SP-14 Blue Chess AI

Final Report

CS 4850-01–Spring 2025

Professor Perry

04/20/25

Jake Strunk Team Leader Documentation	Garrett Cochran Developer	Oliver Fuxell Developer	Avery Kennedy Documentation	Ethan Shockey Documentation

https://studentweb.kennesaw.edu/~ofuxell/chessai

https://github.com/GOLAJ-Enterprises/SP-14-Blue-Chess-AI

STATS AND STATUS		
LOC	2,239	
Components/Tools	Flask, PyTorch, PyInstaller	
Hours Estimate	375	
Hours Actual	520	

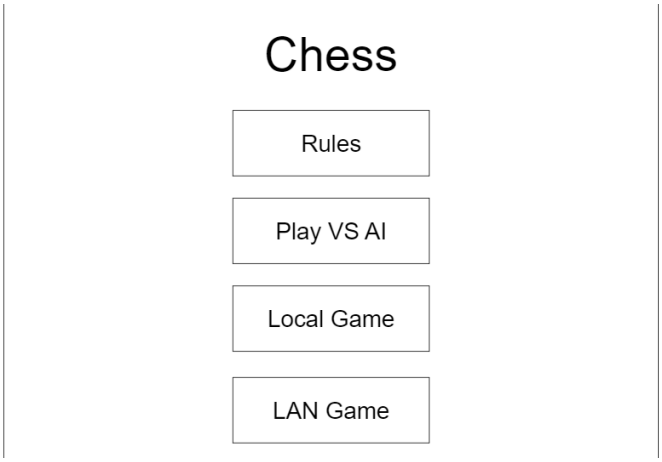
1.0 Project Overview / Abstract (Research)

Chess has been a large part of human culture since its creation in the 6th century. But even with its popularity it is a very complex and difficult game, and we want to make that barrier of entry a little easier. We seek out to create a chess platform in which new players can learn how to play the game against a chess AI and to give people who want to play against others a platform to do so.

We will do this by creating a fully setup rendered chess boards with each piece following their own respected rules to ensure no invalid movements with gameplay features such as pawn promotion, castling, and games checks. The user will be able to play against the chess AI, other players on the same machine, or on another machine.

2.0 Functional Requirements

2.1 Start menu



The program must open to the start menu.

The menu must offer a “rules” page to display the basics of how to play the game in one or two basic text pages.

The menu must offer player-versus-player or player-versus-computer mode. Upon selecting a game mode, a menu must prompt the player to pick white/black pieces or pick randomly.

2.2 Display the Chess Board

The game must display an 8x8 chess board.

The game must clearly display whether pieces are white or black.

The pieces must be displayed on the correct tiles.

The game must indicate whose turn it is.

The game must update after every move.

The game must detect when a king is in checkmate and display the winner.

The game must detect a stalemate and display that a stalemate has occurred.

2.3 Controlling the game

Both players (including the computer) take turns moving pieces.

White pieces must move first.

The game cannot allow piece movements.

The game must prevent moves that place the turn player's king in check.

Pieces, except for knights, cannot move to a square if there is a piece directly between the starting square and the destination.

The player must be able to resign.

Pieces must capture by moving onto a square occupied by an opponent's piece.

2.4 Piece Movement and Attacks

Pawns

Pawns must only move one space forward along their file. If the pawn has not moved yet, it must also be permitted to move two spaces forward. Pawns must be able to capture diagonally forward. White pawns capture pieces on higher ranks, and black pawns capture pieces on lower ranks. Pawns must be able to capture pawns en passant. If pawn A advances two spaces and lands adjacent to pawn B (along the same rank), then pawn B can capture pawn A as though pawn A had advanced one space. Pawns must be able to promote into a different piece if they advance to the other end of the chess board.

Bishops

Bishops must only move and capture any distance diagonally.

Knights

Knights must move two spaces horizontally and one space vertically. Alternatively, it can move two spaces vertically and one space horizontally. Knights, uniquely, do not have their moves blocked if there is a piece between the starting tile and a destination tile. Landing on a piece in this way will capture that piece.

Rooks

Rooks must only move and/or capture any distance along their rank or file. Castling is only permitted if the rook has not moved yet. A player must be able to castle if their king and the appropriate rook have not moved yet and the squares between the two pieces are empty. Castling must not be permitted if the king would leave, pass through, or land on a threatened square. Castling must move the king two spaces towards the rook and place the rook on the space the king moved over.

