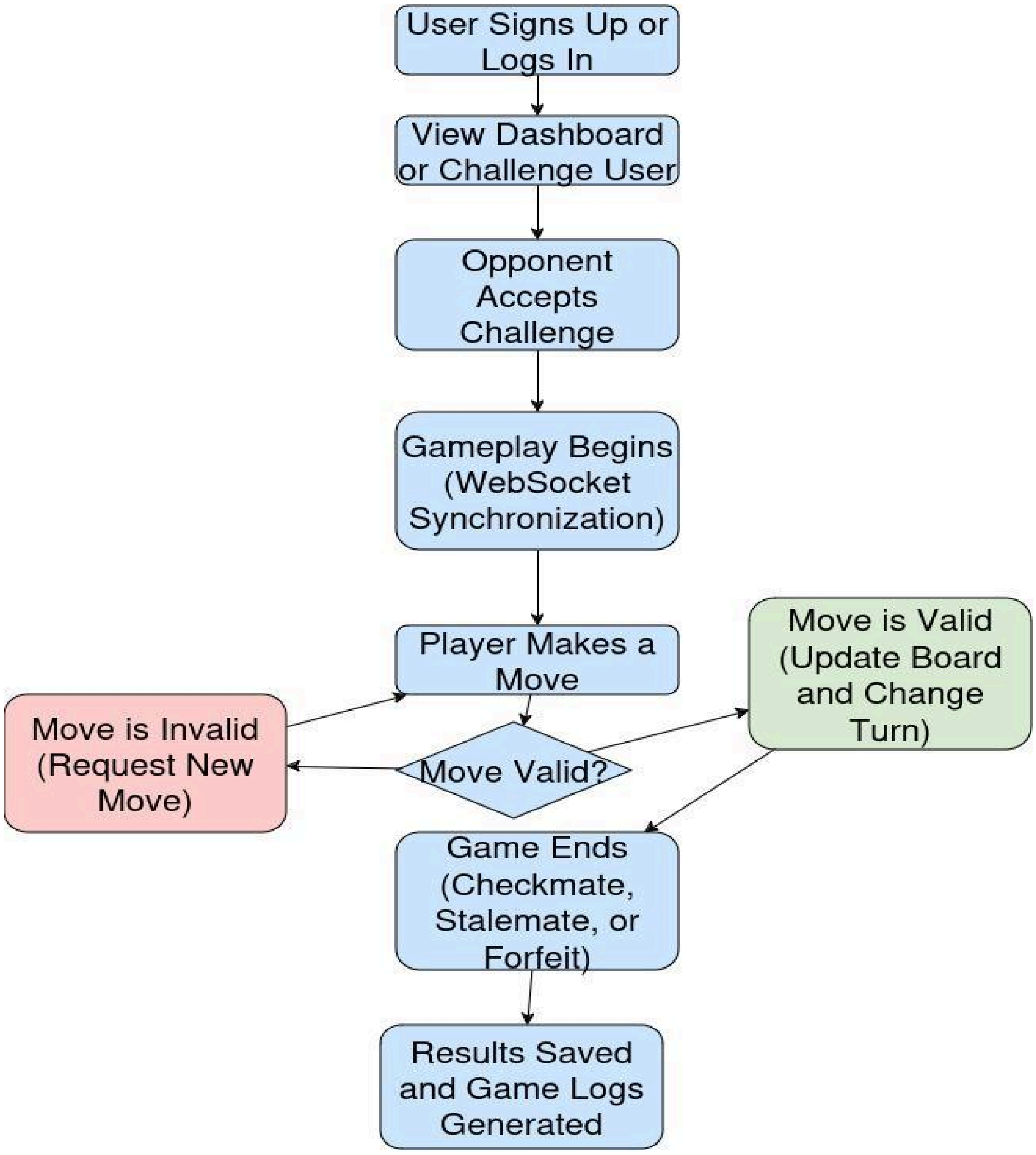

User Signs Up or Logs In
View Dashboard or Challenge User
Opponent Accepts Challenge
Gameplay Begins (WebSocket Synchronization)
Player Makes a Move
Move Valid?
Move is Invalid (Request New Move)
Move is Valid (Update Board and Change Turn)
Game Ends (Checkmate, Stalemate, or Forfeit)
Results Saved and Game Logs Generated

Figure 4.4) Flowchart of system implementation

CHAPTER 5: Implementation And Testing

5.1) Implementation

5.1.1) Tools Used

i) Backend Framework: Django 4.2.16

ii) Programming Language: Python 3.12

iii) Communication Protocol: Websockets

iii) Library To implement WebSocket Protocol: Django Channels 4.1.0

iv) Database Platform: Sqlite3

v) Channel Layer: redis

i) Backend Framework: Django 4.2.16

We chose Django because of its robust ecosystem, built-in features, and seamless integration with tools like Django Channels. It follows the MVC (Model-View-Controller) architectural pattern, which is highly suitable for organizing our application. Django's ORM makes database interactions efficient, while its settings module and middleware simplify configuration and security. Templates are used to dynamically generate the front-end, while ASGI support ensures compatibility with WebSocket communication. This setup ensures a scalable and maintainable backend structure. Also the resources and community for the Django ecosystem is quite large which makes problem solving easier incase of issues in future. [10]

ii) Programming Language: Python 3.12

Python was selected for its readability, extensive library support, and strong community backing. Mainly the language's asynchronous capabilities (async/await) are critical for handling WebSocket connections efficiently in our application. Python's compatibility with frameworks like Django further simplifies backend development. Database operations cannot be used in websockets directly because of its synchronous nature so python latest version further helps in this type of orm operations.

iii) Communication Protocol: WebSockets

WebSockets were used to enable real-time, bidirectional communication between the client and the server. This protocol minimizes latency and eliminates the need for repeated HTTP requests, making it ideal for features like player challenges, real-time game state updates, and move handling. Compared to traditional polling mechanisms, WebSockets are more resource-efficient and provide a smoother user experience.

iv) Library for Implementing WebSocket Protocol: Django Channels 4.1.0

Django Channels extends Django’s capabilities to support asynchronous protocols like WebSockets. It was chosen to abstract away lower-level complexities, such as channel layers and protocol handling, allowing us to focus on application logic. The AsyncWebsocketConsumer class from channels.generic.websocket was used in consumer_gameplay.py and consumer_home.py to handle WebSocket connections. Django Channels integrates seamlessly with Django’s authentication system, enabling us to retrieve user information during WebSocket sessions using the AuthMiddlewareStack.

v) Database Platform: SQLite3

SQLite3 was chosen as the database for development due to its simplicity and zero-configuration setup. It is integrated into Django by default, allowing quick prototyping without requiring an external database. SQLite3's lightweight nature makes it suitable for testing and smaller-scale applications. In future it can be switched to a more robust option like PostgreSQL incase sqlite3 cannot handle load.

vi) Channel Layer: Redis

Redis was selected as the channel layer backend due to its high performance and reliability in managing real-time messaging. It enables communication between different consumers through pub/sub mechanisms, ensuring scalability and responsiveness in WebSocket implementations. Redis’s stability and support for advanced data structures make it more suitable than the in-memory channel layer for production use.It is also quite easier to use and integrate using libraries like channels-redis. So, No complex configuration needs to be performed.

