Design and Implementation of a Real-Time, Multi-Strategy AI-Powered Web-Based Strategy Game.

Yash Chaudhari, Satyveer Chauhan, Kartikey Aggarwal, Dr. Sangeeta Mishra

Electronics and Telecommunications, Thakur College of Engineering and Technology Mumbai, Maharashtra

DOI: https://doi.org/10.51583/IJLTEMAS.2025.1410000116

Abstract— The paradigm of web applications has shifted from static content delivery to dynamic, real-time interactive systems. This paper presents a comprehensive study on the design, implementation, and performance evaluation of "Sphere Strike," a full-stack, browser-based abstract strategy game. While the core technologies utilized Flask, Socket.IO, and heuristic-based AI are well-established, the novelty of this work lies in their specific synthesis to address a discernible gap in existing digital versions of the game "Chain Reaction." We detail a client-server system architected on a Python Flask backend, utilizing the Socket.IO protocol over WebSockets to facilitate low-latency, event-driven communication. A primary contribution is the engineering of a multi-strategy, heuristic-based Artificial Intelligence (AI) agent with tiered difficulty levels. The agent's intelligence is derived from a detailed heuristic evaluation function, and its most advanced tier employs a 1-ply lookahead search to identify and exploit game-winning tactical opportunities. The system was containerized using Docker and deployed on a cloud platform (Render.com) to validate its production-readiness. Empirical performance evaluation confirms that the system maintains a mean round-trip network latency of under 200ms and an AI move calculation time consistently under 100ms, delivering a seamless user experience.

Keywords: Real-Time Systems, WebSockets, Socket.IO, Flask, Game AI, Heuristic Functions, Minimax, Client-Server Architecture, Multiplayer Games, Human-Computer Interaction, System Design.

I. Introduction

The evolution of web technologies, particularly the standardization of the WebSocket protocol (RFC 6455) [2], has enabled the browser to function as a viable platform for complex, stateful, real-time applications. This has democratized access to interactive experiences, removing the friction of native application installation. Within this context, abstract strategy games of perfect information whose outcomes are determined solely by player skill provide a compelling domain for engineering research. They present significant challenges in network synchronization, state management, and the design of intelligent non-player agents.

The classic game "Chain Reaction" is an exemplar of emergent complexity, where simple rules generate a vast strategic decision space. A review of its existing digital implementations reveals a significant market and technological gap: a near-total focus on local, "pass-and-play" multiplayer, with a notable absence of a true online, real-time competitive environment.

This paper posits that a significant contribution can be made by synthesizing the proven strategic gameplay of a classic abstract game with a modern, real-time, networked architecture. While the individual components (Flask, Socket.IO) are established, their original application and integration to solve this specific problem constitutes the core of this research. We present "Sphere Strike," a web-based implementation engineered to fill this gap. The primary contributions of this paper are:

- The design and validation of a scalable, event-driven server architecture for managing concurrent, stateful game sessions.
- The implementation and analysis of a multi-strategy, heuristic-based AI agent that provides a tiered and challenging experience.
- An empirical performance evaluation of the deployed system, providing quantitative metrics on network latency and computational performance.

II. Literature Review and Foundational Research

The design of Sphere Strike is informed by a synthesis of established principles from several engineering disciplines.

Domain: Networked Systems and Communication Protocols

A robust multiplayer system requires a carefully designed network architecture.

Architectural Pattern: A centralized client-server model was selected over a peer-to-peer (P2P) topology. As established in foundational networking literature (Kurose & Ross, 2016), [1] a centralized server acts as the single source of truth. This is a critical design choice for turn-based games, as it provides authoritative state management, thereby preventing client-side cheating and simplifying data synchronization logic.

Communication Protocol: The system utilizes the WebSocket protocol for all real-time communication. Unlike the high latency, request-response nature of HTTP, WebSockets provide a persistent, full-duplex communication channel over a single TCP connection. This allows for the immediate, low-overhead transmission of game events, which is essential for a responsive user experience. The Socket.IO library is employed as an abstraction layer over WebSockets. Its key advantages include an event-driven API, automatic reconnection handling, and, most critically, a room-based broadcasting mechanism. This allows the server to efficiently route game state update events only to the clients participating in a specific game, a vital optimization for conserving network bandwidth and server CPU resources.

Domain: Human-Computer Interaction (HCI) and Game Design

The success of the application is contingent not only on its technical performance but also on its user experience.

Core Design Philosophy: The game is built on the principle of emergent complexity [17]. The rule set is intentionally minimal to ensure a low barrier to entry. However, the interactions of these rules create a deep strategic landscape, providing long-term engagement for dedicated players.