Queens

Queens must only move and capture any distance diagonally or to spaces along their rank or file

Kings

Kings must only move one tile in any direction (including diagonally) unless castling. Players must not be able to make moves that leave their king in check. A king being checkmated must end the game with the checkmated player's loss. Kings must only be able to castle if the king has not moved yet.

Castling

This refers to a move in which a player can move their king and rook simultaneously in a particular fashion. It cannot occur if either the players king or rook has moved, if the king is

in or will pass through “check” (see below), or if there are any pieces between the king and rook. When a player “castles”, they move their king two spaces to the left or right, and then the rook who is currently in the direction the king moved will automatically move next to the king.

En Passant

En passant is a special pawn move that refers to the term “in passing” in French. This occurs when the capturing pawn has moved exactly three ranks, the captured pawn must have moved two squares in one move and being located next to the capturing pawn and must be performed on the following turn that the captured pawn moves.

2.5 Game End Conditions

The game should detect when there are conditions that put a player in check, and it should block the player from doing those actions while displaying an explanation. The game should further detect when a king is in checkmate and give the opposing player victory. Both players should also be allowed to forfeit.

2.6 AI Requirements

At a minimum, the AI must be competent enough at the game to move the pieces around the board correctly and put up resistance to the player’s attacks. Ideally, the AI will be decent enough at chess to predict and counter player strategies and implement some of its own to win the game. The AI also should not tank performance of the application during its operation.

2.7 Monetization

Since our program is web-based, we initially planned to monetize the project using Google AdSense. However, due to time constraints, we were unable to implement this feature before the project deadline. To qualify for AdSense, the website needs to be active for approximately two months and must include key pages such as a privacy policy, terms of service, an "About Us" page, and a contact page. The process involves signing up for Google AdSense and submitting the website for approval, which typically takes around two weeks. Once approved, Google provides a snippet of AdSense code that must be inserted into the <head> or <body> tags of the website. After implementation, Google automatically begins displaying ads that are tailored to the site's content and the user's browsing behavior.

3.0 Non-Functional Requirements

3.1 Usability

It is very important that our game be intuitive to play. Since we cannot simply assume that our users are extremely familiar with chess, it is imperative that understanding how to play is not an obstacle. The gameplay experience should be designed with this in mind. Any other menus should be simple to navigate as well.

3.2 Reliability

Reliability concerns are also hugely important to our development. The app will need to work consistently in all aspects of gameplay to make playing chess on our app enjoyable. Achieving multiplayer will require attention to detail to ensure that the multiplayer experience runs smoothly.

3.3 Interoperability

Achieving LAN multiplayer will require the two systems running the application to be able to share data. The two systems are going to need to share the same board with each player and update the board accordingly. Interoperability will need to be a requirement in order to get LAN multiplayer implemented. We dropped this requirement due to time constraints.

4.0 Design Considerations

4.1 Assumptions and Dependencies

4.1.1 Compatibility:

As a web-based game, our program will be compatible with any operating system. In terms of computer hardware, the requirements will be minimal.

4.1.2 End-User:

We will assume that our users will have basic computer literacy (i.e, comfortability using a mouse and keyboard, launching apps and software, etc.). Users must have a stable internet connection to play the game. Basic chess knowledge will be helpful but not required as the user can refer to the rules tab within the application.

4.2 General Constraints

A big constraint is that without a dedicated host made through port forwarding or paying for one, if two players want to play against each other on separate machines, they must be on the same network using a LAN connection which we did not have time to implement.

4.2.1Environment

The environment we started off with was PyCharm. PyCharm is an IDE developed by JetBrains that was built off their initial IntelliJ platform. We will use Python to code the project. This environment was successful with developing the board and pieces until we ran into issues when developing the GUI. We thought it was best to further develop our code using flask and torch covering the front and back end.

4.2.2 End-user environment

The end user environment should be intuitive and very easy to use and understand. Our users are not expected to have technical knowledge to play our game, so the UI and operation of it should reflect this.

4.2.4 Memory and other capacity limitations

Our chess game should not significantly utilize a computer's disk for memory storage.

4.2.5 Performance requirements

Performance requirements should be low to be able to run this application. The game should require at least a dual-core processor, an up-to-date graphics card and operating system, and 4-8 gigabytes of RAM.

