CollabBoard Pre-Search Document

Author: Jorge Alejandro Diez
Date: February 16, 2026
Project: Week 1 — Real-Time Collaborative Whiteboard with AI Agent

---

Phase 0: Cursor & Claude Code Configuration

Before touching any project code, the development environment must be configured to maximize AI-first output quality. This phase establishes the "context boundaries" that prevent hallucination, enforce consistency, and keep the coding agent aligned with our architectural decisions throughout the sprint.

0.1 — Cursor Setup

Model Selection Strategy

Task Type	Model	Rationale
Boilerplate, file scaffolding, simple CRUD	Composer 1.5	Cheapest, fast, sufficient for routine code
WebSocket logic, CRDT integration, canvas rendering	Claude Sonnet 4.5	Strong at complex multi-file reasoning
Debugging sync race conditions, AI agent tool-calling	Claude Opus	Reserve for "fickle" bugs and architectural decisions
Quick code review, linting suggestions	GPT-4o / Codex Mini	Fast turnaround, low cost

Turn off "Auto" model selection immediately. Manually choose models per task to control cost and quality.

Workspace Configuration Checklist

- [ ] Open Agent Chat (`Cmd+L`), Terminal, and File Explorer as the triple-view default
- [ ] Verify codebase indexing status in `Cursor Settings > Indexing` — must show "Indexed"
- [ ] Disable Max Mode unless debugging a critical sync bug
- [ ] Check context usage indicator (bottom bar) regularly to avoid overflow

External Documentation to Index

Add these URLs in `Cursor Settings > Features > Docs`:

- Konva.js: `https://konvajs.org/docs/`
- Yjs CRDT: `https://docs.yjs.dev/`
- y-websocket: `https://github.com/yjs/y-websocket`
- Socket.io: `https://socket.io/docs/v4/`
- Supabase Auth: `https://supabase.com/docs/guides/auth`
- Supabase JS Client: `https://supabase.com/docs/reference/javascript/introduction`
- OpenAI Function Calling: `https://platform.openai.com/docs/guides/function-calling`
- Anthropic Tool Use: `https://docs.anthropic.com/en/docs/build-with-claude/tool-use`
- Zustand (state management): `https://docs.pmnd.rs/zustand/getting-started/introduction`
- Vitest: `https://vitest.dev/guide/`

0.2 — .cursorrules (Non-Negotiable Project Rules)

Create `.cursor/rules/` directory with the following rule files:

`tech-stack.mdc` — Enforces version pinning:

```
Description: Enforce project tech stack versions
Always: true

- Framework: Next.js 15 (App Router)
- Language: TypeScript (strict mode)
- Canvas: Konva.js + react-konva
- Real-time sync: Yjs (CRDT) + y-websocket for board state
- Real-time transport: Custom WebSocket server (Socket.io or y-
websocket provider)
- Ephemeral sync: Socket.io broadcast for cursor positions (not
persisted)
- State: Zustand for local UI state, Yjs shared document for
board objects
- Auth: Supabase Auth (magic link + email/password)
- Database: Supabase PostgreSQL (persistence layer — NOT the
real-time transport)
- Styling: Tailwind CSS v4
- Testing: Vitest for unit/integration
- Deployment: Vercel (frontend) + Railway or Render (WebSocket
server) + Supabase (DB/Auth)
- AI: OpenAI GPT-4o-mini with function calling
```

`tdd.mdc` — Enforces test-driven development:

```
Description: Test-Driven Development methodology
Always: true
```

```
Before implementing any feature:
1. Write the test file first (Vitest for unit and integration)
2. Confirm the test fails as expected
3. Implement the minimum code to pass the test
4. Refactor only after tests pass

For WebSocket/real-time features:
- Write integration tests that simulate 2+ concurrent Yjs
documents
- Test disconnect/reconnect by destroying and recreating the
WebSocket provider
- Test conflict resolution by applying concurrent updates to the
same Yjs map key
- Verify persistence by snapshotting Yjs state, clearing memory,
and restoring from snapshot

For canvas operations:
- Test object creation returns correct properties from the Yjs
shared map
- Test state persistence after simulated refresh (reload Yjs doc
from snapshot)
- Test coordinate transforms for pan/zoom
```

