



Wentworth Institute of Technology

COMP4960 – Software Engineering

Instructor: Dr. Koorosh Firouzbakht

Project Final Report

for

Arcane Chess

Version 1.1

Prepared By

Team: Next Gen Coders, Team 04

David Costa, costad3@wit.edu

Ibukunoluwa Folajimi, folajimii@wit.edu

Bryce Parkinson, parkinsonb@wit.edu

Nathaly Phrasavath, phrasavathn@wit.edu

GitHub Link:

<https://github.com/costad3atwit/SoftwareEngineering>

11/13/25

Executive Summary

Arcane Chess is a web-hosted application that reimagines the classic game of chess by introducing a strategic card system and unique chess pieces, creating a dynamic blend of traditional chess mechanics and innovative gameplay elements. The application successfully combines familiar chess rules with transformative abilities that fundamentally alter how the game is played.

Upon visiting the site and logging in, players select their custom deck of 16 cards from a pool of 32 available cards before entering matchmaking or connecting directly with another player via unique player IDs. Once matched, gameplay follows standard chess conventions like moving pieces, capturing opponents, promoting pawns, and delivering checks while adding a critical strategic layer through the card system. Players can deploy cards to modify pieces, manipulate turn order, alter the board's state itself, or affect their opponent's card hand, creating countless tactical possibilities beyond traditional chess.

The game features seven specialized pieces accessed through card play: Peons (non-promoting pawns with special abilities), Headhunters (transformed knights with extended forward attack range), Clerics (protective bishops that sacrifice themselves to save nearby pieces), Warlocks (teleporting bishops with same-colored tile movement), Witches (knights that summon peons on marked capture locations), Scouts (long-range reconnaissance pawns that mark enemies), and the formidable Dark Lord (a vampiric queen capable of converting enemy pieces). Each transformation fundamentally changes tactical options and requires careful strategic planning.

The application was developed using a modern web stack: Python powers backend game logic and state management, Node.js handles the server infrastructure, and HTML/CSS provides the user interface. Real-time multiplayer functionality is achieved through WebSocket connections, enabling seamless synchronization between players. The application is deployed on Render.com's free tier, providing public accessibility at a static URL.

The team successfully delivered a fully functional chess variant that maintains the strategic depth of traditional chess while introducing innovative mechanics through the card system and unique pieces. The application demonstrates robust game state management, real-time multiplayer capabilities, and a brief tutorial system to onboard new players. All core features like matchmaking, deck building, standard chess rules, and card effects are operational and accessible through an intuitive web interface.

In addition to the gameplay features, the project incorporates a well-structured client-server architecture, where the browser-based frontend communicates with a Python backend through REST and WebSocket connections to maintain synchronized game states and matchmaking. Testing supported overall stability through unit tests, multiplayer integration checks, and manual UI testing for deck building and in-game interactions. The team met all major functional and non-functional requirements, including real-time multiplayer support, an intuitive interface, correct chess rule enforcement, and full implementation of the card system and unique piece mechanics, resulting in a robust and engaging final product.

1. Contents

| | |
|--|----|
| Executive Summary | 1 |
| 1. Contents..... | 2 |
| 2. Revision History | 3 |
| 3. Introduction | 4 |
| 3.1. Document purpose | 5 |
| 3.2. Product overview..... | 5 |
| 3.2.1 Problem statements | 5 |
| 3.2.2 Proposed solution..... | 5 |
| 3.2.3 Novelty..... | 5 |
| 3.3. Product functionality | 5 |
| 3.4. Definitions..... | 6 |
| 3.5. Acronyms and abbreviations | 6 |
| 4. System requirements..... | 7 |
| 4.1. Functional requirements | 7 |
| 4.2. Non-functional requirements | 9 |
| 4.3. Other requirements | 10 |
| 5. System architecture..... | 10 |
| 5.1. Overall architecture | 10 |
| 5.2. Components mapping | 11 |
| 5.3. Technology stack selection..... | 12 |
| 6. System Design..... | 13 |
| 6.1. UI..... | 13 |
| 6.2. Class diagram | 15 |
| 6.3. Sequence/activity diagram | 15 |
| 6.4. Database | 16 |
| 7. Others | 16 |
| 7.1 Algorithms..... | 17 |
| 7.2 Third-Party Libraries and APIs..... | 17 |
| 8. Test plan..... | 17 |
| 8.1. Game Interface and Player Interaction | 17 |
| 8.2. Matchmaking and Deck Building | 18 |
| 8.3. Game State and Turn Logic..... | 19 |

| | |
|--|----|
| 8.4. Game End and Replay Options | 20 |
| 8.5. UI Display Elements | 20 |
| 8.6. System Performance | 21 |
| 8.7. Accessibility and Compatibility | 21 |
| 8.8. Board Class | 22 |
| 8.9. Piece Class | 23 |
| 8.10. Deck Class | 23 |
| 8.11. Hand Class | 23 |
| 8.12. Move Class | 24 |
| 8.13. All Piece Classes | 24 |
| 8.14. Game Manager | 25 |
| 8.15. Game State | 25 |
| 8.16. WebSocket Communications | 25 |
| 8.17. Coordinate System | 26 |
| 8.18. Player Management | 26 |
| 9. Delivery Metrics | 27 |
| 10. Release Summary | 27 |
| 11. Performance Metrics | 28 |
| 12. Defect Management and Known Issues | 29 |
| 13. Test Summary | 29 |
| 14. Customer/Stakeholder Feedback | 30 |
| 15. Lessons Learned, Recommendations and Future Work | 30 |
| 16. Conclusions | 30 |
| 17. References | 31 |

2. Revision History

| Date | Version | Description | Author(s) |
|------------|---------|---|---|
| 11/13/2025 | 1.1 | Final Report Created and first iteration | Nathaly Phrasavath, David Costa, Bryce Parkinson, Ibukunoluwa Folajimi |
| 11/20/2025 | 1.2 | Final Report V2 | Nathaly Phrasavath, David Costa, Bryce Parkinson, Ibukunoluwa Folajimi |
| 12/1/2025 | 1.3 | Final Report V3 Added to Performance Metrics | Nathaly Phrasavath, David Costa, Bryce Parkinson, Ibukunoluwa Folajimi |

3. Introduction

3.1. Document purpose

The purpose of this Final Report is to outline the technical architecture and design framework for Arcane Chess, a modern version of the classic chess game enhanced with trading card mechanics inspired by Magic: The Gathering. This document translates the functional requirements from the Software Requirements Specification (SRS) into detailed design specifications, describing system structure, component interactions, and data flow to guide developers during implementation.

Arcane Chess aims to blend the strategic depth of traditional chess with the dynamic, customizable gameplay of a trading card system. Players will be able to build decks before matches and decide each turn whether to move a chess piece or play a card, introducing new possibilities for tactical decision-making. This SDD ensures that the design supports these objectives through a scalable, maintainable, and well-structured software architecture.

3.2. Product overview

3.2.1 Problem statements

Traditional online chess platforms provide consistent gameplay experiences but offer limited variability or strategic diversity beyond standard moves. This can make matches feel repetitive and predictable, especially for experienced players seeking new layers of depth. Additionally, most platforms focus solely on classic chess mechanics and lack features that blend other genres or dynamic gameplay systems to enhance engagement.

3.2.2 Proposed solution

Arcane Chess introduces an innovative twist to traditional chess by integrating trading card game mechanics into standard chess rules. Players will build custom decks of 16 cards, each containing unique effects that can alter the state of the board, enhance pieces, or manipulate gameplay conditions. Every turn, a player can choose to either move a chess piece or play a card, creating a dual-layered strategy system that balances tactical positioning with card-based decision-making.

The system will be implemented as a web-based application featuring real-time matchmaking, interactive tutorials, and an intuitive user interface that provides visual feedback for all player actions.

3.2.3 Novelty

Arcane Chess stands out by merging two established gaming genres—classical chess and strategic trading card games—into a single, balanced competitive experience. Unlike other chess variants, this system introduces resource management and deck customization, allowing for personalized strategies and evolving gameplay. The combination of real-time web interactivity, visual effects, and deck-building mechanics provides a fresh, engaging approach that revitalizes the timeless appeal of chess while maintaining its intellectual challenge.