4.2.7 Reliability

Reliability concerns are also hugely important to our development. The app will need to work consistently in all aspects of gameplay to make playing chess on our app enjoyable.

5.0 Development Methods

We will utilize the Software Development Life Cycle (SDLC) methodology to develop this application. This includes completing relevant documentation to track and evaluate our progress throughout the planning, design, development and testing of our game.

5.1 AI Training Methodology

We will start by collecting data from a chess match dataset, extracting board states and evaluations. The data will then be preprocessed by converting each board state into an 8x8 grid with encoded pieces and extracting the move evaluations.

Next, we will design a CNN using PyTorch or Tensorflow to process the grid and output probabilities for possible moves. The model will be trained on the dataset using supervised learning and cross-entropy loss. The model's accuracy will be evaluated by testing on unlabeled data. If needed, hyperparameters will be fine-tuned.

Once the model is trained, we will use self-play to generate new training data and refine it through reinforcement learning. To adjust the difficulty levels, the output probabilities will be modified using temperature-based randomness, controlling the level of exploration and selecting moves based on a specific skill level. This will be similar to playing at different ELO levels in chess.

Finally, we will integrate the trained model into our chess application for real-time gameplay.

5.2 Architectural Strategies

In general, we will utilize an Object-Oriented Programing (OOP) approach to developing this application. Much of the system architecture, therefore, will be based on manipulating inheritance and parameter passing between objects and interfaces and any methods contained within. Objects may include the menus, the game board, individual pieces and/or lanes on the board, and other more technical objects such as handlers for the UI connection. Object-oriented programing is useful for this project as it will allow us to take advantage of a few key benefits, listed below.

5.3 Better Collaboration

We coded this project in the Python language, specifically to take advantage of the PyCharm IDE's 'Code With Me' plugin to enable better collaboration. Most of our team is also familiar with python and so the language was a natural choice.

5.4 Efficient Code Re-use

We can now make more efficient repeated use of some portions of code, primarily through inheritance or abstraction. The main places in which code may be reused will be in movement of pieces and changing screens, as these pieces of code are bound to have some similarities.

5.5 Potential Extension

OOP allows us to consider future extensions to the project, including special rule changes and other changes to the game easily, as all we will need to do is change the object's properties and methods.

5.6 Memory Management

OOP enables a much easier logical organization of memory. Each object will take up a clear space of memory based on its properties and methods. It will be much easier to track and manage memory with this infrastructure in mind.

5.7 Communication Mechanisms

OOP allows for simple parameter passing via constructors and methods for objects, giving us the ability to quickly communicate pieces of data between different sections of the architecture.