`code-patterns.mdc` — Enforces architectural consistency:

```
Description: Code patterns and conventions
Always: true

- All API routes go in /app/api/
- All canvas components go in /components/board/
- All real-time logic goes in /lib/realtime/
- All AI agent logic goes in /lib/ai-agent/
- All types go in /types/
- Single Zustand store in /stores/board-store.ts for local UI
state (selected tool, zoom, pan)
- Yjs shared document is the single source of truth for board
objects — never duplicate into Zustand
- Every function must have TypeScript return types
- No `any` types — use `unknown` with type guards
- Error handling: always use try/catch with typed errors
- Console.log only in development — use structured logging for
production
- WebSocket messages for cursors must follow a typed event schema
- All board objects must have: id, type, x, y, width, height,
createdBy, updatedAt
- Cursor position events must be throttled to 20-30Hz on the
sender side
```

`ai-agent.mdc` — Enforces AI tool-calling patterns:

```
Description: AI Agent development patterns
Always: true
```

```
- AI agent tools must follow OpenAI function-calling schema
- Each tool must have: name, description, parameters (JSON
Schema)
- Tool responses must include success/failure status and the
affected object IDs
- AI commands execute sequentially — no parallel tool calls
- getBoardState() must return a SCOPED view, not the entire
board:
  - For creation commands: return only frame/container context
near target coordinates
  - For manipulation commands: return only the objects matching
the query (e.g., "all pink sticky notes")
  - For layout commands: return only the objects within the
relevant area or selection
  - Hard cap: never send more than 50 objects in a single
getBoardState() response
  - Include a summary line: "{totalObjects} total objects on
board, {returnedCount} included in context"
- All AI-generated board changes must write to the Yjs shared
document (same sync path as manual edits)
- AI responses must complete in <2 seconds for single-step
commands
- System prompt must constrain the agent to board manipulation
only — reject off-topic requests
```

0.3 — .cursorignore

```
.env
.env.local
.env.production
node_modules/
.next/
coverage/
dist/
*.log
```

0.4 — Skills Installation

Install relevant skills from `skills.sh`:

```
# Web design guidelines (for UI polish)
npx skills-cli install web-design --project --symlink

# React/Next.js patterns
npx skills-cli install react-nextjs --project --symlink

# WebSocket best practices (if available)
npx skills-cli install websocket --project --symlink
```

Additionally, consider building a **custom skill** for "Yjs + Konva Canvas Sync" that encapsulates the pattern of observing Yjs shared map changes and rendering them as Konva nodes. This skill should include: how to initialize a Y.Doc, how to create a Y.Map for board objects, how to observe deep changes, and how to snapshot/restore state for persistence. This can be progressively loaded when the agent is working on sync-related code.

0.5 — Claude Code Configuration

If supplementing Cursor with Claude Code (CLI):

- Run `/init` in the project root to generate `CLAUDE.md`
- Add to `CLAUDE.md`:
    - Build command: `npm run build`
    - Test command: `npx vitest run`
    - Lint command: `npx eslint . --fix`
    - Dev server: `npm run dev`
    - Style: "TypeScript strict, functional components, Zustand for state, Konva for canvas"
- Use Claude Code as a **reviewer**: generate code in Cursor, then run `claude review` in terminal for a second opinion on sync logic and security

0.6 — Source Control Methodology

Branching Strategy: Trunk-Based Development

For a solo sprint with a 24-hour MVP gate, trunk-based development is the correct choice over Gitflow. There is no team to coordinate feature branches with, and the cost of merge conflicts against yourself outweighs any isolation benefit.

Branch structure:

```
main                ← production (auto-deploys to Vercel)
├── feat/cursors    ← short-lived feature branches (merge same day)
├── feat/objects
├── feat/ai-agent
└── fix/sync-bug    ← hotfix branches
```

Commit discipline: - Commit after every working feature increment — not at end of day - Commit messages follow Conventional Commits: `feat:`, `fix:`, `refactor:`, `test:`, `docs:` - Every commit on `main` should be deployable. Never push broken code to `main`. - Use `git stash` liberally when context-switching between canvas work and sync work

Cursor + Git workflow: - Before starting a new Cursor Agent chat session, always `git commit` current working state - If the agent produces bad output, `git diff` to review, then `git checkout .` to revert - Use `git log --oneline -10` as a quick sanity check before any major agent prompt - Tag milestones: `git tag mvp-complete`, `git tag friday-submission`, `git tag final`

.gitignore:

```
node_modules/
.next/
.env
.env.local
.env.production
coverage/
dist/
*.log
.vercel/
.supabase/
```

0.7 — System Design Methodology

Before prompting the coding agent to scaffold anything, establish the system design on paper (or in a markdown doc). The agent writes better code when it has a complete architectural picture in context.