3.3. Product functionality

The Arcane Chess system integrates traditional chess gameplay with collectible card mechanics through an interactive, web-based platform. The system's functional requirements outline how players interact with the interface, build decks, enter matchmaking, and engage in matches.

Players will connect through a visual web interface where they can construct a 16-card deck from a catalog of 32 available cards. Once both players confirm their decks, the system automatically pairs them using the matchmaking service. Gameplay follows a turn-based structure, allowing players to move chess pieces or activate card effects that alter the board state.

The interface dynamically reflects game updates in real-time, including animations, piece movements, and visual indicators of captured pieces. Each match is governed by synchronized 15-minute timers per player, ensuring fairness and pacing. When a game concludes, players are prompted to rematch or re-enter the matchmaking queue. Together, these functions ensure a seamless, interactive, and competitive experience that combines the strategic depth of chess with the variability of a trading card game.

3.4. Definitions

Arcane Chess: a digital game that combines traditional chess mechanics with trading card elements, where players can choose to move a chess piece or play a card on their turn.

Card Effect: A special ability or modification applied to the board, pieces, or game state when a card is played. Examples include enhancing a piece, altering movement, or creating environmental effects.

Deck: A collection of cards selected by a player before the match, influencing strategy and gameplay style.

Turn: A single opportunity for a player to either move a piece or play a card.

Piece: A standard chess unit (e.g., pawn, rook, bishop, or any unit new to Arcane Chess) that can move according to traditional chess rules unless modified by card effects.

Board State: The current configuration of all chess pieces, cards in play, and ongoing effects.

3.5. Acronyms and abbreviations

SDD: Software Design Document.

FR: Functional Requirement.

NFR: Non-Functional Requirement.

JS: JavaScript.

HTML: Hypertext Markup Language.

CSS: Cascading Style Sheets.

WS: Web Socket.

ASGI: Asynchronous Server Gateway Interface.

UI: User Interface(s).

4. System requirements

4.1. Functional requirements

1. FR1: The system shall provide a visual web interface for players to interact with using their cursor.

Design Implementation:

- 1.1. Implemented using JavaScript for a responsive, browser-based interface.
- 1.2. The chessboard and cards shall be rendered dynamically using JS.
- 1.3. Event listeners in JavaScript shall capture player actions (clicks, hovers, drags) for piece movement and card play.
- 1.4. The interface communicates with the backend through a WebSocket connection for real-time updates and synchronization between players.
- 1.5. Visual feedback such as animations or color changes will be achieved using JS.
2. FR2: Once connected, players shall build their deck of 16 cards from the available 32 and click a button to enter matchmaking.

Design Implementation:

- 2.1. A deck-building page created with JavaScript will display all available cards as selected elements.
- 2.2. When a player selects or deselects cards, JavaScript shall update the deck configuration dynamically.
- 2.3. The selected deck (16 cards) will be validated client-side, then sent to the server.
- 2.4. The server validates deck size and uniqueness before storing it for matchmaking.
3. FR3: Once two players have joined the queue, players shall be placed into a match together.

Design Implementation:

- 3.1. The client sends a match request to the server using a WebSocket message.
- 3.2. The back-end Game Manager module manages an in-memory queue that pairs available players.
- 3.3. When two players are matched, the server creates a new Game State instance and notifies both clients through WebSocket messages to start the game.
4. FR4: Players shall be able to view an enlarged view of each of their cards by hovering their cursor on the card.

Design Implementation:

- 4.1. JavaScript shall be used to give updates and show animations that will visually represent state change.
- 4.2. The visual state of the game will always reflect the latest synchronized data from the server.
5. FR5: All player actions shall be performed by selecting a piece or card using the cursor, then selecting either where the piece shall move, or which piece/location the card shall affect.

Design Implementation:

- 5.1. JavaScript shall handle all interaction logic for player inputs (selecting a piece, card, or target square).
- 5.2. Each input event triggers a WebSocket message sent to the server containing the move or card details.
- 5.3. The backend validates each action and updates the Game State accordingly.
- 5.4. The updated state is broadcast back to both clients to refresh their views in real time.
- 6. FR6: After performing an action, the player's screen shall be updated according to the action they performed.

Design Implementation:

- 6.1. JavaScript shall be used to give updates and show animations that will visually represent state change.
- 6.2. Visual effects such as a color change or sprite alteration shall indicate the effect of a card.
- 6.3. A piece being moved shall appear in its new location.
- 7. FR7: All Pieces taken shall appear between the board and card playing sections. Opponents taken pieces shall appear near the top while players taken pieces shall appear near the bottom.

Design Implementation:

- 7.1. A visual bar will be displayed along the top of the game interface, showing all pieces that the opponent has captured from the player.
- 7.2. The bottom of the game interface will display the pieces that the player has captured from the opponent for visual balance.
- 7.3. When a piece is captured, the backend sends a WebSocket message to both clients, identifying the captured piece type, its color, and the capturing player.
- 7.4. Upon receiving the event, the frontend JavaScript code updates the captured pieces bar by adding the appropriate piece icon or increasing the count if the same piece type has already been captured.
- 7.5. The captured pieces are non-interactive, serving only as a visual indicator of captured material throughout the match.
- 8. FR8: Cards held in hand shall appear along the right of the view.

Design Implementation:

- 8.1. A card display panel shall be positioned along the right side of the game interface, showing all cards currently held by the player.
- 8.2. Each card will be represented by a rectangular placeholder asset containing the card's name and a short description of its effect.
- 8.3. The front-end JavaScript manages the player's hand state locally, updating the right-side panel in real time as cards are drawn, played, or discarded.
- 8.4. The backend server sends updates to each player through WebSocket messages whenever a card is added to or removed from their hand.
- 9. FR9: The game state shall include a timer for each player which begins at 15 minutes and counts down for each player while it is their turn. If a player's timer reaches 0, they lose the game automatically.

Design Implementation:

- 9.1. A JavaScript timer displays each player's remaining time in the game interface.
 - 9.2. The backend manages the official countdown
 - 9.3. WebSocket updates keep both clients synchronized with the server clock.
 - 9.4. When a timer reaches zero, the server ends the match and declares the opponent as the winner.
10. FR10: When a game concludes, both players shall be prompted to either rematch, or rejoin the queue to find a new opponent.

Design Implementation:

- 10.1. The backend will send a game-end message via WebSocket containing the result and player options.
- 10.2. JavaScript displays a dialog box with "Rematch" and "Return to Queue" options.
- 10.3. Selecting "Rematch" sends a signal to the server to restart the game with the same players.
- 10.4. Selecting "Return to Queue" re-enqueues the player through the matchmaking system.

4.2. Non-functional requirements

1. NFR1: The system shall update the opponent's screen no more than 5 seconds after a player's move has been registered.

Design Implementation:

- 1.1. The backend transmits move updates via WebSocket immediately after processing a turn.
 - 1.2. The front-end listens to these messages and updates the board state in real time.
 - 1.3. If a delay exceeds 5 seconds, the client triggers a resync request to fetch the current state.
 - 1.4. This ensures both players' views remain synchronized throughout the game.
2. NFR2: The server which manages the game state shall have a maximum capacity of 20 concurrent games.

Design Implementation:

- 2.1. Each active game session is managed by a separate Game State instance in memory.
 - 2.2. The server enforces a connection limit of 40 players (20 matches).
 - 2.3. Attempts beyond this limit return a "server full" message to connecting clients.
 - 2.4. Resource usage is monitored to ensure stable performance under maximum load.
3. NFR3: The server shall validate each player's turn, through confirmation of movements and/or played cards based on the previous state of the gameboard and players' hands.

Design Implementation:

- 3.1. Each move or card play is checked against the Game State before being accepted.
- 3.2. Invalid or duplicate actions are ignored, and an error message is returned to the client.
- 3.3. Validation includes piece legality, card rules, and turn ownership.
- 3.4. All confirmed actions are logged for debugging purposes.

4. NFR4: The system shall have a dedicated URL for players to access the game.

Design Implementation:

- 4.1. The web app is hosted at a fixed URL.
 - 4.2. A landing page provides a “Play Game” button linking to the main game client.
 - 4.3. Routing ensures secure and direct access to the multiplayer environment.
5. NFR5: The server shall be hosted on a team member’s machine during the duration of the project.