5.1.2) Implementation Details of Modules

Module Implementation Details

I. Channels and WebSocket Integration

The implementation of WebSocket communication using Django Channels forms the backbone of our real-time system. Django Channels enables asynchronous communication, allowing WebSocket connections to handle real-time events efficiently. The key components include:

1. **WebSocket Consumers**:
   - We implemented WebSocket consumers in consumer_gameplay.py and consumer_home.py by inheriting the AsyncWebsocketConsumer class from channels.generic.websocket.
   - The connect method is used to accept or decline incoming WebSocket connections. Groups are created here to associate WebSocket clients with specific contexts, such as games or user challenges.
   - The disconnect method performs cleanup tasks, such as removing users from groups.
   - The receive method listens for events from the frontend. Messages are processed based on their type, and appropriate responses are sent back using self.send or group-based messaging.
2. **Group Messaging**:
   - Groups are dynamically created to manage WebSocket communication for specific use cases, such as managing game state updates or broadcasting challenges.
   - For example, when a challenge is accepted, the challenging user’s group receives a notification.
3. **ASGI Configuration**:
   - WebSocket routing is defined in asgi.py, linking URL patterns to the respective WebSocket consumers.
   - The AuthMiddlewareStack is used to retrieve the authenticated user’s information during WebSocket sessions, simplifying user-specific operations.

4. **Redis as Channel Layer**:
    - Redis is configured as the channel layer backend for message passing between consumers. It ensures stability and scalability, handling heavy traffic loads and maintaining real-time updates efficiently.

II. Redis Integration

Redis serves as the backbone for managing session data and game states during real-time communication. Its features make it indispensable for our architecture:

1. **Pub/Sub Mechanism**:
    - Redis's publish/subscribe model facilitates efficient communication between WebSocket consumers. Messages are broadcast to subscribers, ensuring real-time updates to all relevant clients.
2. **Session Persistence**:
    - Redis stores session-related data, including game states and user connections, ensuring consistency even in case of client or server restarts.
3. **Atomic Operations**:
    - Redis ensures that game state updates (e.g., moves in a chess game) are atomic, preventing race conditions in multi-user environments.

III. Python-Chess for Move Validation

The python-chess library is used to validate and manage chess moves. This library simplifies the handling of chess logic and integrates seamlessly with our real-time communication framework.

1. **Move Validation**:
    - Each move sent by the client is validated using python-chess to ensure it adheres to chess rules.
    - If the move is invalid, an error message is sent back to the client, and the game state remains unchanged.
2. **Game State Management**:
    - The current board state is maintained using python-chess. Moves are applied using board.push().
    - Checkmate and stalemate conditions are checked after every move using methods like board.is_checkmate() and board.is_stalemate().

3. **Synchronization with Redis**:
    - Valid moves are saved to Redis, ensuring that the game state is always synchronized across clients.
    - If a player disconnects, the game state is retrieved from Redis and sent to the reconnecting player to resume the match seamlessly.

IV. Checkmate Detection Algorithm

The checkmate detection algorithm determines whether a player's move results in a checkmate. Key steps include:

1. **Move Execution**:
    - After a move is received, it is applied to the board using python-chess.
2. **Check Detection**:
    - The king's safety is assessed to see if it is in check, using board.is_check().
3. **Legal Moves Evaluation**:
    - If the king is in check, all possible moves are evaluated. If no legal moves exist, a checkmate is declared.
4. **Game State Update and Notification**:
    - The server updates the game state and sends the result (checkmate, stalemate, or draw) to both players.

V. Minimax Algorithm with WebSocket and Python-Chess

For AI-based gameplay, the Minimax algorithm is integrated with WebSockets and python-chess. Steps include:

1. **Initialization**:
    - The chessboard is initialized, and the depth of the Minimax algorithm is set.
    - The evaluation function scores board states based on material advantages.
2. **Player Move Handling**:
    - Player moves are received over WebSocket, validated, and applied to the board.
    - Endgame conditions are checked after each move.
3. **AI Move Simulation**:
    - The AI evaluates all legal moves using the Minimax algorithm.
    - Scores are calculated recursively, considering the opponent's responses.

4. **Move Execution**:
    - The best move for the AI is selected and applied to the board.
    - The updated game state is sent to the player over WebSocket.

These components collectively ensure a robust, real-time chess application with seamless user experience and efficient backend processes.

The Code Snippets are provided at the appendix section for further information about its implementation.

5.2) Testing

To ensure the chess application functions seamlessly and delivers a reliable user experience, a thorough testing strategy has been implemented. The testing approach includes unit testing and integration testing, covering both individual components and the interaction between them.

5.2.1) Unit Testing

Unit testing focuses on verifying the functionality of individual components in isolation. The primary goal is to ensure that each module behaves as expected under various conditions.

5.1) Here is the unit testing table based on test cases:

S.N.	TestCase Name	Test Case Description	Step	Expected Result	Status
1	Validate Move	Check if the move validation method works correctly.	Try moving a pawn from e2 to e4 (valid) and e2 to e5 (invalid).	Valid move returns True, invalid move returns False.	Pass
2	Execute Move	Ensure the move execution updates the board correctly.	Execute move from e2 to e4 and check board state.	The piece should be on e4 if move was valid	Pass
3	Get Board	Ensure board position	Retrieve	“E2” should have a	Pass

	Position	mapping returns correct data.	board position data.	pawn symbol.	
4	Initialize Board	Ensure the board gets initialized correctly to initial state.	Call initialize_board() method from ChessEngine class.	Board should reset to the standard FEN position.	Pass
5	Checkmate Detection	Verify if the game detects Checkmate.	Execute a checkmate sequence.	is_match_over() should return true	pass
6	Retrieve Match Result	Ensure the match result is returned correctly.	Execute a checkmate sequence for black color.	get_match_result() should return “0-1”	Pass
7	Board Evaluation	Test if the board evaluation function returns an integer.	Call evaluate_board() from chessEngine class.	Should return an integer score.	Pass
8	MiniMax Algorithm	Ensure minimax evaluation returns an integer.	Call minimax(depth=3,is_maximizing=True) for pc.	Should return an integer evaluation.	Pass
9	Get Best Move	Ensure Minimax algorithm finds a valid move.	Call get_best_move(depth=3) which calls the minimax algorithm.	Should return a legal move from the board state.	Pass

Table: 5.1) Test Case For Unit Testing

Code Reference: Unit tests have been implemented for critical components, as outlined in the appendix section, which includes test cases for move validation, checkmate detection, MinMax algorithm Logic.