5.8 Version Control

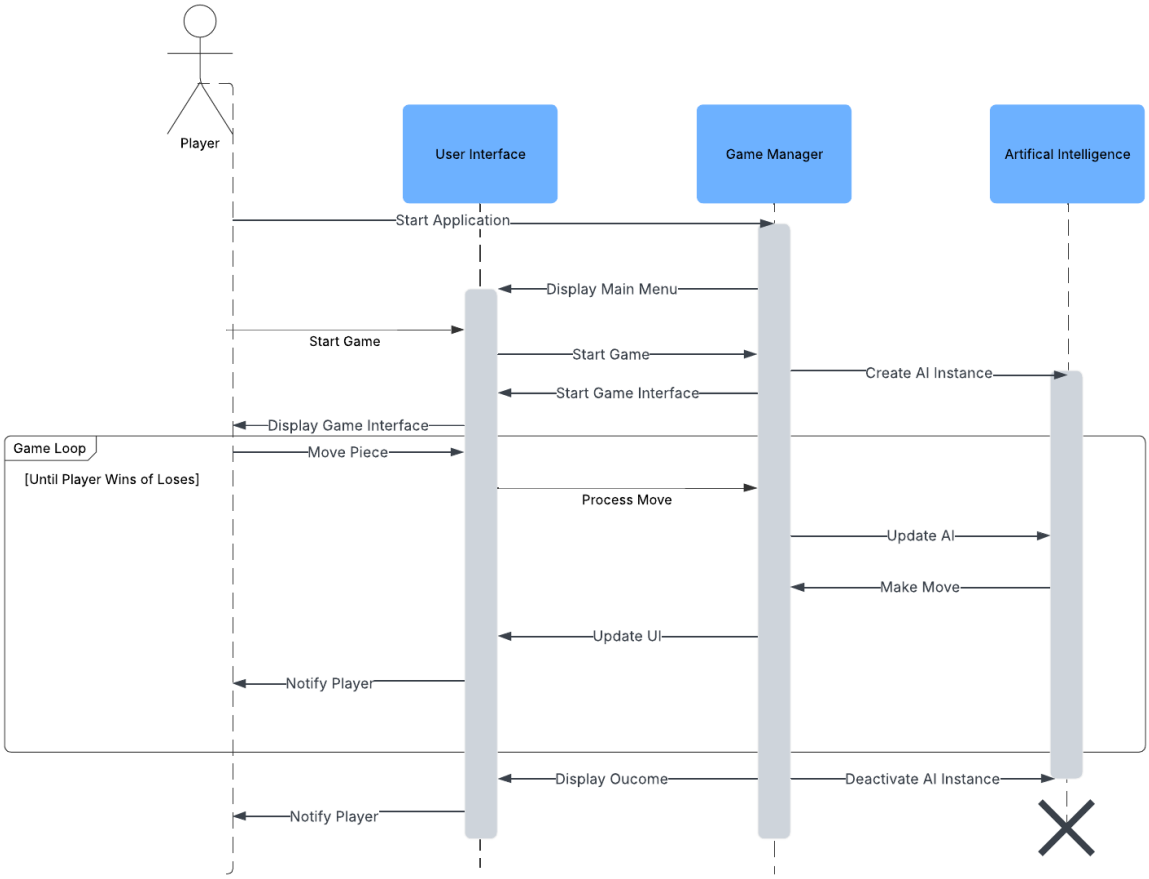
GitHub was used throughout the development of the SP-14 Blue Chess AI project to manage version control and track the evolution of the system. All work was organized under the main branch, with regular commits documenting each phase of the project's buildout. Early commits focused on establishing the basic project structure, setting up the Flask web server, and developing the initial game interface with HTML, CSS, and JavaScript. As development continued, new features were gradually introduced, such as implementing a real-time communication system using Flask, integrating the bitboard-based backend for chess move management, and connecting the neural network model to drive the AI opponent's decision-making process through Monte Carlo Tree Search. The commit messages were kept clear and descriptive, making it easy to follow the logic behind each change as the project expanded.

Later stages of the project captured more advanced updates, including creating a standalone executable version using PyInstaller. Commit activity remained steady throughout the development timeline, reflecting a workflow where features were built, tested, and refined step-by-step. Rather than using multiple branches, all updates were made directly to the main branch, allowing for a straightforward, linear development history. This approach made it easy to trace the introduction of key components like neural network integration, server optimizations, and UI improvements leading up to the final, polished version. Overall, version control played a critical role in organizing the project, keeping development progress transparent, and ensuring that each feature was cleanly added to the final product.