Step 1: Data Flow Diagram

Map the three real-time data flows before writing any code:

```
[User A Browser]                        [User B Browser]
     │                                        │
     │         ┌───────────────────────┐      │
     ├── cursor:move ──► Socket.io ◄── cursor:move ──┤
     │         │   Server (in-memory)  │      │
     │         │   broadcast, no DB    │      │
     │         └───────────────────────┘      │
     │                                        │
     │         ┌───────────────────────┐      │
     ├── Yjs updates ──► y-websocket ◄── Yjs updates ─┤
     │         │   Server (in-memory)  │      │
     │         │   CRDT merge + relay  │      │
     │         │         │             │      │
     │         │   debounced snapshot  │       │
     │         │   every 500ms or on   │      │
     │         │   disconnect ──► Supabase    │
     │         │             (Postgres)│       │
     │         └───────────────────────┘      │
     │                                        │
     └── ai:command ──► Next.js API ──► OpenAI ──┐
                           │                     │
                           ◄── tool result ──────┘
                           │
                           └── writes to Yjs doc ──► y-websocket
──► All clients
```

Step 2: State Ownership Map

Define where each piece of state lives and who owns it. This prevents the coding agent from creating duplicate state or conflicting sync paths.

State	Owner	Sync Method	Persistence
Cursor positions	Client (local)	Socket.io broadcast (20-30Hz throttled)	None (ephemeral)
Board objects	Yjs shared document (Y.Map)	y-websocket CRDT sync	Debounced snapshots → Supabase Postgres
Selected tool	Client (Zustand)	None (local only)	None
Zoom / pan level	Client (Zustand)	None (local only)	None
Presence (who's online)	Yjs awareness protocol	y-websocket awareness	None (ephemeral)
AI command history	Server (Postgres)	None (query on demand)	Full (DB)
Auth session	Client (cookie)	Supabase Auth	Session cookie

Step 3: Event Schema Contract

Define the typed WebSocket event shapes before any implementation. This becomes the contract between frontend and backend that the coding agent must follow.

```
        // === CURSOR EVENTS (Socket.io broadcast — ephemeral, high frequency) ===
        // Throttled to 20-30Hz on sender side
        type CursorMoveEvent = {
          type: 'cursor:move';
          userId: string;
          userName: string;
          x: number;
          y: number;
          color: string; // user-assigned cursor color
        };

        // === BOARD OBJECT SYNC (Yjs — CRDT, automatic) ===
        // No manual event types needed. Yjs handles sync via:
        // - Y.Map for the board objects collection (key = objectId, value = object data)
        // - yDoc.on('update', ...) for observing remote changes
        // - Awareness protocol for presence (who's online, cursor positions as backup)
        //
        // Object schema stored in Yjs Y.Map:
        type BoardObject = {
          id: string;
          type: 'sticky_note' | 'rectangle' | 'circle' | 'line' | 'connector' | 'frame' | 'text';
          x: number;
          y: number;
          width: number;
          height: number;
          rotation: number;
          zIndex: number;
          properties: Record<string, unknown>; // color, text, fontSize, connectedTo, etc.
          createdBy: string;
          updatedAt: string; // ISO timestamp
        };

        // === PERSISTENCE (Debounced snapshot — Yjs → Supabase) ===
        // Server-side: every 500ms or on last client disconnect,
        // serialize Y.Doc via Y.encodeStateAsUpdate() and store as bytea in Postgres
        // On board load: fetch snapshot from Postgres, apply via Y.applyUpdate()

        // === AI EVENTS (REST — request/response, lowest frequency) ===
        type AICommandRequest = {
          boardId: string;
          command: string; // natural language
        };

        type AICommandResponse = {
          success: boolean;
          actions: AIToolCall[];
          objectsAffected: string[]; // IDs of created/modified objects
          error?: string;
        };

        // AI getBoardState scoped response
        type ScopedBoardState = {
          totalObjects: number;
          returnedCount: number;
          objects: BoardObject[]; // max 50, filtered by relevance to command
        };
```

Step 4: Feed the Design to the Agent

Once these three artifacts (data flow, state ownership, event schema) exist as markdown files in the project root, reference them in every Cursor prompt:

- “Based on `@system-design.md`, scaffold the Yjs document provider and Socket.io cursor broadcast”
- “Following the event schema in `@system-design.md`, implement the Y.Map observer that syncs Konva nodes”

This eliminates the most common failure mode: the agent inventing its own state management strategy that conflicts with your sync architecture.