5.2.2) System Testing

System testing ensures that the individual components work together as expected. It verifies the overall flow of the application, focusing on interactions between the chessboard, WebSocket, Redis, and AI logic.

Here is the System testing table based on test cases:

S.N.	TestCase Name	Test Case Description	Step	Expected Result	Status
1	Player Move Handling	Test the complete process of handling a player move via websocket.	Send a move via Websocket and validate the query.	If Move is applied correctly then the game state is updated else an error is shown at top.	Pass
2	MiniMax Gameplay	Test if the Minimax algorithm generates valid moves and interacts correctly.	Trigger minimax algorithm move generation and apply it to the board.	The move is valid and game state is updated.	Pass
3	Redis State Management	Ensure game state is synced across clients and restored correctly after a reconnection.	Disconnect and reconnect a player, then check game state.	Game state remains consistent after reconnection.	Pass
4	Game End Scenarios	Test game-ending conditions such as checkmate, stalemate and draw.	Simulate game-ending moves and verify the result.	Correct game result is displayed	Pass

Table: 5.2) Test Case For System Testing

CHAPTER 6: Conclusion and Future Recommendation

6.1) 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:

- **Seamless Gameplay:** Real-time synchronization and rule enforcement ensure smooth and fair gameplay for players.
- **User-Friendly Interface:** The application features a clean, responsive, and easy-to-navigate design, providing a positive user experience across all devices.
- **Reliable Backend:** Built with Python and Django Channels, the backend efficiently handles game sessions, player interactions, and data management.
- **Scalable Architecture:** The modular system supports future expansion and integration of additional features, ensuring the platform adapts to user needs.
- **Secure Deployment and Support:** The application is securely hosted and regularly updated, ensuring reliable performance and protection of user data.

In summary, the Multiplayer Chess Game project successfully meets its goals, delivering a well-designed and efficient platform for casual and competitive chess players. The application's thoughtful implementation, focus on user satisfaction, and adaptability ensure it is both a functional and sustainable solution, ready to evolve with future requirements.

6.2) Future Recommendations

Migration to PostgreSQL

- Transition the database from the current solution to PostgreSQL for better scalability, advanced querying capabilities, and improved performance in handling larger datasets.

Adding a Game Timer

- Integrate a game timer to enhance competitive gameplay, allowing players to set time limits for their moves or the entire game.

Implementing AI Opponent

- Add an AI feature to enable players to practice and play against different difficulty levels, improving their skills and making the platform more versatile.

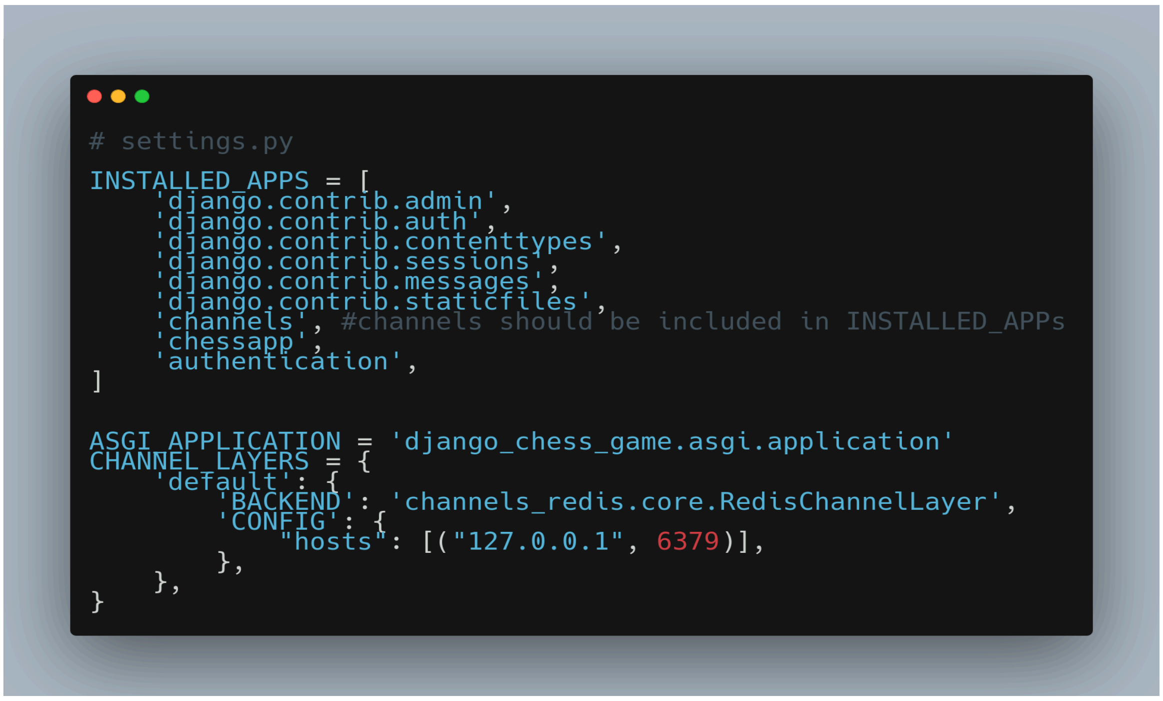
References

[1] "The WebSocket API (WebSockets) - Web APIs," MDN Web Docs. Accessed:

Dec.18,2024. [Online].
Available: https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API
[2] “python-chess: a chess library for Python — python-chess 1.11.0 documentation.” Accessed:Dec.18,2024. [Online]. Available: https://python-chess.readthedocs.io/en/latest/
[3] “django-redis,” PyPI. Accessed: Dec. 18, 2024. [Online].
Available: https://pypi.org/project/django-redis/
[4] “Docs.” Accessed: Dec. 18, 2024. [Online]. Available: https://redis.io/docs/latest/
[5] “Django Channels — Channels 4.2.0 documentation.” Accessed: Dec. 18, 2024. [Online]. Available: https://channels.readthedocs.io/en/latest/
[6] Lichess.org, "The best free online chess game," 2025. [Online]. Available: https://lichess.org/. [Accessed: 28-Jan-2025].
[7] Chess.com, "Play chess online for free," 2025. [Online]. Available: https://www.chess.com/. [Accessed: 28-Jan-2025].
[8] ChessBase, "Playchess.com – the online chess server," 2025. [Online]. Available: https://play.chessbase.com/. [Accessed: 28-Jan-2025].
[9] Chess24, "Play, learn, and watch chess online," 2025. [Online]. Available: https://chess24.com/. [Accessed: 28-Jan-2025].
[10] “Django documentation,” Django Project. Accessed: Dec. 18, 2024. [Online].
Available: https://docs.djangoproject.com/en/5.1/

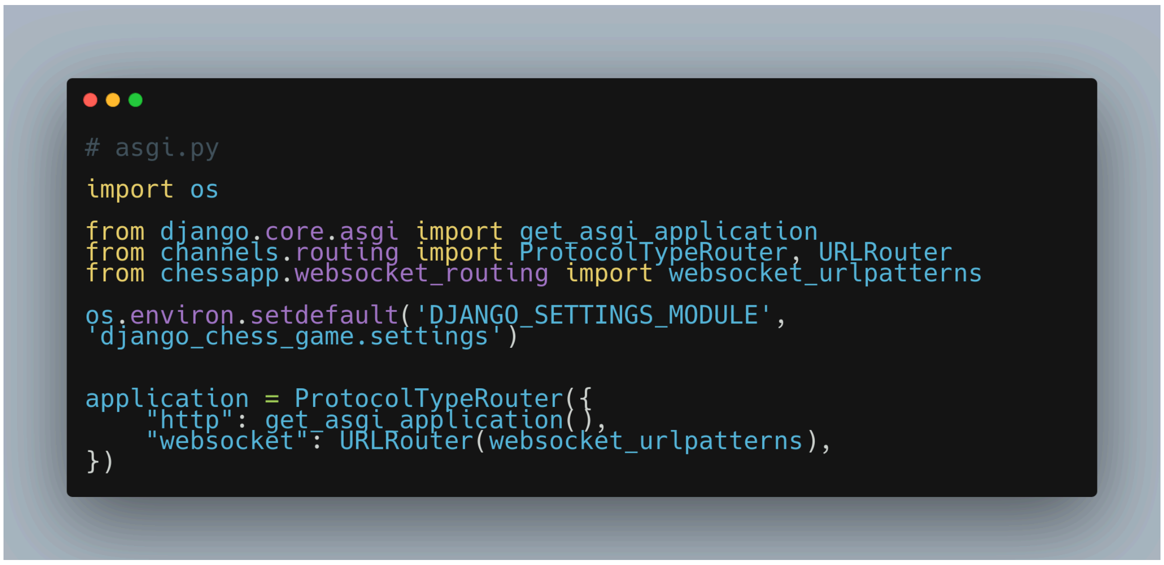
Appendices

AppendixA:Code Snippets



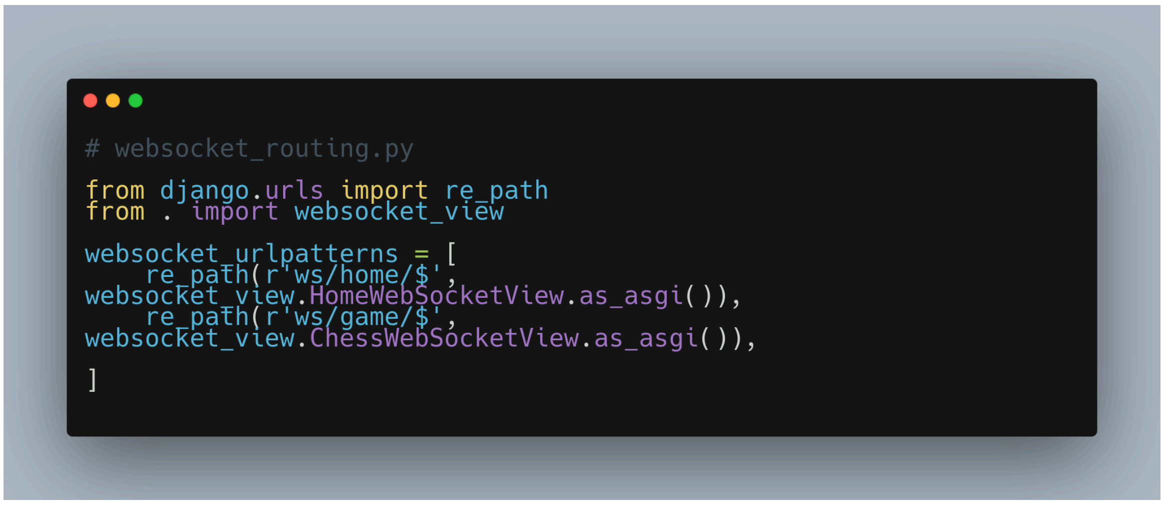
```python
# settings.py
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'channels', #channels should be included in INSTALLED_APPs
    'chessapp',
    'authentication',
]

ASGI_APPLICATION = 'django_chess_game.asgi.application'
CHANNEL_LAYERS = {
    'default': {
        'BACKEND': 'channels_redis.core.RedisChannelLayer',
        'CONFIG': {
            "hosts": [("127.0.0.1", 6379)],
        },
    },
}
```

This appendix demonstrates settings.py configuration for using channels and redis in django project. The hosts part in config defines the url for redis server. Also channels need to be included in settings.py.