6.0 System Architecture

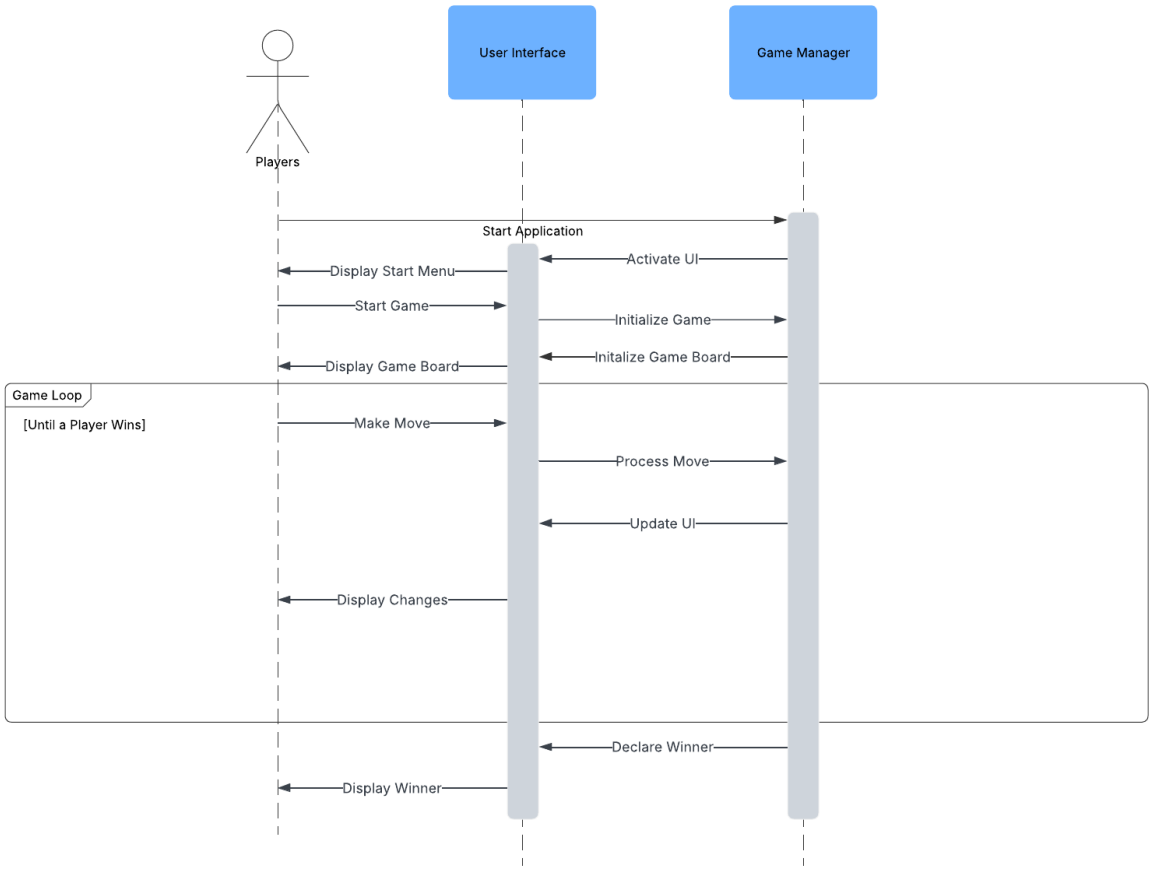
The application for our chess game consists of 3 main components. These are: the User Interface, the Artificial Intelligence, and the Game Manager. Below, we will provide a few sequence diagrams to demonstrate how the system's different components will work together to provide the best chess experience.

6.1 Player VS AI



The principal game-mode of our application, Player vs AI, will consist of 3 main components that respond to user input. The first to start is the game manager, which handles the game's computations and manages the actions the player and AI take in the game on a technical level. The application software also manages both the UI and the AI starting up and their interactions with each other. The user interface will display and take in user input both on the start menu, and on the interactive chess board itself. The AI, meanwhile, is responsible for reacting to the player's moves in-game and making moves to contest the player at a suitable level of difficulty.

6.2 Player vs Player Local



Another game-mode will be Player vs Player Local. This is when two human players can play against each other on the same machine. In this mode, we only need to be concerned with the user interface, which handles the visual changes, and the game manager which will handle the changes within the game application.

7.0 Detailed System Design

7.1 Game Manager

7.1.1 Classification

The game manager is a subsystem of the application.

7.1.2 Definition

The game manager serves as the manager for our chess application. It will handle the computational tasks to keep the application running and manage the moves players and AIs make on a digital level.

7.1.3 Constraints

The game manager will need to be designed in multiple clear-cut modules. This way multiple different classes and sub-sections of the game manager can be separate but connected to ensure game integrity and logical clarity. This will include a module for managing the game board and general play, a module for calculating if a victory has been achieved, and two modules for managing the UI and AI activation and communication.

7.1.4 Resources

The game manager will utilize processors and RAM on any machine running this application. The manager will also make use of a small amount of local memory on the player's machine. We may make use of certain software libraries in this layer, either to perform computations relevant to the game itself.

7.1.5 Interface/Exports

The game manager is the component responsible for managing the game state and any and all changes made during the course of play. It should provide positional information, in the form of coordinates, of all pieces both for internal use in calculating attacks and possible moves, and to export this information to the user interface for placement on the visual board, or to the AI for it to calculate the next move it will make to contest the player. The game manager will also provide the current valid moves and attacks for each piece on the board, again for the same use by the UI and AI. The manager will also be responsible for most of the error handling in our application.

7.2 User Interface (UI)

7.2.1 Classification

The user interface is a subsystem of the application

7.2.2 Definition

The user interface will provide an easy-to-use set of screens for the players to interact with and play the game.