Design Implementation:

- 5.1. The game server runs locally using Uvicorn or a similar lightweight Python web server.
 - 5.2. External players connect via local network port forwarding.
 - 5.3. Regular backups and logs are maintained by the hosting team members.
 - 5.4. Hosting configuration is documented for easy handoff if needed.
6. NFR6: The game shall be accessible from any web browser regardless of the user’s machine or operating system.

Design Implementation:

- 6.1. The front-end is built using JavaScript for universal compatibility.
- 6.2. Tested browsers include Chrome, Edge, Firefox, and Safari.
- 6.3. The UI automatically scales for different screen resolutions.
- 6.4. No installation or plugins are required — the game runs entirely in the browser.

4.3. Other requirements

N/A.

5. System architecture

5.1. Overall architecture

The Arcane Chess system follows a hybrid Model–View–Controller (MVC) architecture, extended with a Game Management (Service) layer. The MVC structure promotes modularity and maintainability by separating domain logic, presentation, and user interaction. The additional Service layer provides centralized orchestration for managing multiple concurrent games and encapsulating server responsibilities. This combination ensures that the system remains scalable, testable, and adaptable.

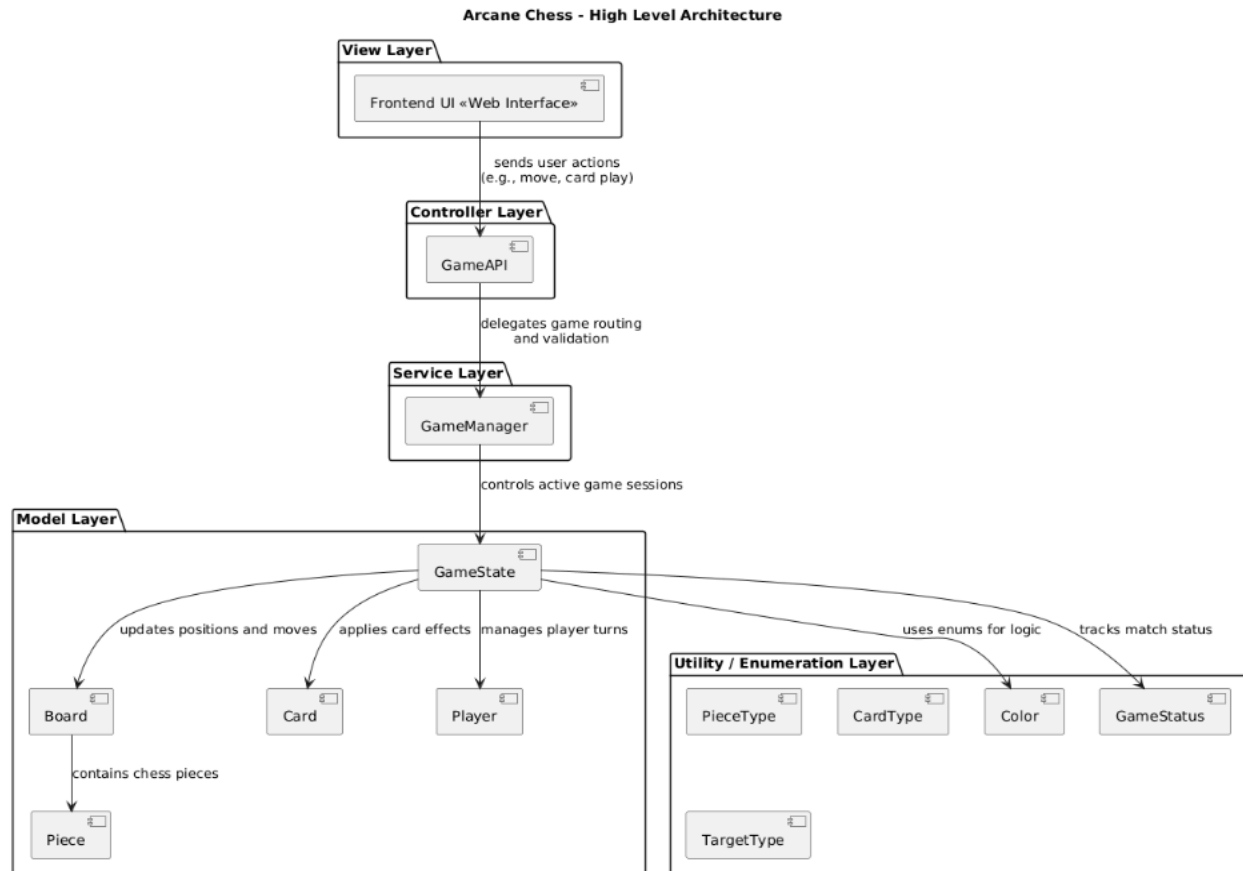


Figure 1: High-Level Architecture

5.2. Components mapping

5.2.1. Functional requirements

| Requirement | Component Classes | System Architecture Component(s) | Technologies |
|-------------|---------------------------------|----------------------------------|-------------------------------|
| FR1 | N/A | View | CSS, Javascript |
| FR2 | Deck, Player, GameManager | Model, Controller | Python, WebSocket, JavaScript |
| FR3 | GameManager, Player, GameState, | Controller, Service | Python, WebSocket |
| FR4 | Card | View | JavaScript |

| | | | |
|--------------------|--------------------------------|-------------------|-------------------------------|
| <u>FR5</u> | Piece, Card, Board, GameState | Controller, Model | Python, WebSocket, JavaScript |
| <u>FR6</u> | Board, GameState, Move | Controller, View | JavaScript, WebSocket |
| <u>FR7</u> | Board, Piece, GameState | View, Model | JavaScript, WebSocket |
| <u>FR8</u> | Card, Player, GameState | View | JavaScript, WebSocket |
| <u>FR9</u> | GameState | Controller | Python, WebSocket, JavaScript |
| <u>FR10</u> | GameManager, Player, GameState | Controller | Python, WebSocket, JavaScript |

5.2.2. Non-functional requirements

| Requirement | Component Classes | System Architecture Component(s) | Technologies |
|--------------------|--|----------------------------------|--------------------------------|
| <u>NFR1</u> | GameState, Board | Controller, Communication | Python (WebSocket), JavaScript |
| <u>NFR2</u> | GameManager, GameState, GameAPI | Controller | Uvicorn, Python |
| <u>NFR3</u> | GameState, Board, GameManager, GameAPI | Model, Service | Python, Websocket |
| <u>NFR4</u> | GameAPI | View, Controller | Uvicorn, Websocket, Javascript |
| <u>NFR5</u> | GameAPI, GameManager | Service | Websocket, Uvicorn |
| <u>NFR6</u> | N/A | View, Service | Javascript |

5.3. Technology stack selection

Python 3.8+: A versatile language which supports OOP paradigms that are useful for design of chess pieces/objects. 3.8+ because only those versions support FastAPI

FastAPI: A modern framework for building web API's with python to act as the controller. Will handle interactions between the model (game logic) and the view (web page).

Uvicorn: An ASGI web server implementation for Python. Which supports WS connections. This will facilitate communication between clients and our server/application.

