TicTacToe AI

Document Version: 1.1
Last Updated: 2025-08-14

Table of Contents

Project Description

A web-based TicTacToe game where a human player competes against a state-of-the-art neural network AI that learns and improves through gameplay. All AI computation occurs on the backend, with the client serving as a UI layer. The system supports multiple simultaneous games with scalability for up to 100 concurrent connections. The AI uses only neural networks with no symbolic rules and only knows the rules from the start (i.e., it learns all strategy from gameplay).

Non-Goals

- No hard-coded strategic rules beyond basic mechanics (valid moves, turn order, win/draw).
- No user accounts in MVP (anonymous play); accounts may be introduced later.
- No blockchain or external leaderboards in MVP.

Glossary

- EMA (Exponential Moving Average): Rolling metric emphasizing recent outcomes; configurable N (default 3).
- Acquiesced/Disabled Learning: Session flag excluding its data from global learning/stats.
- Self-Play: AI vs AI games under compute quotas and iteration caps.

Target Audience

- AI enthusiasts interested in observing machine learning in action
- Casual gamers wanting an evolving opponent
- Educators demonstrating neural AI approaches
- Anyone curious about neural artificial intelligence
- Researchers exploring reinforcement learning and learning-from-scratch dynamics

Desired Features

Game Mechanics

- Standard 3x3 TicTacToe gameplay
- Human vs AI matches
- Human chooses who goes first (X or O)
- First play alternates between human and AI
- Clear win/loss/draw detection
- Game history tracking
- Spectating: Allow users to observe ongoing games in real-time without participating (available by default for all games)
- Offline Mode: Fallback to local random-move AI when disconnected
- Human vs Human: Optional local or online matches via rooms; AI may spectate/learn if enabled
- Tutorial Mode: Guided onboarding explaining rules, turns, and AI behavior
- Undo/Redo: Available in casual games; disabled in competitive modes

Neural AI Features

- Initial State: Starts with ZERO strategic knowledge - only basic game mechanics
- Pure Learning Approach: All strategy discovered through gameplay against humans or an AI robot (self-play)
- Pattern Recognition: State-of-the-art neural network (lightweight CNN with residual blocks) learns from board states, move sequences, and outcomes
- Multi-Level Learning: At move and game levels, using advanced RL algorithms like PPO
- Continuous Learning: Updates neural weights after every move for real-time adaptation (all handled on backend, with optimizations for speed); learns across all games by default unless learning has been acquiesced/disabled for specific sessions
- Performance Tracking: Displays AI thinking time (computed and sent from backend); tracks loss ratio using a selectable N-game exponential moving average (EMA) with recency bias alpha=0.3 (default N=3, max 20) to show learning progress (given perfect play results in draws); excludes acquiesced/disabled sessions from stats; handles initial games with SMA fallback and note; smooths anomalies with thresholds; weights draws neutrally with secondary draw ratio metric
- Acceleration Techniques: Experience replay, self-play (capped at 500 iterations for gradual improvement), advanced optimizers for faster and more complete learning; optional GPU acceleration
- Difficulty Levels: Toggle for beginner (random, high target loss ratio ~40-50%), learning (variable), or advanced (pre-trained, low target loss ratio ~0%) modes; each with configurable target loss ratios
- Reset Capability: Ability to reset the neural net to initial state (no knowledge) for any specific robot or difficulty level
- Model Versioning: Register and tag checkpoints with metadata (date, games played, loss ratios)
- Export/Import: Download and restore model weights for sharing and backups
- Reproducibility: Seed management for deterministic evaluation runs
- Safeguards: Compute quotas and monitoring around self-play to prevent resource exhaustion or overfitting
- Explainability: Optional brief rationale or board saliency hints (non-binding, educational)

Data Storage

- MongoDB for neural weights and game history
- Redis for sessions, real-time state, and EMA caching
- MongoDB also for frontend and backend logging data
- Indexes for `gameId`, `sessionId`, timestamps; include migration plan

- Retention policies: Logs 30 days by default (configurable)
- Backups: Automated daily backups for MongoDB/Redis with restore runbooks
- Privacy: Anonymize game data; opt-out/erasure workflows