7.2.3 Constraints

The UI should be as simple to understand as possible, as our users should not be required to have beyond basic computer literacy. To this end, most if not all UI elements should be accessible simply via clicking or clicking and dragging with a mouse.

7.2.4 Resources

The UI will need to make use of the user's monitor and graphics card to display the menus and interactive chess board. It will also need to be able to take in user input from devices, including the mouse or trac-pad, the keyboard, and potentially the touchscreen (if it is integrated with mouse-like functionality). Some memory space will need to be used by the UI to store the sprites and display elements utilized by the UI to display the current game state, pieces, and the rules and main menus.

7.2.5 Interface/Exports

The UI takes in all the information provided by the user in this application though menus and an interactive chessboard. It will relay menu navigation to the game manager to handle the initialization of other menus and the interactive chessboard. During play, the UI will signal to the game manager to change the coordinate positions of various pieces, along with coordinates for their potential lanes of attack and acceptable movement.

7.3 Artificial Intelligence (AI)

7.3.1 Classification

The AI will be a subsystem of our application

7.3.2 Definition

The AI will be responsible for reacting to player actions in the Player vs AI game mode, functioning as a computer version of another human player. It will attempt to counter the player's strategies, or at the very least oppose the player, attempting to win the game for itself.

7.3.3 Constraints

At a minimum, the AI must be competent enough at the game to move the pieces around the board correctly and put up resistance to the player's attacks. Ideally, the AI will be decent enough at chess to predict and counter player strategies and implement some of its own to win the game. The AI should not be a chess master, however, and should either be beatable by an average chess player or have a difficulty setting. Striking this balance will be the main constraint on AI development. The AI also should not tank performance of the application during its operation.

7.3.4 Resources

The AI will need to use CPU power and RAM to perform calculations and will need a small amount of memory storage space to keep saved responses to user actions.

7.3.4 Interface/Exports

The AI will make moves on the chess board like a human player, which will then be processed by the move handling portion of the game manager and displayed onscreen by the UI, which will notify the player that the AI has moved, and it is their turn.

7.4 Development Notes

7.4.1 Chessboard

We will keep utilizing the traditional 8x8 board for our chess game, including all the traditional rules.

7.4.2 Chess game engine

The engine was made from scratch and it utilizes native python. It generated the board, pieces, colors, and the checkerboard look just like every normal chess board. The visual aspect was incorporated to have a basic chess board feel. The game stats is also display on the game engines screen including castling rights, en passant, halfmove count, full move count, and what colors turn it is.

7.4.3 AI Development

We designed a CNN using PyTorch to process the grid and output probabilities for possible moves. The model was trained on the dataset using supervised learning and cross-entropy

loss. The model's accuracy will be evaluated by testing on unlabeled data. If needed, hyperparameters will be fine-tuned.

The AI was trained and hooked up with Monte Carlo Tree search creating efficient moves from the AI in the game play we messed around with. The percentage rate has come a long way and developed to have better moves as we messed around with training different models.

7.4.3.1 Min max vs CNN

Convolutional Neural Networks (CNN) is a deep learning model designed for pattern recognition. In our chess AI, CNN processes board positions as 8x8 grids, learning patterns from previously played or watched games to predict the best possible move. Unlike traditional based methods such as min max, CNNs do not rely on predefined heuristics but instead learn from experiences making it highly effective in positional play and long term-strategy. There are two methods that we could use to train the AI, one being through supervised learning where we will expose the AI to expert game sets and the other being reinforced learning where the AI learns from directly playing itself.

We decided to use CNN over min max because CNNs learn from experience, recognizing deep positional and strategic patterns that traditional rule-based methods struggle with. Unlike min max, which relies on heuristics and searches through an exponentially growing game tree, CNNs can generalize from data, making them highly effective for long-term planning and complex positions. Additionally, when combined with Monte Carlo Tree Search, CNNs can provide more efficient and adaptive decision-making, reducing brute force calculations while improving move accuracy.

