ChatGPT

Advanced Tic-Tac-Toe System Design

Problem Statement

Design an **online Tic-Tac-Toe platform** with rich features and high scalability. The system should support **dynamic board sizes** (e.g. 3×3, 4×4, up to N×N) with configurable win conditions (e.g. K in a row horizontally/vertically/diagonally) [1] [2]. It must allow **multiple simultaneous games** and players: include a **matchmaking** mechanism to pair players into game sessions and handle thousands of concurrent games with low latency [3]. The platform should support all play modes: **human vs. human**, **human vs. AI**, and **AI vs. AI**, with the AI offering multiple **difficulty levels** (for example, easy = random move, hard = strategic algorithm) [4] [5].

The game logic must be **extensible** – e.g. a flexible rules engine to allow custom game variants or rule changes at runtime [6]. In addition, implement **persistent game state** so players can **save and resume** games, and maintain **user accounts and profiles** with stats/leaderboards (wins, losses, win rates) [4]. Finally, sketch a **scalable backend architecture** (e.g. RESTful APIs or WebSockets) using appropriate techniques (load balancing, caching, microservices or event-driven components) to reliably serve thousands of concurrent players. (Candidates may also discuss client-server interaction design or UI considerations as applicable.)

Assumptions

- **User Accounts**: Assume players have registered accounts (with profiles, authentication) before playing.
- **Session Management**: Each game session is uniquely identified and managed (e.g. via a game ID).
- **Communication**: You may choose RESTful HTTP or WebSocket protocols for real-time play. For example, WebSockets (with a message broker like Redis Pub/Sub) can sync game state across clients [7].
- **Datastores**: Use appropriate storage (SQL/NoSQL) for persistence. For instance, a relational DB for user accounts/stats and an in-memory store (Redis or similar) for active game state (with periodic snapshots for persistence).
- **Scalability Targets**: Design for on the order of 10,000+ concurrent active games and users.
- **AI Implementation**: AI logic runs server-side. Easy AI might choose a random empty cell [5]; harder AI might use minimax or learning.

Candidate Expectations

A strong design should address the following aspects:

- **Core Components**: Identify major modules/services. For example:
- *Game Service*: Manages game sessions, game state (board, moves, turn order) and win/draw detection. Implement efficient win checking (e.g. maintain counters per row/column/diagonal for O(1) move processing) [1] [8].
- *Matchmaking Service*: Pairs players (human or AI) into new games, handles queueing or invites.
- *Player Service*: Manages user profiles, authentication, and stats tracking (wins/losses).

- *AI Service*: Provides opponent moves. Implement multiple strategies (random, heuristic/minimax, etc.) and selectable difficulty levels [5].
- *Rules Engine*: Encapsulate game rules to allow variants. For example, use a Strategy or Plugin pattern so the win condition (K in a row) and board size can be configured at runtime [6].
- *Persistence Layer*: Design how to store game history (moves, final result), saved games, user data. Perhaps a database schema for Users, Games, Moves, and Stats.
- *API Layer*: Define REST endpoints or WebSocket events for clients to create/join games, make moves, query game state, pause/resume, etc.

- *Notification / Real-Time Sync*: If using WebSockets, describe how server pushes game updates to clients (e.g. using Socket.IO) and how multiple servers stay in sync (e.g. Redis Pub/Sub as shown in practice [7]).

- **Data Modeling**: Propose data structures/classes for Game, Board, Player, Move, etc. Follow SRP: e.g., a `Board` class to store the grid and validate moves, a `GameController` to manage turn flow and check game state [9]. Show how these models support N×N boards and K-in-row wins.

- **Concurrency & State Management**: Explain how to handle concurrent moves and avoid race conditions (e.g. use locking per game session, optimistic concurrency, or atomic updates). Ensure one-player's move cannot conflict with another's. Discuss session timeouts or disconnection handling.

- **Matchmaking & Multiplayer**: Detail how players find opponents (random pairing, friend invites). Discuss lobby design or queuing system. If supporting spectator mode or game lobbies, mention that too.

- **AI Design**: Describe at least one simple AI algorithm (e.g. random picker [5]) and one advanced (e.g. minimax for perfect play). Explain how you would scale AI computations (e.g. precomputed moves, difficulty scaling by search depth).

- **Rules Extensibility**: Describe how to make the rule set pluggable. For example, a rule interface with implementations for standard tic-tac-toe vs variants (different win conditions, board shapes). The design should *not* hardcode a 3×3 grid or "3-in-a-row" rule [2] [6].