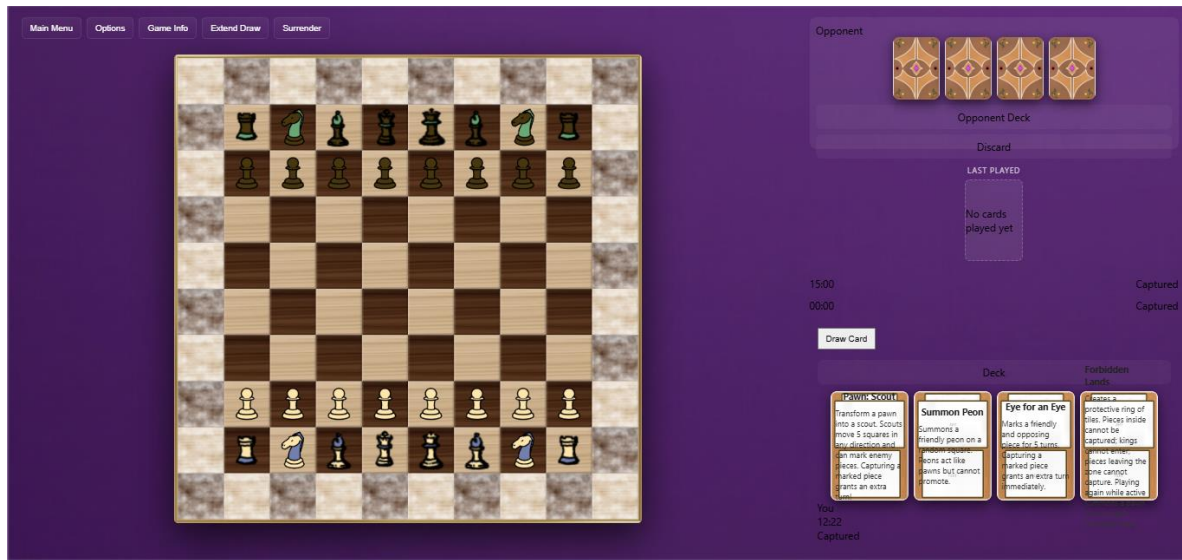
JavaScript: A versatile, high-level programming language used to create dynamic and interactive features on websites and applications. It enables developers to build responsive interfaces, handle real-time updates, and interact seamlessly with backend services. In our case, JavaScript

provides the flexibility and power needed to build the front end efficiently while delivering a modern, engaging user experience.

CSS: The language used to style HTML documents and, in this case, JavaScript components. It is absolutely necessary for the webpage to appear professionally built.

6. System Design

6.1. UI



Figure

2: Game Board



Figure 3: Web Landing Page

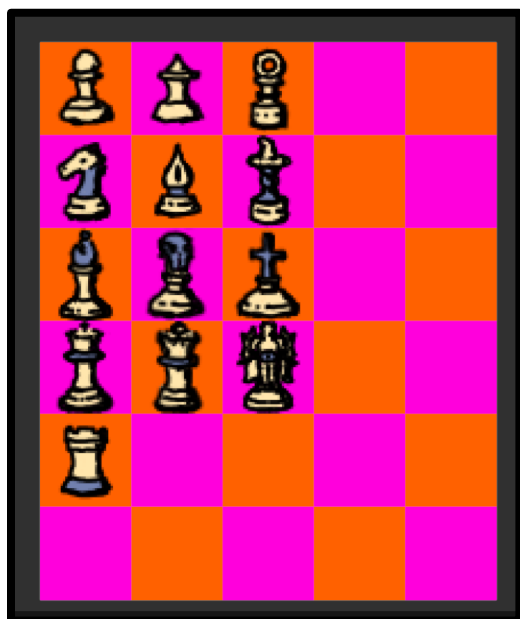


Figure 4: Piece (White)

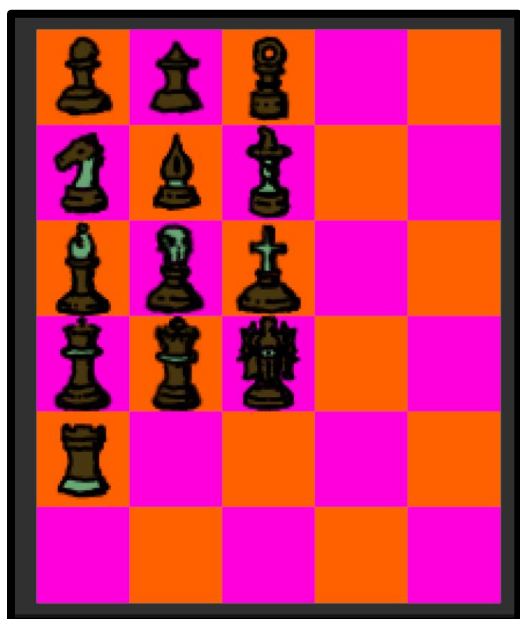
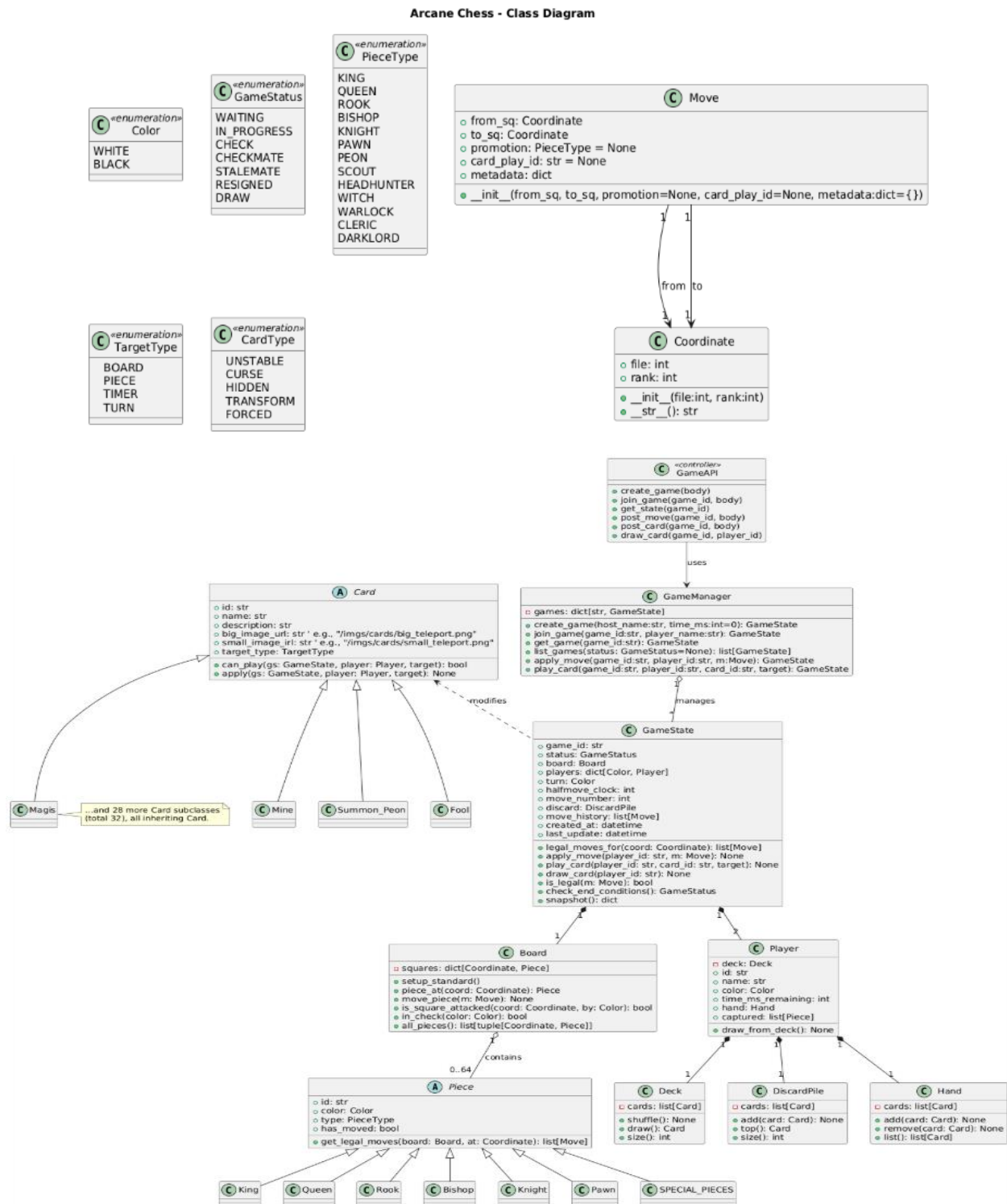
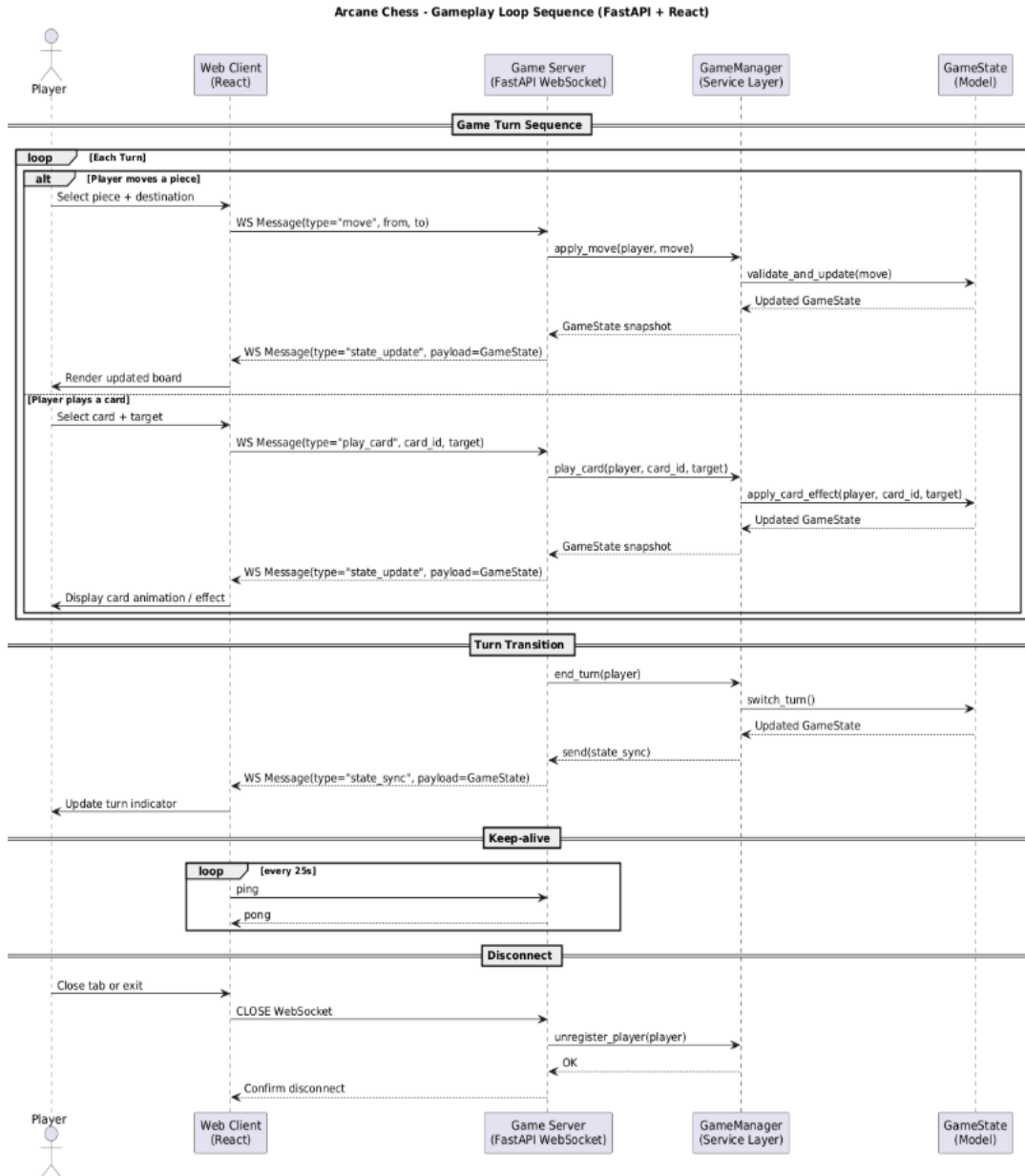