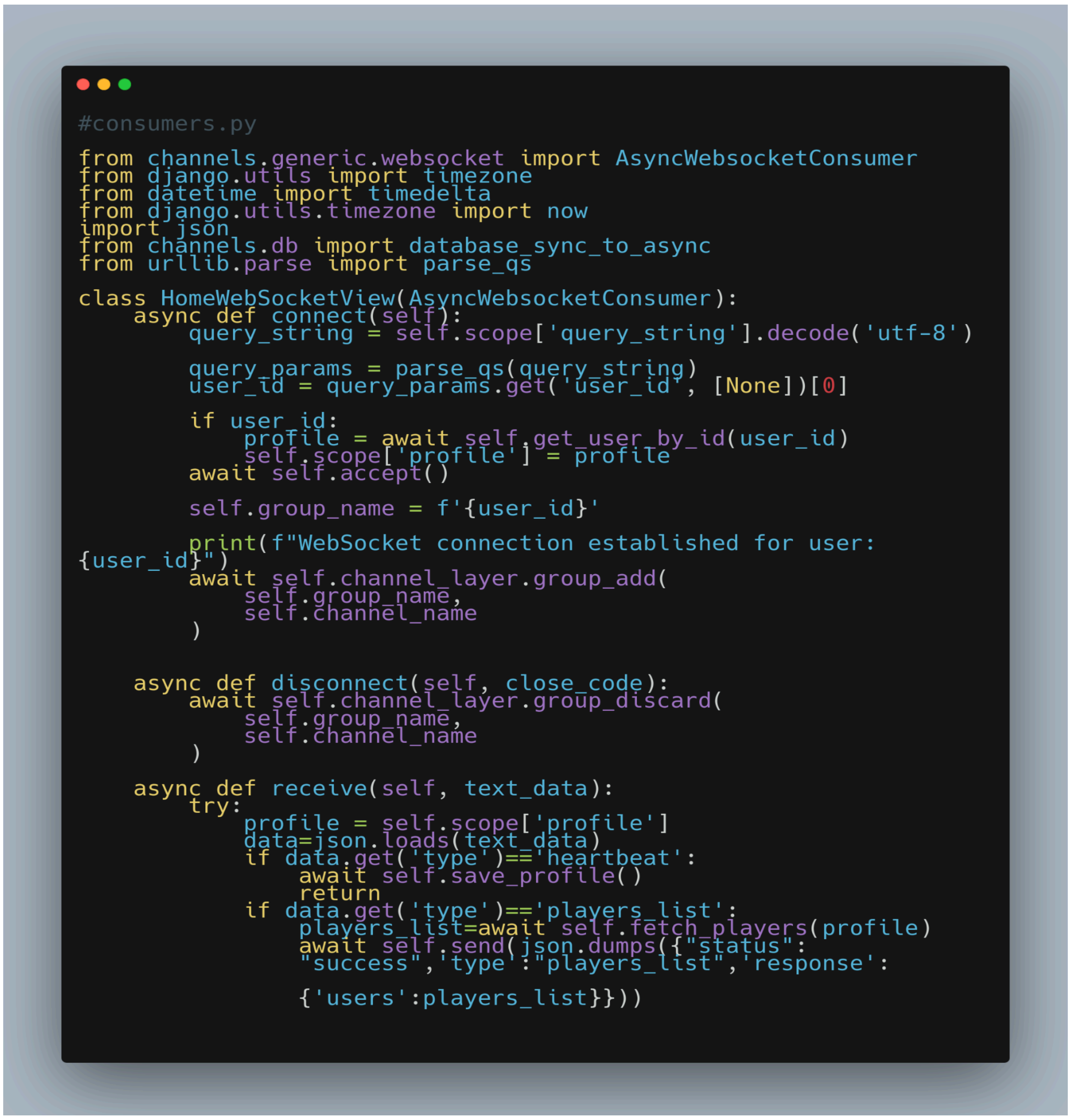
```python
# asgi.py
import os

from django.core.asgi import get_asgi_application
from channels.routing import ProtocolTypeRouter, URLRouter
from chessapp.websocket_routing import websocket_urlpatterns

os.environ.setdefault('DJANGO_SETTINGS_MODULE',
'django_chess_game.settings')

application = ProtocolTypeRouter({
    "http": get_asgi_application(),
    "websocket": URLRouter(websocket_urlpatterns),
})
```

This snippet demonstrates the usage of asgi.py as this needs to configured for websockets.wsgi cannot work with websockets due to the constraint in channels library. So, asgi needs to be configured to route both http and websocket urls.

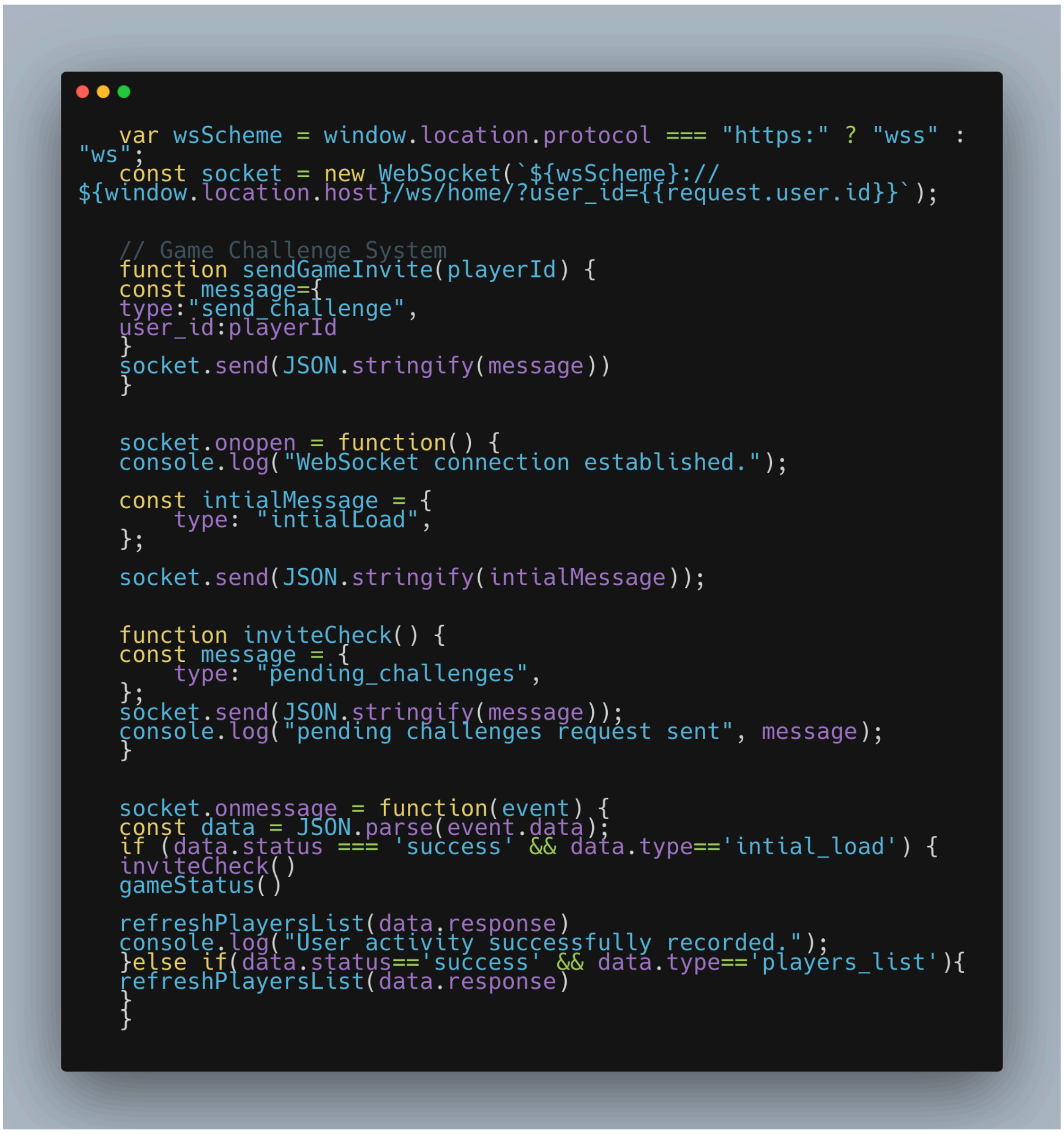


```python
# websocket_routing.py

from django.urls import re_path
from . import websocket_view

websocket_urlpatterns = [
    re_path(r'ws/home/$',
websocket_view.HomeWebSocketView.as_asgi()),
    re_path(r'ws/game/$',
websocket_view.ChessWebSocketView.as_asgi()),
]
```