User Interface

- Responsive design: Mobile-first, tablet, desktop support
- Game Board: Touch-friendly grid with animations
- AI Insights Panel: Shows neural evaluations, decisions, thinking time (fetched from backend; collapsible on mobile)
- Learning Dashboard: Charts of AI progress (e.g., win rate over games, real-time N-game EMA loss ratio displayed numerically and graphically vs. games played using Chart.js line charts with tooltips; selectable N defaulting to 3, persistent via local storage with presets and color-coding; data synced from backend; reset button for recalculations); alerts for learning milestones shown as informative, non-intrusive banners (e.g., 0% loss = 'Expert', <20% loss = 'Advanced', <40% loss = 'Intermediate', <70% loss = 'Beginner', else 'Novice'; triggered on improvements)
- Spectating Mode: Read-only view of game state, with join/leave functionality (default for all games)
- Lobby: Global list of public games for easy spectating/joining
- Optimistic Updates: Client-side prediction for smoother UX
- Accessibility: WCAG 2.1 AA (ARIA grid roles, keyboard nav, contrast), screen reader support
- Theming: Light/dark themes; sound/haptics toggles
- Robust Error States: Disconnect/retry flows, offline indicators, overload and invalid input messaging
- Internationalization: i18n scaffolding

Technical Requirements

- Real-time updates during gameplay via Socket.IO (built on WebSockets)
- Efficient neural inference (backend-only, using latest optimizations)
- Mobile performance optimization (lightweight client)
- Architecture:
  - Backend handles all AI (inference, training, updates) with state-of-the-art tech for rapid learning
  - Frontend communicates via Socket.IO for state sync, moves, and AI data (using events, rooms, and broadcasts)
  - Support for multiple simultaneous games via Socket.IO rooms or namespaces
  - Scalability for up to 100 concurrent connections (soft target with queuing for overflows), with horizontal scaling on AWS if needed; rate limiting and JWT authentication to prevent abuse
  - Spectating via room joining in observer mode, with role-based ac-

    cess control (available by default for all games, with optional privacy toggle); no login required, rejection messages for caps, includes in connection count
  - Shared learning across all games unless acquiesced/disabled (e.g., via per-game flag to skip updates from that session); global progress display with per-session views
- Security: Enforce HTTPS/WSS, strict CORS, input validation for all events, JWT-based optional identity, CSRF for REST, secret management
- Networking: Heartbeats, exponential backoff on reconnect, acks with timeouts, idempotent move handling with nonces
- Deployment: Dockerized services; environment configuration via env vars/SSM; CDN for static assets

Testing Requirements

- Unit, integration, E2E testing with 90%+ coverage (using Jest for unit/integration, Cypress for E2E)
- AI-specific testing for learning progression, convergence, performance, and benchmarks against baselines (including shared vs. acquiesced/disabled scenarios, EMA loss ratio calculations with variable N, target ratios, and edge cases like small samples/anomalies)
- Comprehensive network testing for Socket.IO reliability and sync
  - Note that this is a particularly difficult area to get working correctly and particular attention must be paid to this area both in development and testing
- Load testing for concurrent games and scalability (e.g., simulate up to 100 connections using Artillery)
- Spectating-specific tests for observer sync, join/leave, non-interference, and default availability
- UI testing for milestone banners (triggers, non-intrusiveness, accessibility)
- UI testing for AI dashboard (e.g., Learning Dashboard interactions, charts, alerts, and responsiveness)
- Contract tests for event schemas and acks; negative-path tests for invalid moves
- Security testing (OWASP ZAP/Snyk) and dependency vulnerability scanning
- Cross-browser/device matrix tests (mobile, throttled networks, offline)
- Chaos testing: server restarts, socket drops, packet loss; verify graceful recovery
- Performance SLO tests with pass/fail thresholds

Unified Logger

- The goal is to have all debug logging in one place. This should minimize having to look at the browser console as all logs should be routed to the server where they are placed in the database.

- Logs are stored in a MongoDB
- Server logging is included in the unified logger.
- Client logging is also included in the unified logger.
- Ability to select level of logging.
- Ability to clear log db.
- Structured logs with correlation IDs (gameId, sessionId, requestId)
- Runtime-configurable levels and sampling to reduce noise
- Client-side pre-filtering before send to limit bandwidth
- Export logs (CSV/JSON) for analysis; retention aligned with data policy

Start / stop Scripts

- there should be 3 scripts: 1) start server in foreground, 2) start client in foreground, and 3) start both in background.
- There should be a single stop script that stops all occurences of both the server and client
- Start scripts should begin by killing all processes currently running for that function (either server or client)
- Idempotent, with explicit port cleanup (e.g., 3001, 5173)
- Graceful shutdown with timeouts; persist in-flight games safely
- Background process manager (e.g., PM2) for auto-restart and log rotation
- Health checks and readiness probes; PID files for precise stop targeting

Non-Functional Requirements (SLOs)