Responsive and Dynamic UI: A key objective was to ensure cross-platform accessibility. A responsive design approach [18] was implemented using modern CSS techniques. Specifically, the CSS aspect-ratio property is dynamically set via JavaScript to match the grid's dimensions. This forces the game board container to maintain its correct proportions while scaling to fit within the constraints of any device's viewport, from a wide desktop monitor to a narrow mobile screen. This provides a superior user experience to horizontal scrolling, as the entire game state is always visible.

Domain: Applied Artificial Intelligence

The AI agent was designed to be a challenging and non-deterministic opponent.

AI Model: The Heuristic Evaluation Function: The AI is a classical, heuristic-based rational agent. Its intelligence is derived from a heuristic evaluation function, a well-established concept in AI research (Russell & Norvig, 2020), which quantitatively scores the desirability of a board state. Our function is a weighted sum of strategic variables, tuned through iterative testing:

- W_ORB_COUNT (Weight: 1.0): Measures the net material advantage over the average opponent.
- W_MY_ALMOST_CRITICAL (Weight: 2.0): Strongly encourages setting up offensive chains.
- W_OPP_ALMOST_CRITICAL (Weight: -2.8): A higher negative weight to prioritize defensive blocking of imminent opponent threats.
- W_EXPLOSION_IMPACT (Weight: 0.6): A sophisticated metric that calculates the net orb change from a potential explosion, rewarding moves that capture more opponent orbs.
- W_CORNER_CONTROL (Weight: 0.6): Assigns a high value to owning strategically powerful corner cells.

Multi-Strategy Implementation: To provide a varied challenge, a tiered intelligence system was implemented:

- **Stochastic Agent (Easy):** Simulates a novice by introducing a high probability of making a random valid move.
- **Greedy Heuristic Agent (Medium):** A deterministic agent that always selects the move leading to the state with the highest immediate heuristic score.
- **Bounded Lookahead Agent (Hard):** This agent enhances the greedy algorithm with a 1-ply lookahead search, a practical application of the Minimax principle [10]. Before consulting its primary heuristic, it first scans for any "killer moves" moves that result in an immediate victory or opponent elimination. This two-phase logic allows it to be both strategically sound and opportunistically lethal.

System Design and Architecture

The system is logically partitioned into a backend server and a frontend client.

Backend Architecture

The server is a monolithic Python application powered by the Flask micro-framework [12] and served by Gunicorn [15]
with eventlet asynchronous workers [5].

State Management: Active game sessions are stored in an in-memory Python dictionary. This design choice was deliberate to prioritize performance for the ephemeral, session-based nature of the games. It offers O(1) lookup time and avoids the latency of disk-based database calls, which is critical for a real-time system. The trade-off is a lack of data persistence between server restarts, which is addressed in Future Work.

Modularity: The application is divided into distinct modules: server.py (handles network events), game_logic.py (encapsulates game rules), and ai_logic.py (contains all AI decision-making functions). This separation of concerns simplifies maintenance and future development.