Figure 5: Pieces (Black)

6.2. Class diagram



6.3. Sequence/activity diagram



6.4. Database

N/A.

7. Others

7.1 Algorithms

- **Matchmaking Algorithm:**
The matchmaking system pairs players based on their availability in the queue. It ensures that two players are matched in the order they entered the queue to maintain fairness and reduce waiting time.
- **Card Effect Resolution Algorithm:**
Each card has unique effects defined. When a card is played, the system resolves its effect using a priority-based stack.
- **Move Validation Algorithm:**
Movement logic is implemented using coordinate-based validation, verifying that each piece move complies with its rules.

7.2 Third-Party Libraries and APIs

- **Backend:**
 - *Python.*
- **Frontend:**
 - *Javascript.*
 - *HTML.*
 - *CSS.*
- **Server Hosting:**
 - *Uvicorn (Python).*
 - *Websockets.*
- **Version Control:**
 - *GitHub* for repository hosting.

8. Test plan

8.1. Game Interface and Player Interaction

8.1.1. Launching the Game UI

Table 1

| No. | Test case | User input | Pass criteria |
|-----|-----------------|------------------------------|--|
| 1 | Launch the game | Open the game URL in browser | The interface loads successfully with Web landing page visible |

8.1.2. Piece and Card Interaction

Table 2

| No. | Test case | User input | Pass criteria |
|-----|-----------|------------|---------------|
|-----|-----------|------------|---------------|

| | | | |
|----|---|--|--|
| 1 | Correct squares are marked as available moves | Click a piece | Marked squares appear on user's screen in accordance with expected available moves depending on piece type |
| 2 | Select and move a piece | Click a piece -> click invalid target | Piece should not move to the select target and should display invalid move message |
| 3 | Select and move a piece | Click a piece -> click valid target | Piece moves correctly and board updates |
| 4 | Play a card | Click a card -> click again to see the option to activate card -> select yes | Card animation plays and state updates |
| 5 | Play a card | Click a card -> click again to see the option to activate card -> select no | No card animation plays, state is maintained, and the player is able to reperform the action |
| 6 | Hover over card | Hover cursor over card | Enlarged card view appears with effect description |
| 7 | Play valid card | Player plays card from hand on their turn | Card removed from hand, added to discard, new card drawn |
| 8 | Play card not in hand | Attempt to play card_id not in player's hand | Returns error "Card not in hand" |
| 9 | Play card on opponent's turn | Attempt to play card when not current player | Returns error "Not your turn" |
| 10 | Hand size limit | Start with Max_hand_size cards, play 1 | Hand size returns to Max_hand_size after drawing |
| 11 | Legal rook move | Move rook vertically on empty file | Move succeeds, piece position updates |
| 12 | Rook blocked by own piece | Attempt rook move through friendly piece | Move rejected, returns "Illegal move" |
| 13 | Rook capture | Move rook to square with enemy piece | Piece captured, added to captured list, rook moves |
| 14 | Bishop diagonal move | Move bishop diagonally 3 squares | Move succeeds |
| 15 | Knight L-shape move | Move knight in L-pattern over pieces | Move succeeds regardless of blocking pieces |
| 16 | Pawn initial double move | Move pawn 2 squares on first move | Move succeeds |
| 17 | Pawn single move only | Attempt 2-square pawn move after first move | Move rejected |
| 18 | Pawn diagonal capture | Move pawn diagonally to capture | Capture succeeds |
| 19 | Pawn forward capture attempt | Attempt pawn forward move onto enemy piece | Move rejected |

8.2. Matchmaking and Deck Building

8.2.1. Deck Creation

Table 3

| No. | Test case | User input | Pass criteria |
|-----|-------------------------|-----------------------------------|--|
| 1 | Select cards for deck | Choose cards from available 32 | Selected cards highlight and count updates |
| 2 | Validate deck size | Attempt to select >16 cards | System prevents selecting more than 16 cards |
| 3 | Validate deck size | Attempt to select <16 cards | System prevents selecting less than 16 cards |
| 4 | Confirm deck submission | Click "Confirm Deck" | Deck submits successfully and server validates it |
| 5 | Reject invalid deck | Call add_to_queue() with 15 cards | Returns error "Deck must contain exactly 16 cards" |
| 6 | Reject invalid deck | Call add_to_queue() with 17 cards | Returns error "Deck must contain exactly 16 cards" |

8.2.2. Matchmaking

Table 4

| No. | Test case | User input | Pass criteria |
|-----|-------------------------------|--|---|
| 1 | Enter matchmaking queue | Click "Find Match" | Player enters queue and waits for opponent |
| 2 | Match creation | Two players in queue | Players are paired, removed from queue, and game starts |
| 3 | Add player to queue | Call add_to_queue() with valid deck (16 cards) | Player added successfully, returns success message |
| 4 | Prevent duplicate queue entry | Add same player to queue twice | Second attempt returns "Already in matchmaking queue" |