This snippet represents the websocket urls configuration. It is simply used as routing file for websockets.



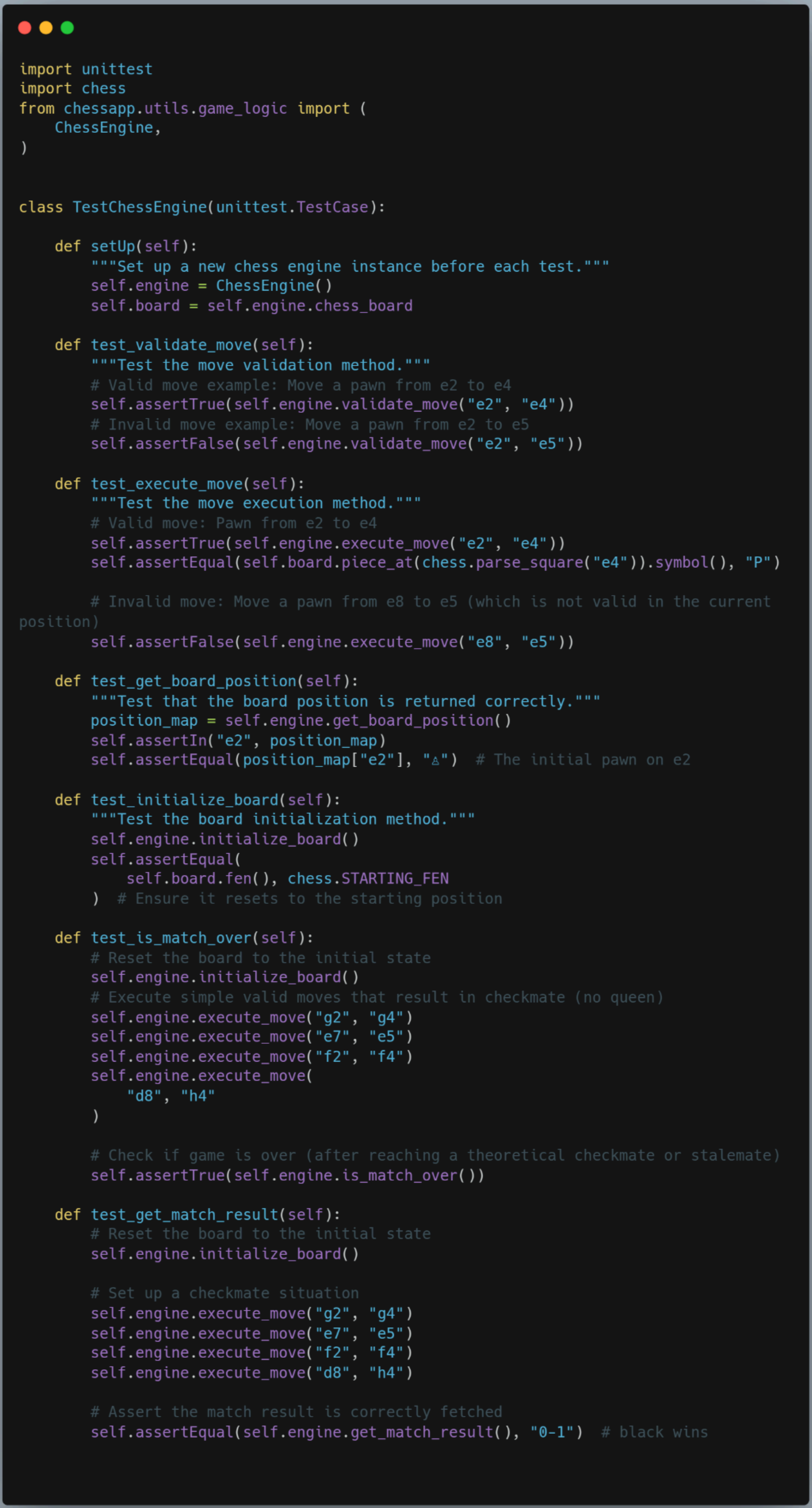
```python
#consumers.py

from channels.generic.websocket import AsyncWebsocketConsumer
from django.utils import timezone
from datetime import timedelta
from django.utils.timezone import now
import json
from channels.db import database_sync_to_async
from urllib.parse import parse_qs

class HomeWebSocketView(AsyncWebsocketConsumer):
    async def connect(self):
        query_string = self.scope['query_string'].decode('utf-8')

        query_params = parse_qs(query_string)
        user_id = query_params.get('user_id', [None])[0]

        if user_id:
            profile = await self.get_user_by_id(user_id)
            self.scope['profile'] = profile
        await self.accept()

        self.group_name = f'{user_id}'

        print(f"WebSocket connection established for user:
{user_id}")
        await self.channel_layer.group_add(
            self.group_name,
            self.channel_name
        )


    async def disconnect(self, close_code):
        await self.channel_layer.group_discard(
            self.group_name,
            self.channel_name
        )

    async def receive(self, text_data):
        try:
            profile = self.scope['profile']
            data=json.loads(text_data)
            if data.get('type')=='heartbeat':
                await self.save_profile()
                return
            if data.get('type')=='players_list':
                players_list=await self.fetch_players(profile)
                await self.send(json.dumps({"status":
                "success",'type':"players_list",'response':

                {'users':players_list}}))
```