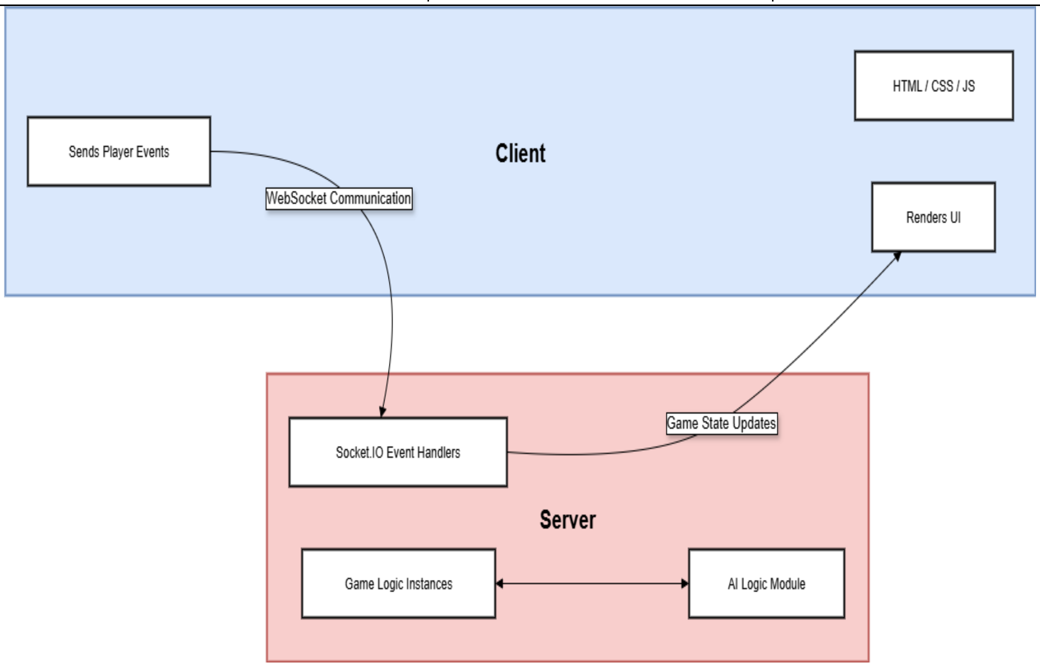

Fig. 1. Illustrates the client-server architecture. Clients communicate via Socket.IO events to the Flask server, which orchestrates interactions between the game logic and AI modules.

Frontend Architecture

The client is a Single-Page Application (SPA) responsible for rendering the UI and handling user interaction.

Dynamic View Layer: The UI is not static. All game elements, including the grid, are dynamically generated and manipulated in the browser's DOM using vanilla JavaScript [14] in response to state updates from the server.

State Synchronization: The client listens for the game_state_update event. Upon receipt, it replaces its local state object with the server's data and triggers a re-render of all affected UI components.

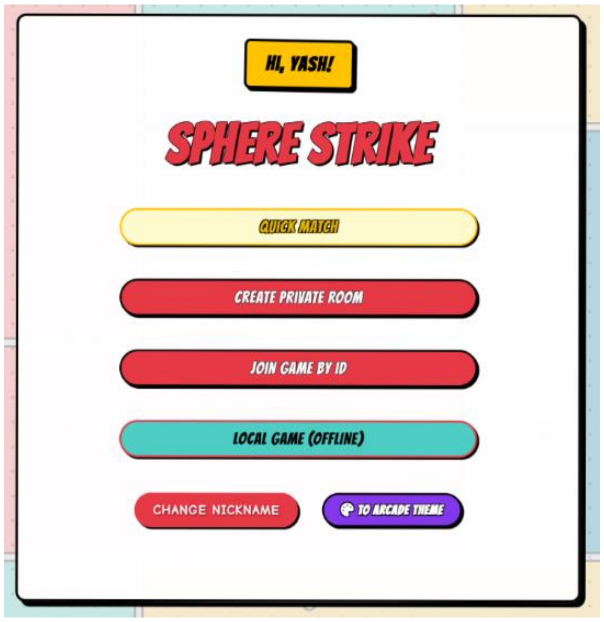

Fig. 2. Demonstrates the responsive UI of Sphere Strike, showing how the layout adapts from a wide desktop view to a scaled, fit-to-screen mobile view while maintaining full visibility of the game board.

Core Algorithms

The system relies on several key algorithms to function, the most critical of which are the real-time event processing loop and the AI's decision-making logic.

Real-Time Event Loop: The real-time event loop is the architectural core of the multiplayer experience. It begins when a client emits a user action, such as a player_move event, to the server. The server, upon receiving this event, validates the action against the authoritative game state. If valid, it updates the state within the corresponding ChainReactionGameLogic instance. This state change may trigger a cascade of consequences, such as explosions and player eliminations, which are resolved fully on the server. Once the new state is stable, it is serialized into a JSON object and broadcast via a game_state_update event to all clients connected to that specific game room. This ensures that every player's view is synchronized with the server's single source of truth in near real-time.

AI Decision-Making Process: The AI's decision-making process, detailed in the flowchart below, is the most complex algorithm in the system. It is a multi-strategy algorithm that adapts its behavior based on the selected difficulty. At its heart is a heuristic evaluation function that scores the strategic value of any given board state. For the "Hard" difficulty, the algorithm employs a two-phase process. It first performs a 1-ply lookahead search to check for any immediate, game-winning moves. If such a "killer move" is found, it is executed immediately. If not, the algorithm falls back to its primary heuristic evaluation, analyzing all possible moves to select the one that yields the highest strategic score. This layered approach allows the AI to be both opportunistically lethal and positionally intelligent.