8.3. Game State and Turn Logic

8.3.1. Timer

Table 5

| No. | Test case | User input | Pass criteria |
|-----|----------------------|---|--|
| 1 | Timer countdown | Begin player's turn | Timer starts counting down from 15 minutes |
| 2 | Timeout condition | Let timer reach 0 | Game ends and opponent declared winner |
| 3 | Timer countdown | Wait 5 seconds during turn | Current player's time decreases by 5 seconds |
| 4 | Turn switching | Complete valid move | Turn switches to opponent, timer resets for new turn |
| 5 | Timer task starts | Server startup with start_timer_updates() | Background task running, updates every second |
| 6 | Multiple game timers | 5 active games running | All 5 games' timers update correctly |

| | | | |
|---|-------------------------|---|---|
| 7 | Timer stops on game end | Game ends in checkmate | Timer for that game stops updating |
| 8 | Timeout callback | Player runs out of time | Callback function called with game object |
| 9 | Timer task cleanup | Server shutdown with stop_timer_updates() | Background task cancelled cleanly |

8.3.2. Move Validation

Table 6

| No. | Test case | User input | Pass criteria |
|-----|------------------------|------------------------|--|
| 1 | Valid move execution | Select legal move | Move executes correctly and syncs with opponent |
| 2 | Invalid move rejection | Attempt illegal move | System rejects move and displays warning |
| 3 | Capture event | Capture opponent piece | Captured piece appears in capture bar Captured opponent piece is replaced by player piece |

8.4. Game End and Replay Options

8.4.1 Game Conclusion

Table 7

| No. | Test case | User input | Pass criteria |
|-----|------------------|---------------------------------|---|
| 1 | End by checkmate | Perform final move to checkmate | Game ends and displays winner/loser message |
| 2 | End by timeout | Allow timer to hit 0 | Match ends and opponent declared winner |

8.4.2 Post-Game Options

Table 8

| No. | Test case | User input | Pass criteria |
|-----|-----------------------|-------------------------|---|
| 1 | Request rematch | Click "Rematch" | New match starts with the same players if the opponent also clicked "Rematch" Otherwise, player re-enters matchmaking system |
| 2 | Return to matchmaking | Click "Return to Queue" | Player re-enters matchmaking system |

8.5. UI Display Elements

8.5.1 Captured Pieces Display

Table 9

| No. | Test case | User input | Pass criteria |
|-----|-------------------------|---------------------------|---|
| 1 | Display captured pieces | Capture opponent's piece | Piece appears in correct capture bar (top/bottom) |
| 2 | Multiple captures | Capture multiple pieces | Each piece icon updates with count |
| 3 | Opponent capture | Opponent captures a piece | Captured piece appears on opponent's bar |
| 4 | Refresh synchronization | Reload browser | Board state reload accurately |

8.5.2 Card Hand Panel

Table 10

| No. | Test case | User input | Pass criteria |
|-----|---------------|------------------------------|---|
| 1 | Display hand | Start game | 4 drawn cards appear on right panel |
| 2 | Draw new card | Play turn that triggers draw | New card appears immediately in hand |
| 3 | Play card | Select and play valid card | Card removed from hand and action processed |
| 4 | Hover preview | Hover over card | Enlarged card preview appears |

8.6. System Performance

8.6.1. Screen Update Delay

Table 11

| No. | Test case | User input | Pass criteria |
|-----|----------------------|------------------------|--|
| 1 | Move synchronization | Player makes move | Opponent board updates within 5 seconds |
| 2 | Network lag recovery | Temporarily disconnect | Board resyncs automatically on reconnect |

8.6.2. Server Load Capacity

Table 12

| No. | Test case | User input | Pass criteria |
|-----|-----------------------|------------------------------|---|
| 1 | Multiple active games | Start 20 concurrent matches | Server runs all matches without failure |
| 2 | Over-capacity test | Attempt 21st game connection | System denies connection with "server full" message |

8.7. Accessibility and Compatibility

8.7.1. Cross-Browser Access

Table 13

| No. | Test case | User input | Pass criteria |
|-----|---------------------|---------------------|-------------------------|
| 1 | Access from Chrome | Open URL in Chrome | Game loads successfully |
| 2 | Access from Firefox | Open URL in Firefox | Game loads successfully |
| 3 | Access from Safari | Open URL in Safari | Game loads successfully |

8.7.2. Hosting and URL

Table 14

| No. | Test case | User input | Pass criteria |
|-----|--------------------------|----------------------------|--|
| 1 | Access via dedicated URL | Enter official game URL | Player reaches Arcane Chess homepage |
| 2 | Server availability | Access when server offline | System displays "server unavailable" message |

8.8. Board Class

Table 15

| No. | Test case | User input | Pass criteria |
|-----|---|---|---|
| 1 | Initialize Board and verify it starts empty | Call Board() | Board is created with squares = {} and dmzActive = False |
| 2 | Setup standard board layout | Call setup_standard() | 32 pieces are placed correctly in starting positions |
| 3 | Retrieve a piece at a given coordinate | piece_at_coord(Coordinate(5,1)) | Returns the White King (id='wK') |
| 4 | Check if coordinate is inside bounds | is_in_bounds(Coordinate(5,5)) | Returns True if DMZ is inactive; otherwise checks full 10×10 grid |
| 5 | Verify empty and occupied squares | is_empty(Coordinate(5,5)) , is_empty(Coordinate(5,1)) | Returns True for empty and False for occupied square |
| 6 | Move a piece to a valid coordinate | Move from Coordinate(5,1) → Coordinate(5,2) | Piece is moved, origin cleared, and destination filled |
| 7 | Attempt invalid move (no piece) | Move from empty coordinate | Raises ValueError("No piece at...") |
| 8 | Test capture functionality | Move to a coordinate occupied by an opponent | Returns captured piece object |
| 9 | Test cloning of board | clone() | Returns deep copy; original and clone do not affect each other |

| | | | |
|----|-----------------------|-----------|--|
| 10 | Serialize board state | to_dict() | Returns JSON dictionary with "dmzActive" and "pieces" list |
|----|-----------------------|-----------|--|

8.9. Piece Class

Table 16

| No. | Test case | User input | Pass criteria |
|-----|-------------------------------|--|---|
| 1 | Initialize a specific piece | Rook("wR01", Color.WHITE) | Piece is created with correct ID, color, and type |
| 2 | Generate legal moves | get_legal_moves(board, Coordinate(1,1)) | Returns all valid moves within boundaries |
| 3 | Generate capture moves | get_legal_captures(board, Coordinate(1,1)) | Returns only enemy-occupied coordinates |
| 4 | Test serialization | to_dict(at=Coordinate(1,1)) | Returns dictionary with id, type, color, and position |
| 5 | Test serialization with moves | to_dict(at=Coordinate(1,1), include_moves=True, board=board) | Includes "moves" list in output |

8.10. Deck Class

Table 17

| No. | Test case | User input | Pass criteria |
|-----|-------------------------|-----------------------------|---|
| 1 | Add a valid card | deck.add_card(card) | Card is added; deck size increases by 1 |
| 2 | Add invalid object | deck.add_card("NotACard") | Raises TypeError |
| 3 | Enforce deck size limit | Add 17th card | Raises ValueError("Deck cannot hold more than 16 cards.") |
| 4 | Draw a card | deck.draw() | Returns and removes the top card |
| 5 | Draw from empty deck | Empty deck then call draw() | Raises ValueError("Deck is empty.") |
| 6 | Shuffle deck | deck.shuffle() | Randomly reorders cards (order differs from before) |
| 7 | Check top card | deck.top() | Returns last card in list without removing it |
| 8 | Verify size method | deck.size() | Returns number of cards currently in deck |

8.11. Hand Class

Table 18

| No. | Test case | User input | Pass criteria |
|-----|-----------|------------|---------------|
|-----|-----------|------------|---------------|

| | | | |
|---|--------------------------|------------------------|---|
| 1 | Add a valid card to hand | hand.add(card) | Card is added successfully |
| 2 | Add invalid type | hand.add("NotACard") | Raises TypeError |
| 3 | Remove card in hand | hand.remove(card) | Card is removed from hand |
| 4 | Remove non-existent card | hand.remove(fake_card) | Does nothing, no error raised |
| 5 | Get list of cards | hand.list() | Returns list of current cards |
| 6 | Check hand length | len(hand) | Returns number of cards currently in hand |