- **Persistence**: Explain how to save and resume games. For instance, when a player quits, serialize the game state to the database, and on resume reload it. Also persist completed game results for stats/leaderboards. Mention backups or redundancy if relevant.

- **Scalability & Architecture**: Propose how to support high load: e.g. multiple app servers behind a load balancer, stateless or distributed game logic, caching of active games (Redis) [7], database sharding/replication. Consider whether to use a microservices architecture (e.g. separate services for game logic, user management, matchmaking) or a modular monolith. Discuss where an event-driven approach (message queues) might help (e.g., broadcasting "GameOver" events to update stats asynchronously).

- **APIs and Client Interaction**: Outline key API endpoints or message flows. For example, `POST /games` to create/join a game, `POST /games/{id}/move` to make a move, websockets to receive opponent moves. Consider security and validation (only the correct player can play their turn).

- **Performance Considerations**: Address efficient winner-checking for large boards (e.g. using counters [1]), minimizing latency (real-time updates), and load testing aspects.

- **Trade-offs and Edge Cases**: Comment on design trade-offs (e.g. consistency vs. responsiveness, eventual consistency for non-critical stats). Handle edge cases: invalid moves, stale sessions, network failures, draw conditions, simultaneous game joins, etc.

Evaluation Criteria

Submissions will be assessed on the following:

- **Requirement Coverage**: All specified features (board variants, multiplayer modes, AI, persistence, scalability) are addressed. For example, did the design allow arbitrary N×N boards and flexible win rules [1] [2]?

- **Modularity and Clarity**: Clean separation of concerns. Each class/service should have a single responsibility (e.g. GameController vs Board vs Player [9]). Avoid one "god object" or mixing UI with logic [10]. Diagrams or bullet lists should clearly show component interactions.

- **Extensibility**: Ease of adding new features (different game variants, new AI strategies, additional game rules). Designs that use patterns (Strategy for AI, State or Factory for game rules) are looked upon favorably. The solution should not hardcode values (like board size) but allow configuration [2] [8].

- **Scalability and Performance**: Viability at high load. Appropriate use of caching (e.g. Redis for active games [7]), database indexing, load balancing, etc. Consideration of concurrency control and throughput. Efficient algorithms (e.g. O(1) win check) are a plus.

- **Technical Depth**: Use of suitable technologies and patterns (e.g. microservices vs monolith choice, event-driven design for game events, real-time comms with WebSockets). Citing known approaches (e.g., WebSockets + Redis Pub/Sub for real-time sync [7]) shows awareness of industry practices.

- **Completeness and Thoughtfulness**: The answer should not only list components but explain interactions, data flow, and reasoning for design decisions. Anticipation of edge cases (e.g. handling disconnects, draw detection) and articulate trade-offs is expected.

- **Presentation**: The design should be communicated clearly and systematically. Clean diagrams or structured descriptions that make it easy to follow how the system works end-to-end.

Good designs will have **clear responsibilities** for each module, **clean state management**, and **optimized logic** (e.g., incremental win tracking) [10]. They will also demonstrate **easy extensibility** to new requirements [10]. Red flags include monolithic code (everything in one class), mixing UI with backend logic, or failure to consider concurrency and invalid inputs.

---

[1] [5] Design Tic-Tac-Toe and QPS data structures | Databricks Interview Question
https://prachub.com/interview-questions/design-tic-tac-toe-and-qps-data-structures

[2] [4] [6] TicTacToe/discussion.md at main · scaleracademy/TicTacToe · GitHub
https://github.com/scaleracademy/TicTacToe/blob/main/discussion.md

[3] Systems Design 2 — TicTacToe. An example where Pseudo-Code gets you… | by Alexander Knips | Medium
https://medium.com/@alexanderknips/systems-design-2-tictactoe-c145ca27cc4f

[7] Build a Real-Time Multiplayer Tic-Tac-Toe Game Using WebSockets and Microservices
https://www.freecodecamp.org/news/build-a-real-time-multiplayer-tic-tac-toe-game-using-websockets-and-microservices/

[8] [9] [10] Designing Tic Tac Toe: A Low-Level System Design Interview Favorite | by Anuragkumbhare | Jan, 2026 | Medium
https://medium.com/@anuragkumbhare2043/designing-tic-tac-toe-a-low-level-system-design-interview-favorite-b058104e51b5