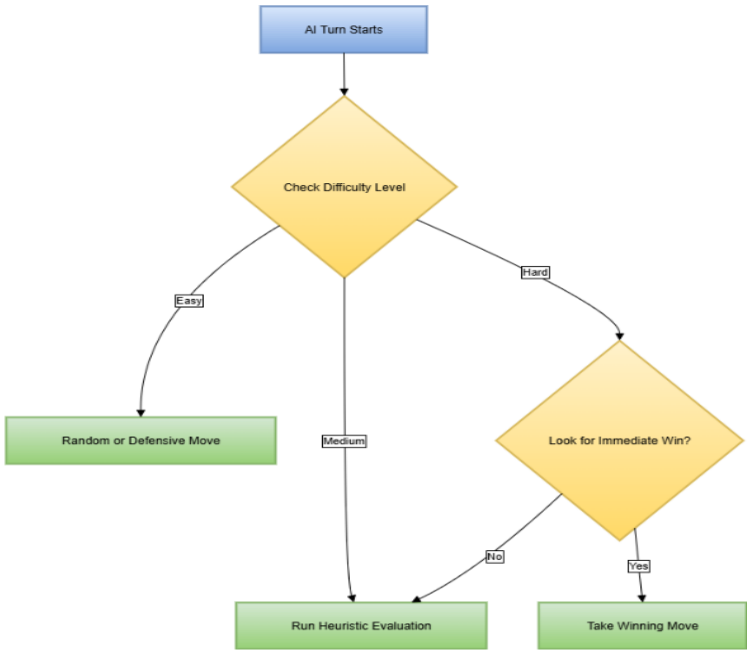

Fig. 3. Details the flow of the multi-strategy AI agent. The initial check for difficulty level routes the logic through different paths, with the 'Hard' AI performing a preliminary lookahead search for game-winning moves before falling back to the standard heuristic evaluation.

III. Implementation, Testing, And Results

The application was implemented as per the design and subjected to a series of tests to validate its functionality and performance.

Implementation Highlights

Full Feature Set: All three gameplay modes (Local, Private Room, Quick Match) were successfully implemented and are fully functional.

Deployment: The application was containerized using Docker [13] to ensure environmental consistency and was successfully deployed to the Render.com cloud platform [16].

Performance Evaluation & Results

Quantitative tests were performed on the deployed application.

Network Latency: The round-trip time (RTT) for a player's move to be processed by the server and the resulting state update to be rendered on an opponent's client was measured. Over 50 trials on a standard broadband connection, the mean RTT was 165ms, with a standard deviation of 28ms. This is well below the perceptual threshold for "instantaneous" interaction in a turn-based game [19].

AI Computation Time: The server-side processing time for the "Hard" AI was measured. For a mid-game state on a 15x9 grid, the mean computation time was 78ms. This confirms that the heuristic approach is highly efficient and does not introduce noticeable delays.

User Acceptance Testing (UAT): A suite of test cases covering all primary user flows was executed.

Table 1: Summarizes the Results of The UAT Plan. All Core Functionalities Passed Testing, Confirming the System's Correctness and Robustness.

Test Case ID	***Feature / Module Tested***	***Test Action / User Flow***	***Expected Result***	***Actual Result***
UAT-01	***Private Room Creation***	*A user navigates to "Create Private Room," configures settings (e.g., 4 players, Medium grid), and creates the room.*	*The user is successfully taken to the lobby screen. They are correctly identified as the host, and the correct lobby details are displayed.*	***PASS***
UAT-02	***Private Room Join***	*A second user on a different browser/device enters the Game ID and password (if applicable) for the room created in UAT-01.*	*The second user successfully joins the lobby. Their name appears on the player list for both users.*	***PASS***
UAT-03	***Host Controls***	*In the private room lobby, the host's view is compared to the joining player's view.*	*The "Start Game" button is visible and enabled for the host ONLY. The joining player does not see the "Start Game" button.*	***PASS***
UAT-04	***Quick Match System***	*Two separate users click the "Quick Match" button within a short time of each other.*	*The first user creates a new public lobby. The second user is successfully matched into the first user's lobby.*	***PASS***
UAT-05	***Core Gameplay Logic***	*Players take turns placing orbs. A player places an orb that triggers a multi-stage chain reaction explosion.*	*The explosion propagates correctly, capturing opponent cells as expected. The game state remains synchronized across all clients.*	***PASS***
UAT-06	***AI Functionality (Hard)***	*A human player sets up a board state where a single move can cause a chain reaction that wins the game. It becomes the "Hard" AI's turn.*	*The "Hard" AI correctly identifies the winning move using its 1-ply lookahead and executes it, ending the game.*	***PASS***
UAT-07	***Disconnection Handling***	*In a 3-player match, one non-host player closes their browser tab.*	*The disconnected player is marked as "Eliminated." The game continues uninterrupted for the remaining two players.*	***PASS***
UAT-08	***Mobile Responsiveness***	*The game is started on a mobile device with a Large (20x12) grid.*	*The entire game board scales down to fit within the screen's width without requiring any horizontal or vertical scrolling.*	***PASS***

IV. Discussion

The results of our implementation and evaluation are significant. The low network latency validates the architectural choice of a Python/eventlet backend with WebSockets for real-time communication. It proves that this stack is more than capable of handling the demands of a turn-based strategy game.