8.12. Move Class

Table 19

| No. | Test case | User input | Pass criteria |
|-----|----------------------|----------------------------------|---|
| 1 | Create coordinate | Coordinate(3,4) | Initializes correctly with file=3, rank=4 |
| 2 | Check hashing | Use Coordinate as dictionary key | No TypeError; key works correctly |
| 3 | Offset coordinate | offset(1,1) from (3,4) | Returns new coordinate (4,5) if in bounds |
| 4 | Algebraic conversion | to_algebraic() for (4,5) | Returns "e5" |

8.13. All Piece Classes

Table 20

| No. | Test case | User input | Pass criteria |
|-----|--------------------------------|--|---|
| 1 | Initialize a piece | Create any subclass, e.g. Rook("wR01", Color.WHITE) | Object is created with correct ID, color, and type from PieceType |
| 2 | Verify string representation | __str__(piece) | Returns formatted string like "W R (wR01)" |
| 3 | Generate legal moves | Call get_legal_moves(board, coordinate) | Returns a list of valid Move objects according to movement rules and board bounds |
| 4 | Generate capture moves | Call get_legal_captures(board, coordinate) | Returns only moves that capture enemy pieces |
| 5 | Prevent out-of-bounds movement | Place piece near board edge and call get_legal_moves() | Returned moves stay within valid coordinate range |
| 6 | Obey blocking rules | Place allied piece along a valid move path | Legal move list stops before allied piece (cannot move through or capture it). |
| 7 | Allow capturing enemies | Place enemy piece along valid move path | Returned move list includes that coordinate once, then stops beyond it. |

| | | | |
|----|--------------------------------|--|---|
| 8 | Verify move exclusivity | Piece in open area (no blockers) | Legal moves do not duplicate coordinates and follow movement pattern correctly. |
| 9 | Validate color awareness | Place two pieces of same color nearby | No moves include friendly piece coordinates. |
| 10 | Validate algebraic notation | Call <code>algebraic_notation()</code> | Returns correct single-character notation (e.g., K, Q, R, B, N, P). |
| 11 | Check serialization | Call <code>to_dict(at=Coordinate(...))</code> | Returns dictionary with "id", "type", "color", and "position". |
| 12 | Include moves in serialization | Call <code>to_dict(at=..., include_moves=True, board=board)</code> | Dictionary includes "moves" key with correct from and to coordinates. |

8.14. Game Manager

Table 21

| No. | Test case | User input | Pass criteria |
|-----|------------------------|--|---|
| 1 | Create sample game | Call <code>create_sample_game()</code> | Game created with valid <code>game_id</code> , two players, and initialized board |
| 2 | Retrieve game by ID | Call <code>get_game(game_id)</code> | Returns correct GameState object |
| 3 | Find player's game | Call <code>get_player_game(player_id)</code> | Returns game the player is in, or None |
| 4 | Get game statistics | Call <code>get_stats()</code> | Returns correct counts for active games, queue size, finished games |
| 5 | Cleanup finished games | Call <code>cleanup_finished_games()</code> | Removes all non-IN_PROGRESS games from memory |

8.15. Game State

Table 22

| No. | Test case | User input | Pass criteria |
|-----|--------------------------|--------------------------------------|---|
| 1 | Initialize game state | Create new GameState | Board initialized, White's turn, both players have 900s on clock |
| 2 | Get player perspective | Call <code>to_dict(player_id)</code> | Returns state with player's hand visible, opponent's hand size only |
| 3 | Halfmove clock increment | Make non-capture, non-pawn move | Halfmove clock increments by 1 |
| 4 | Halfmove clock reset | Capture piece or move pawn | Halfmove clock resets to 0 |
| 5 | Fullmove increment | Black completes turn | Fullmove number increments by 1 |
| 6 | Fifty-move rule | Reach 100 halfmoves | Game status becomes DRAW with reason "Fifty-move rule" |

8.16. WebSocket Communications

Table 23

| No. | Test case | User input | Pass criteria |
|-----|------------------------------|---|---|
| 1 | Client connection | Connect to <code>/ws/{client_id}</code> | WebSocket accepted, client added to active connections |
| 2 | Game start notification | Two players matched | Both receive <code>{"type": "game_started"}</code> with <code>game_state</code> |
| 3 | Error message | Send invalid move | Receive <code>{"type": "error", "message": "..."} </code> |
| 4 | Ping/pong | Send <code>{"type": "ping"}</code> | Receive <code>{"type": "pong"}</code> |
| 5 | Client disconnect | Close WebSocket connection | Client removed from active connections and queue |
| 6 | Opponent disconnect handling | One player disconnects mid-game | Other player notified, game marked as forfeit |
| 7 | Broadcast to game | One player makes move | Both players receive identical game state update |

8.17. Coordinate System

Table 24

| No. | Test case | User input | Pass criteria |
|-----|--------------------------|---|--|
| 1 | Parse algebraic notation | <code>Coordinate.from_algebraic("e4")</code> | Returns <code>Coordinate</code> with <code>file=4</code> , <code>rank=3</code> (0-indexed) |
| 2 | Invalid notation | <code>Coordinate.from_algebraic("z9")</code> | Raises exception or returns <code>None</code> |
| 3 | Coordinate offset | Call <code>offset(1, 1)</code> on <code>"e4"</code> | Returns <code>"f5"</code> coordinate |
| 4 | Offset out of bounds | Call <code>offset(5, 0)</code> on <code>"h4"</code> | Returns <code>None</code> (off board) |
| 5 | Coordinate equality | Compare two <code>Coordinate("e4")</code> objects | Returns <code>True</code> |
| 6 | Convert to string | Call <code>str()</code> on <code>Coordinate</code> | Returns algebraic notation <code>"e4"</code> |

8.18. Player Management

Table 25

| No. | Test case | User input | Pass criteria |
|-----|-----------------------|---|---|
| 1 | Player initialization | Create Player with deck | Player has correct color, 4 cards in hand, 12 in deck |
| 2 | Draw card | Call <code>draw_card()</code> | Card moved from deck to hand, returns card object |
| 3 | Draw from empty deck | Call <code>draw_card()</code> when deck empty | Returns <code>None</code> , hand unchanged |

| | | | |
|----|-----------------------|--|--|
| 4 | Play card | Call <code>play_card(card_id)</code> | Card moved from hand to discard, returns card object |
| 5 | Has card check | Call <code>has_card(card_id)</code> for card in hand | Returns True |
| 6 | Capture piece | Call <code>capture_piece(piece)</code> | Piece added to captured list |
| 7 | Get hand size | Call <code>hand_size()</code> | Returns correct integer |
| 8 | Get deck size | Call <code>deck_size()</code> | Returns correct integer |
| 9 | Serialize player | Call <code>to_dict()</code> | Returns dict with id, name, color, hand, captures |
| 10 | Player representation | Call <code>str(player)</code> | Returns formatted string with name and color |

9. Delivery Metrics

Number of Features Delivered

- Implemented core Arcane Chess game logic
- Added card system with multiple playable cards
- Completed backend API endpoints for gameplay and user actions
- Delivered JavaScript UI screens for game board and homepage

Sprint Velocity

- Averaged 2–3 points per sprint (roughly one per team member)
- Smaller tasks delivered continuously throughout each sprint

Cycle Time

- Average cycle time for a feature: 3–5 days for major creations such as visuals and front-end connecting to back-end
- Minor UI tasks completed within 1–2 days

10. Release Summary

Initial Release (Alpha Build)

- Introduced core Arcane Chess gameplay: board setup, piece movement, turn system
- Added foundational backend structure (game state, piece classes, card framework)
- Implemented basic UI for game board rendering and user interactions

Feature Release 1 (Beta Build)

- Added first set of playable cards (e.g., Mine, Shroud, Pawn Queen, Warlock)
- Integrated backend card logic with frontend UI for smooth triggering and animations
- Fixed early defects in movement validation, card resolution order, and API responses