- p95 AI inference latency 300ms per move under load; p95 end-to-end move RTT 500ms
- Support 100 concurrent active connections with stable behavior (soft cap with queuing)
- Target availability 99.9% during active play windows
- Browser/device support matrix: Latest Chrome/Firefox/Edge, Safari 17+; iOS 16+/Android 11+

Technical Stack

Frontend

- React with TypeScript, Redux Toolkit
- Socket.IO client (˜4.x) for real-time (with namespaces, acknowledgments, heartbeats/auto-reconnect, authentication plugin, client-side prediction, HTTP polling fallback)
- Tailwind CSS, Headless UI, Framer Motion
- Chart.js (with react-chartjs-2) for graphs
- ESLint, Prettier; Husky + lint-staged; TypeScript strict mode

Backend

- Node.js for game server with Socket.IO (^4.x) for real-time communication
- PyTorch (latest version) for neural AI (server-side only), integrated with RL libraries like Stable Baselines3 or Ray RLlib
- Async queues (e.g., Bull.js) for concurrent updates
- Express.js for RESTful endpoints (health, stats, exports)
- Node.js v20+

Database

- MongoDB primary, Redis cache

CI/CD and Deployment

- GitHub Actions: lint/typecheck/tests, E2E, security scans, Docker build/publish
- Staging and production environments; blue/green or canary deploys with rollback
- Docker Compose for local dev; AWS ECS/EKS for scaling; Redis persistence configuration
- Secrets via AWS SSM/Secrets Manager (or equivalent)

Observability

- Metrics (Prometheus): move latency, error rates, socket reconnects, queue depth
- Tracing (OpenTelemetry) across move lifecycle (client intent → server process → model inference → emit)
- Dashboards and alerting for SLO breaches and model drift

Reliability and Networking

- Reconnect/backoff strategies; idempotent move handling with server-side deduplication
- Backpressure policy with bounded queues; load shedding on overload
- Graceful shutdown with in-flight state persistence and replay on restart

Security and Privacy

- Enforce HTTPS/WSS; strict CORS and content security policy
- JWT-based optional identity; rate limiting and abuse detection (spam rooms, rapid reconnects)
- CSRF for REST; input validation for all payloads; schema validation
- PII minimization/redaction in logs; data opt-out and erasure workflows
- SBOM generation and dependency vulnerability scanning

AI/ML Operations

- Model registry/versioning with metadata; artifact storage
- Baseline evaluation suite (vs random and perfect play) with tracked scores
- Drift detection and alarm thresholds; scheduled evaluation runs
- Compute budgets/quotas for training and self-play; kill switches

Other Notes

- Use Socket.IO for structured real-time sync between client and server (events, rooms, reconnection), with support for multiple concurrent games (up to 100 connections), spectating (default for all games), and shared AI learning unless acquiesced/disabled
- Comprehensive testing before advancing bewteen phases, with focus on custom sync robustness, concurrency, scalability, spectating features, learning modes (including real-time EMA loss ratio tracking with selectable N, milestone alerts via banners, first-play turn alternation, and handling of average issues like small samples/anomalies), and exclusion of acquiesced data.
- Potential challenges: Socket.IO configuration for scalability, network latency for AI moves and real-time metrics/alerts, backend resource management with concurrent loads (up to 100), compute demands for advanced RL, handling acquiesced/disabled learning without affecting shared model integrity, privacy for default spectating, optimizing EMA with variable N for performance, true unified logging, alternating first-play between human / AI, and accuracy (e.g., caching, fallbacks, smoothing)
- Cross-reference: See `docs/implementation_plan.md` (for step tracking) and `docs/notes.md` for implementation details and decisions.
- License: Adopt an open-source license (e.g., MIT) to enable collaboration.

Version Control Requirements

- Use Git for version control from the beginning of the project.
- Initialize a local Git repository in the project root.
- Create and maintain a GitHub repository for the project (e.g., github.com/username/tictactoe-ai).
- Commit changes regularly with meaningful messages.
- Push commits to GitHub after major updates or milestones.
- Use branches for feature development (e.g., main for stable, develop for ongoing work).
- Include a .gitignore file to exclude unnecessary files (e.g., node_modules, .DS_Store).
- Adopt Conventional Commits; require PRs with templates and CODEOWNERS; mandatory CI checks
- Branching: Trunk-based with short-lived feature branches preferred

Legal and Governance

- CONTRIBUTING.md to guide external contributions
- Privacy notice outlining data collection, retention, and opt-out for learning

Future Enhancements

- Private rooms, invites, and simple matchmaking
- Push notifications or email alerts for game invites (optional)
- Native mobile wrappers using React Native as pure UI clients