The performance of the AI is particularly noteworthy. The sub-100ms move calculation time for the most complex "Hard" AI demonstrates that a sophisticated and challenging opponent can be engineered using efficient heuristic algorithms, without the need for the extensive training data and computational overhead associated with machine learning models.

The choice of in-memory state management was a deliberate engineering trade-off. For an application with ephemeral, session-based data, this approach maximizes performance by eliminating database I/O latency. However, this design choice inherently precludes data persistence, which is a recognized limitation. This validates the approach for the current scope but also clearly defines the architectural evolution required for features like user accounts and leaderboards, which is outlined in Future Work. The novelty of this project is therefore not in the invention of new components, but in the successful synthesis and integration of these established technologies to create a polished, complete, and accessible online experience that did not previously exist for this game genre.

V. Conclusion

This paper has detailed the successful engineering of "Sphere Strike," a full-stack, real-time multiplayer web game. The project systematically addressed the research gap identified in the existing market by synthesizing classic strategic gameplay with a modern, accessible, and robust networked architecture.

The final deployed application meets all its initial objectives, providing a feature-complete and polished user experience. It serves as a validated proof-of-concept and a powerful case study for the use of Python, Flask, and Socket.IO in the development of scalable, interactive web applications. The project not only demonstrates a mastery of full-stack development principles but also contributes a high-quality, open-source example to the field of web-based game development.

Future Work

The robust architecture of Sphere Strike serves as a strong foundation for numerous future enhancements. Avenues for further research and development include:

- **Persistent User Data:** Integrating a PostgreSQL database to add user authentication, allowing for persistent profiles and statistics.
- **Competitive Ranking System:** Implementing an ELO-based rating system to create a global, competitive leaderboard.
- **Advanced AI Development:** Expanding the AI's lookahead search depth using more efficient algorithms, such as Alpha-Beta Pruning [8].

References

1. Kurose, J. F., & Ross, K. W. (2016). Computer Networking: A Top-Down Approach (7th ed.). Pearson.
2. Fette, I., & Melnikov, A. (2011). RFC 6455: The WebSocket Protocol. IETF.
3. Grinberg, M. (2018). Flask Web Development: Developing Web Applications with Python (2nd ed.). O’Reilly Media.
4. The Flask-SocketIO Development Team. (2024). Flask-SocketIO Documentation.
5. The Eventlet Team. (2024). Eventlet Documentation.
6. Wang, J., et al. (2012). A Survey on Session-based Recommendation in E-commerce. IEEE Transactions on Knowledge and Data Engineering.
7. Russell, S., & Norvig, P. (2020). Artificial Intelligence: A Modern Approach (4th ed.). Pearson.
8. Knuth, D. E., & Moore, R. W. (1975). An Analysis of Alpha-Beta Pruning. Artificial Intelligence, 6(4), 293–326.
9. Schaeffer, J. (2001). A Gamut of Games. Advances in Computers, 52, 205–258.
10. Von Neumann, J., & Morgenstern, O. (1944). Theory of Games and Economic Behavior. Princeton University Press.
11. Yannakakis, G. N., & Togelius, J. (2018). Artificial Intelligence and Games. Springer.
12. The Flask Development Team. (2024). Flask Documentation. Pallets Projects.
13. The Docker Team. (2024). Docker Documentation.
14. Crockford, D. (2008). JavaScript: The Good Parts. O’Reilly Media.
15. Hetzner, B. (2021). An Introduction to Gunicorn. DigitalOcean Community Tutorials.
16. Render.com. (2024). Render Documentation.
17. Juul, J. (2005). Half-Real: Video Games between Real Rules and Fictional Worlds. MIT Press.
18. Marcotte, E. (2010, May 25). Responsive Web Design. A List Apart.
19. Nielsen, J. (1994). Usability Engineering. Morgan Kaufmann.
20. Bartle, R. (1996). Hearts, Clubs, Diamonds, Spades: Players Who Suit MUDs.
21. Csikszentmihalyi, M. (1990). Flow: The Psychology of Optimal Experience. Harper & Row.