Final Release (Stable Build)

- Delivered full card library with completed effects, cooldowns, and hidden mechanics
- Implemented UI enhancements
- Completed system integration, resolving major bugs and improving performance and stability

11. Performance Metrics

- To validate NFR1, which states that “the system shall update the opponent’s screen no more than 5 seconds after a player’s move has been registered”, a performance test was conducted using 20 concurrent Arcane Chess games running simultaneously.
- For each game instance:
 - White made a move, and the system recorded the time until the Black player’s board rendered the update.
 - Black made a move, and the system recorded the time until the White player’s board rendered the update.
- Each game produced two measurements:
 - White → Black render delay
 - Black → White render delay
- These measurements capture the total round-trip delay including:
 - WebSocket send latency
 - Backend processing time
 - Server → client update delivery
 - Frontend game state update
 - Canvas render and browser paint
- Test Configuration:
 - Concurrent game sessions: 20
 - Moves measured per game: 2 (1 white move, 1 black move)
 - Total measurements collected: 40
 - Server load: Maximum allowed (aligned with NFR2)
 - Environment: Local server hosting (Uvicorn), browser clients, WebSocket real-time synchronization
 - Measurement method:
 - Timestamp logged at sendMove()
 - Timestamp logged after board re-render
 - Difference = opponent render delay

Summary of Results

| Metric | Result |
|-----------------------------|----------|
| Number of concurrent games | 20 |
| Total moves recorded | 40 |
| Requirement threshold | ≤ 5000ms |
| Average White → Black delay | 45.69ms |
| Maximum White → Black delay | 62.70ms |

| | |
|-----------------------------|-----------|
| Average Black → White delay | 48.79ms |
| Maximum Black → White delay | 67.90ms |
| Overall pass/fail | pass/fail |

Per-Game Render Times

| Game | White → Black (ms) | Black → White (ms) |
|------|--------------------|--------------------|
| 1 | 41.30ms | 47.00ms |
| 2 | 50.80ms | 48.30ms |
| 3 | 51.60ms | 38.80ms |
| 4 | 43.70ms | 42.20ms |
| 5 | 37.20ms | 41.80ms |
| 6 | 45.30ms | 45.30ms |
| 7 | 47.60ms | 47.20ms |
| 8 | 41.10ms | 48.40ms |
| 9 | 62.70ms | 45.90ms |
| 10 | 39.70ms | 47.80ms |
| 11 | 50.40ms | 37.60ms |
| 12 | 56.40ms | 65.30ms |
| 13 | 48.20ms | 44.60ms |
| 14 | 52.20ms | 67.90ms |
| 15 | 49.00ms | 51.00ms |
| 16 | 46.90ms | 46.10ms |
| 17 | 46.30ms | 41.90ms |
| 18 | 45.80ms | 45.40ms |
| 19 | 43.50ms | 46.50ms |
| 20 | 51.70ms | 39.10ms |

12. Defect Management and Known Issues

- Reported logic defects and resolved it such as incorrect card effects triggering, inconsistent piece movement validation, or unexpected behavior.
- Resolved UI/UX defects, including occasional delays in player movement or hover states.
- Identified and fixed backend/API defects, such as improper state updates or mismatched data returned to the frontend.

13. Test Summary

- Executed unit tests for key backend components.
- Achieved approximately 70–80% test coverage across core game logic.
- Conducted manual UI testing for card interactions, board visualization, and user flow.

- Identified issues mainly related to edge-case card behaviors and resolved them before final integration.

14. Customer/Stakeholder Feedback

The professor, acting as the primary stakeholder, provided overall positive feedback on the Software Design Documentation presentation. Feedback noted that the demo was strong, the server was functioning properly, and the UI appeared polished and well-designed. Additional comments highlighted that the architecture mapping visual was well executed. For improvements, the professor advised adding a slide to explain the game rules. The professor also indicated that the technology stack slide contained excessive text and should be simplified by keeping the content broad and removing lengthy descriptions. Finally, although the functional and non-functional requirements tables were viewed positively, the professor stated that the tables were too busy for the main presentation and should be moved to the appendix, with a more streamlined version presented in the slides.

15. Lessons Learned, Recommendations and Future Work

Lessons Learned

- Real-time WebSocket communication requires early end-to-end testing to avoid sync issues.
- Clear API/message formatting between frontend and backend is essential for smooth integration.
- Visual assets are a core focus of any polished product like Arcane Chess but given the time they take to draw/create it should be more stressed that work on these should start early.
- It's difficult to plan the entire project ahead of time, and that isn't part of Agile.

Recommendations

- Define API structures and data models before implementation.
- Introduce automated testing earlier in the development cycle.
- Use smaller feature branches and frequent code reviews to reduce merge conflicts.

Future Work

- Add ranked matchmaking, player profiles, and statistics.
- Expand card sets and introduce new piece transformations.
- Improve responsiveness for mobile devices and potentially develop a mobile app.
- Implement the cards that were not implemented:
 - Curse: Principle - Upon activation any effigy that is captured by the enemy has its effects transferred to this one. If an enemy piece attempts to capture this effigy before it is allowed to activate, capture the enemy piece. If the secondary effect is activated, the effigy becomes a neutral piece, which neither player controls, but both players can capture.
 - Curse: Death - Upon this effigy's activation, if Unstable: Death is played, every uncaptured piece on both sides returns to its home square and the effigy dies. Any additional pieces will fill from back rank forward, left to right in order of summon.

Barriers are placed upon D4 and E5 if white plays Death, or D5 and E4 if black plays death. The player who played Death can move through these barriers, but cannot capture through them. All copies of this card are removed from the deck and replaced with blank cards.

- Curse: Power of Two - Upon activation, wait 5 turns, and becomes a second king. Checks no long force moves until one king is captured. Kings become susceptible to explosions. If both kings die simultaneously, lose the game.
- Intangible - select one piece of value > 1 but < 6 it becomes intangible for 3 moves, neither able to capture, nor be captured. While intangible, it is able to occupy the same square as something else. If it becomes tangible on the same tile as another piece, both pieces are captured, unless it is a piece of higher value, in which case, only the intangible piece dies. Cannot move through barricade, nor into the DMZ.
- Forced: Play - User is able to move this turn, opponent is forced to play a card next turn.
- Unstable: Sun - does not notify other player it is played, allows user to move this turn, user to play 2 other cards next turn. Unstable cards are hidden, and cannot appear, nor be used more than once per game.
- Unstable: Moon - does not notify other player it is played, allows user to move this turn, next turn player will be able to move, and then play a card. Unstable cards are hidden, and cannot appear, nor be used more than once per game.
- Unstable: Fool - does not notify other player it is played, allows player to move this turn, upon either player reaching checkmate, switch kings. If both kings are in a situation that would cause checkmate, tie. Unstable cards are hidden, and cannot appear, nor be used more than once per game.
- Unstable: Hierophant - does not notify other player it is played, allows player to move this turn, select 3 pieces to combine, and select one of those pieces to become any piece equal to the sum of those pieces or below next turn. These pieces cannot be moved this turn. If any of the three pieces are captured, or a Forced card is played, Magis is interrupted, cancelled, and discarded. Unstable cards are hidden, and cannot appear, nor be used more than once per game.
- Unstable: Death - see Curse: Death, cannot be played until Curse: Death is active.

16. Conclusions

- Arcane Chess successfully integrates traditional chess mechanics with a dynamic card-based system, delivering a unique and engaging real-time multiplayer experience.
- The system met its core functional and non-functional requirements, including real-time synchronization, deck building, matchmaking, and stable game state management.
- Testing confirmed strong performance, reliable gameplay flow, and effective communication between frontend and backend components.
- Key recommendations include improving early API planning, expanding automated testing, and enhancing workflow through feature-based branching and reviews.
- Future work includes adding ranked matchmaking, expanding the card library, and improving mobile accessibility.

17. References

- UML diagrams made using **PlantUML** online editor: <https://plantuml.com/>
- Drawn created with **Paint.net**: <https://www.getpaint.net/index.html>
- **FastAPI** documentation: <https://fastapi.tiangolo.com/>
- **Uvicorn** documentation: <https://uvicorn.dev/>