7.4.3.2 Residual Neural Networks

Residual Neural Networks (ResNets) are essential in modern chess AIs because they allow neural networks to lean deeply and effectively. They interpret chess positions as grids, recognizing patterns to suggest strong moves and evaluate positions without relying on manually programmed rules. Through reinforcement learning and self-play, ResNets continuously improve their understanding of chess strategy

7.5 Database Connection

For the implementation of our chess AI, we'll utilize the Lichess Chess Dataset, a comprehensive collection of chess games publicly available. We'll begin by downloading this dataset locally, ensuring a controlled and secure environment for our data processing. Once downloaded we'll leverage Pythons Pandas library to preprocess and clean the data. After preprocessing, we'll transform this data into tensors compatible with PyTorch. PyTorch will be used to build and train our Convolutional Neural Network model. All

training will occur entirely on local hardware, without any external database connections, enhancing data security, privacy, and accessibility. This approach will allow us to maintain full control over the dataset, preprocessing steps, and model training parameters.

7.6 Steps to Set Project Up

To get the program running, start by installing Python 3.11, as the project is built specifically for this version. Once Python 3.11 is installed, open the project directory and ensure you are using the correct Python version. Then, install the Poetry package manager by running `pip install poetry`. With Poetry installed, navigate to the project folder and run `poetry install` to create a virtual environment and automatically install all required dependencies. After the setup completes, you can run the program by executing `poetry run python run.py`. This will start the application using the configured environment.

8.0 Testing Methodologies

8.1 Playtesting

Blind Playtesting

Playtesting, particularly blind testing, involved allowing users with no prior involvement in the development of the chess AI to interact with the application. This was essential to simulate real-world usage and gather unbiased feedback. Testers were not briefed on the AI's internal mechanics, UI design decisions, or specific features; instead, they were simply asked to use the application to play chess games. This helped us identify usability issues, confusing interactions, and potential bugs that internal developers may have overlooked. Feedback from blind testing sessions guided improvements in the user interface, ensuring a smoother and more intuitive player experience.

8.2 Stress Testing

Stress testing focused on evaluating the performance and reliability of the chess interface under high load or edge-case scenarios. We tested rapid sequences of moves, repeated redo actions, and excessive use of game reset or restart functions to see if the application could remain stable. These tests helped us identify and resolve UI freezes, unexpected behavior due to invalid state transitions, and other stability-related issues that could arise during prolonged use.

8.3 In-Code Tests

At the current stage, in-code testing was used to verify the correctness of fundamental game mechanics and UI logic. We implemented basic unit tests and runtime assertions for essential components such as piece movement, rule enforcement (e.g., turn-taking, legal move validation), and board state updates. These checks ensured that the system behaved as expected during move input, piece captures, and board resets.

9.0 Software Test Chart

Function to Test	Testing Method	Pass/Fail	Severity (of failure)
Pawn Movement/Attacks	In-Code Test	Pass	N/A
Bishop Movement/Attacks	In-Code Test	Pass	N/A
Rook Movement/Attacks	In-Code Test	Pass	N/A
Knight Movement/Attacks	In-Code Test	Pass	N/A
Queen Movement/Attacks	In-Code Test	Pass	N/A
King Movement/Attacks	In-Code Test	Pass	N/A
Game state Tracking	Playtest/In-code tests	Pass	N/A
Game board display	Playtest	Pass	N/A
Special Movement (En Passant, Castling)	In-Code Test	Pass	N/A
Check/Checkmate Detection	Playtest/In-code tests	Pass	N/A
Usability	Blind Playtest	Pass	N/A
Reliability	Stress-testing	Pass	N/A

AI Competency	Rigorous In-code testing prior to full integration	Pass	N/A
AI Difficulty	Rigorous Playtesting	*not implemented	N/A
AI Optimization	Playtesting	Pass	N/A
Player Ranking/Stats	Playtesting	*not implemented	N/A
WIn/Lose Ratio	Playtesting	*not implemented	N/A

10.0 Conclusion

We successfully deliver a functional, engaging, and educational chess-playing platform that caters to both new and experienced players. Through a combination of traditional rule enforcement, intuitive UI design, and a responsive AI opponent trained using convolutional neural networks and Monte Carlo Tree Search, we created an application that replicates the strategic depth of chess while lowering the barrier for entry. Our use of object-oriented design and collaborative development tools enabled us to build a modular, scalable system with a solid foundation for future feature additions, such as online multiplayer and AI difficulty scaling. While time constraints limited the implementation of certain aspects—such as web monetization and full statistical tracking—the core goals of usability, reliability, and AI integration were successfully met. With additional development, this application could become a competitive platform for chess education, entertainment, and competition across broader user bases.