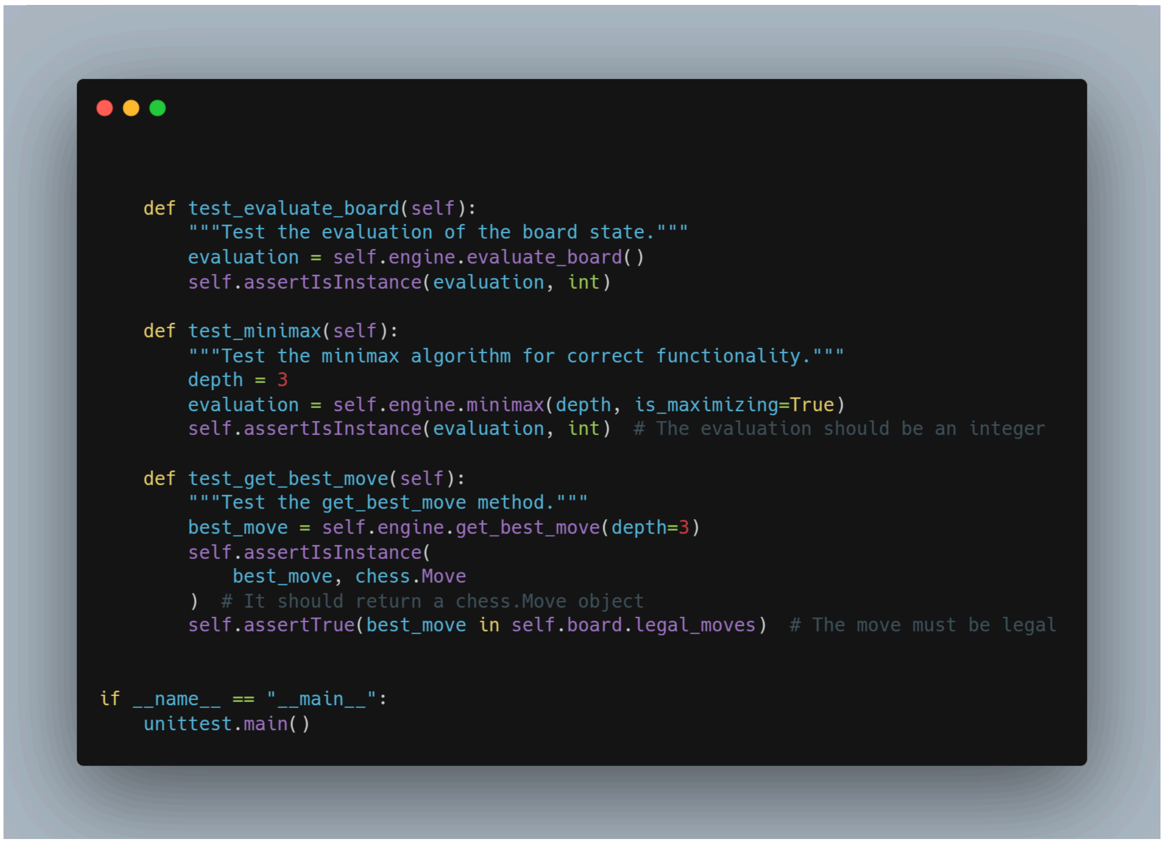
```
    var wsScheme = window.location.protocol === "https:" ? "wss" :
"ws";
    const socket = new WebSocket(`${wsScheme}://
${window.location.host}/ws/home/?user_id={{request.user.id}}`);

    // Game Challenge System
    function sendGameInvite(playerId) {
    const message={
    type:"send_challenge",
    user_id:playerId
    }
    socket.send(JSON.stringify(message))
    }


    socket.onopen = function() {
    console.log("WebSocket connection established.");

    const intialMessage = {
        type: "intialLoad",
    };

    socket.send(JSON.stringify(intialMessage));


    function inviteCheck() {
    const message = {
        type: "pending_challenges",
    };
    socket.send(JSON.stringify(message));
    console.log("pending challenges request sent", message);
    }


    socket.onmessage = function(event) {
    const data = JSON.parse(event.data);
    if (data.status === 'success' && data.type=='intial_load') {
    inviteCheck()
    gameStatus()

    refreshPlayersList(data.response)
    console.log("User activity successfully recorded.");
    }else if(data.status=='success' && data.type=='players_list'){
    refreshPlayersList(data.response)
    }
    }
```

The two snippets defined above represent how websocket works by sending events and messages from frontend to backend.

Websocket is a bidirectional protocol which maintains a persistent connection between two ends. This allows us to send data from one end to another and vice-versa without having to reconnect again and again.

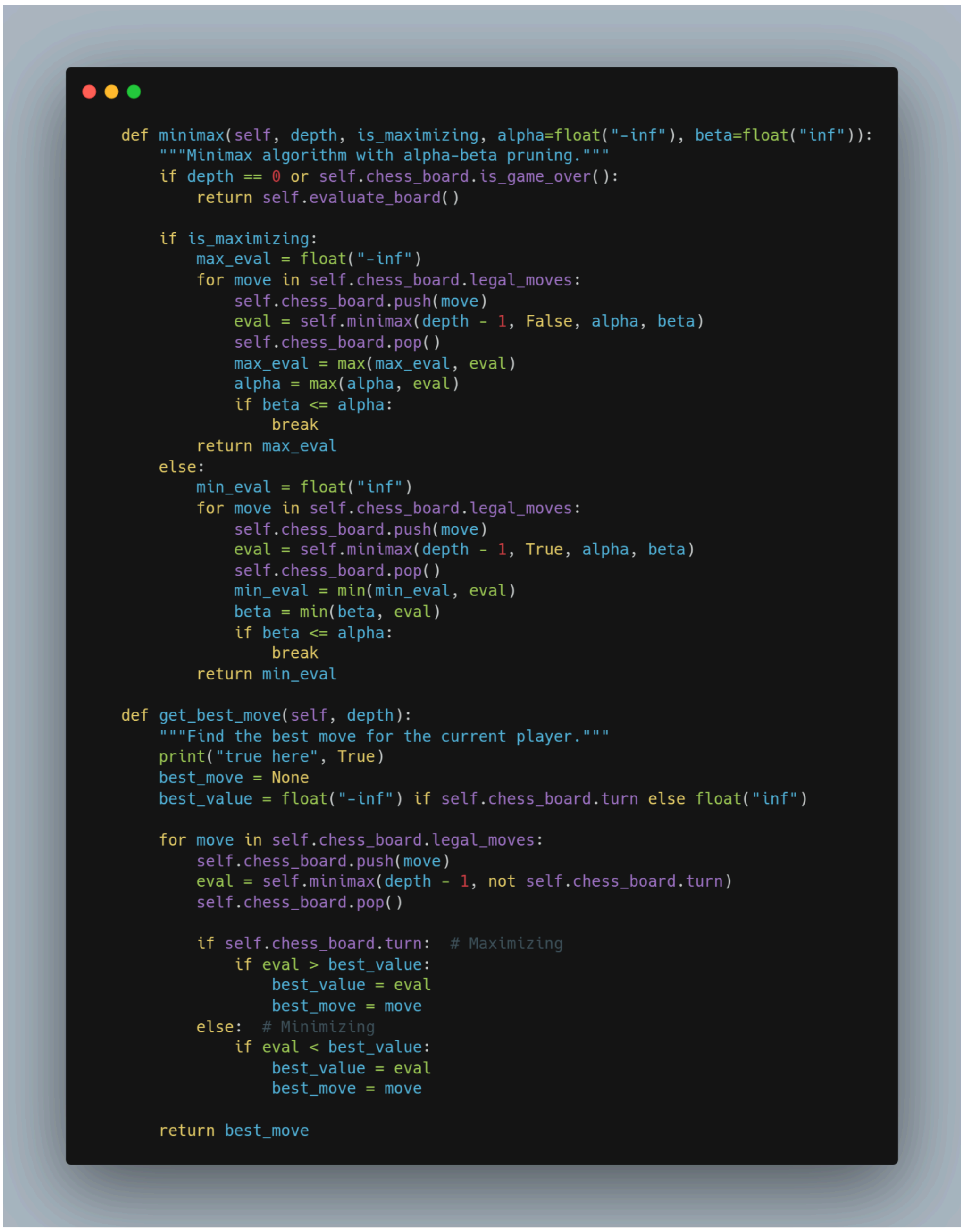


```python
import unittest
import chess
from chessapp.utils.game_logic import (
    ChessEngine,
)


class TestChessEngine(unittest.TestCase):

    def setUp(self):
        """Set up a new chess engine instance before each test."""
        self.engine = ChessEngine()
        self.board = self.engine.chess_board

    def test_validate_move(self):
        """Test the move validation method."""
        # Valid move example: Move a pawn from e2 to e4
        self.assertTrue(self.engine.validate_move("e2", "e4"))
        # Invalid move example: Move a pawn from e2 to e5
        self.assertFalse(self.engine.validate_move("e2", "e5"))

    def test_execute_move(self):
        """Test the move execution method."""
        # Valid move: Pawn from e2 to e4
        self.assertTrue(self.engine.execute_move("e2", "e4"))
        self.assertEqual(self.board.piece_at(chess.parse_square("e4")).symbol(), "P")

        # Invalid move: Move a pawn from e8 to e5 (which is not valid in the current
position)
        self.assertFalse(self.engine.execute_move("e8", "e5"))

    def test_get_board_position(self):
        """Test that the board position is returned correctly."""
        position_map = self.engine.get_board_position()
        self.assertIn("e2", position_map)
        self.assertEqual(position_map["e2"], "♙")  # The initial pawn on e2

    def test_initialize_board(self):
        """Test the board initialization method."""
        self.engine.initialize_board()
        self.assertEqual(
            self.board.fen(), chess.STARTING_FEN
        )  # Ensure it resets to the starting position

    def test_is_match_over(self):
        # Reset the board to the initial state
        self.engine.initialize_board()
        # Execute simple valid moves that result in checkmate (no queen)
        self.engine.execute_move("g2", "g4")
        self.engine.execute_move("e7", "e5")
        self.engine.execute_move("f2", "f4")
        self.engine.execute_move(
            "d8", "h4"
        )

        # Check if game is over (after reaching a theoretical checkmate or stalemate)
        self.assertTrue(self.engine.is_match_over())

    def test_get_match_result(self):
        # Reset the board to the initial state
        self.engine.initialize_board()

        # Set up a checkmate situation
        self.engine.execute_move("g2", "g4")
        self.engine.execute_move("e7", "e5")
        self.engine.execute_move("f2", "f4")
        self.engine.execute_move("d8", "h4")

        # Assert the match result is correctly fetched
        self.assertEqual(self.engine.get_match_result(), "0-1")  # black wins
```



```python
    def test_evaluate_board(self):
        """Test the evaluation of the board state."""
        evaluation = self.engine.evaluate_board()
        self.assertIsInstance(evaluation, int)

    def test_minimax(self):
        """Test the minimax algorithm for correct functionality."""
        depth = 3
        evaluation = self.engine.minimax(depth, is_maximizing=True)
        self.assertIsInstance(evaluation, int)  # The evaluation should be an integer

    def test_get_best_move(self):
        """Test the get_best_move method."""
        best_move = self.engine.get_best_move(depth=3)
        self.assertIsInstance(
            best_move, chess.Move
        )  # It should return a chess.Move object
        self.assertTrue(best_move in self.board.legal_moves)  # The move must be legal


if __name__ == "__main__":
    unittest.main()
```

The above code snippet covers everything related to Unit Testing. The tests are performed for the main game logic mainly for evaluating moves and game status.



```python
def minimax(self, depth, is_maximizing, alpha=float("-inf"), beta=float("inf")):
    """Minimax algorithm with alpha-beta pruning."""
    if depth == 0 or self.chess_board.is_game_over():
        return self.evaluate_board()

    if is_maximizing:
        max_eval = float("-inf")
        for move in self.chess_board.legal_moves:
            self.chess_board.push(move)
            eval = self.minimax(depth - 1, False, alpha, beta)
            self.chess_board.pop()
            max_eval = max(max_eval, eval)
            alpha = max(alpha, eval)
            if beta <= alpha:
                break
        return max_eval
    else:
        min_eval = float("inf")
        for move in self.chess_board.legal_moves:
            self.chess_board.push(move)
            eval = self.minimax(depth - 1, True, alpha, beta)
            self.chess_board.pop()
            min_eval = min(min_eval, eval)
            beta = min(beta, eval)
            if beta <= alpha:
                break
        return min_eval

def get_best_move(self, depth):
    """Find the best move for the current player."""
    print("true here", True)
    best_move = None
    best_value = float("-inf") if self.chess_board.turn else float("inf")

    for move in self.chess_board.legal_moves:
        self.chess_board.push(move)
        eval = self.minimax(depth - 1, not self.chess_board.turn)
        self.chess_board.pop()

        if self.chess_board.turn:  # Maximizing
            if eval > best_value:
                best_value = eval
                best_move = move
        else:  # Minimizing
            if eval < best_value:
                best_value = eval
                best_move = move

    return best_move
```

Above code is for implementing MinMax algorithm with alpha beta pruning to get the best move based on certain depth. The depth used in our code is 3.