0.8 — agents.md (Project-Level Context)

Create `agents.md` in root with high-level non-negotiables:

```
        # CollabBoard — Agent Context

        ## What We're Building
        A real-time collaborative whiteboard (like Miro) with an AI agent
that manipulates the board via natural language.

        ## Architecture Summary
        - Board object state lives in a Yjs shared document (Y.Map),
synced via y-websocket
        - Cursor positions are broadcast via Socket.io (ephemeral, not
persisted)
        - Persistence: Yjs doc snapshots are debounced to Supabase
Postgres every 500ms
        - Auth: Supabase Auth (JWT sessions)
        - AI agent writes to the Yjs doc — same sync path as manual edits

        ## Architecture Priorities (in order)
        1. Multiplayer sync must be bulletproof — Yjs CRDT handles
conflict resolution automatically
        2. State persistence — Yjs snapshots survive all users leaving
        3.
Performance — 60fps during pan/zoom, <100ms object sync, <50ms
cursor sync
        4. AI agent — single-step tool calls first, multi-step second

        ## Critical Constraints
        - Must support 5+ concurrent users without degradation
        - Must handle 500+ objects without performance drops
        - Must handle network disconnection and reconnection gracefully
(Yjs handles this natively)
        - AI agent changes must write to the Yjs shared document, not
bypass it
        - Cursor events must be throttled to 20-30Hz on the sender side
        - getBoardState() for AI must be scoped — never send more than 50 objects
        - Deployment must be publicly accessible with authentication

        ## DO NOT
        - Use `any` types
        - Skip error handling on WebSocket events
        - Implement AI agent before multiplayer sync works
        - Store board objects in Zustand — Yjs is the single source of truth
        - Write board objects directly to Postgres — always go through Yjs
        - Send unthrottled cursor events
        - Pass the entire board (500+ objects) to the AI agent in one call
```

Phase 1: Define Your Constraints

1. Scale & Load Profile

Users at launch: 2–5 concurrent users for MVP demonstration and grading. The evaluation explicitly tests "5+ concurrent users without degradation."

Users at 6 months (projected): 100–1,000 if this becomes a showcase project or gets adopted by a small team.

Traffic pattern: Spiky. Collaborative whiteboards see burst usage during meetings and workshops, then go idle. This means the backend must handle sudden connection spikes but doesn't need sustained high throughput.

Real-time requirements: WebSockets are mandatory. The spec requires <50ms cursor sync and <100ms object sync latency. Polling is not viable. Three approaches were evaluated:

- **Supabase Realtime** (managed): Broadcast channels for cursors, Postgres Changes for object sync. Simplest to set up, but object sync goes through a DB round-trip on every operation (client → Postgres write → CDC trigger → Realtime broadcast → other clients). Under rapid drag/resize with 5 users and 500 objects, this latency path is unlikely to meet <100ms consistently.
- **Socket.io direct relay** (self-managed): In-memory state on the server, broadcast diffs to all clients. Fastest possible latency, but requires building custom conflict resolution and persistence.
- **Yjs (CRDT) + y-websocket** (self-managed): Full CRDT-based conflict-free merging via Yjs shared documents, synced over WebSocket. Conflict resolution, undo/redo, and offline reconnect come free. The setup cost is ~2 extra hours on day one, but eliminates an entire class

of sync bugs that would otherwise eat debugging time from Wednesday through Sunday.

Decision: Yjs + y-websocket for board object sync, Socket.io broadcast for cursor positions. This hybrid gives us CRDT conflict resolution without the Supabase Realtime latency bottleneck. Cursors stay on a separate fast path (Socket.io broadcast, ephemeral, no persistence needed). Supabase is still used for auth and persistence (Yjs doc snapshots stored as bytea in Postgres), but it is NOT the real-time transport.

Why not pure Supabase Realtime: The DB round-trip on every object mutation is the dealbreaker. During a drag operation, a user generates 30-60 position updates per second. Writing each one to Postgres and waiting for CDC to fire is architecturally wrong for this use case. Yjs holds state in memory and syncs diffs over the wire directly — the DB only gets a debounced snapshot every 500ms.

Cold start tolerance: The WebSocket server (Railway or Render) needs to be always-on. Unlike Supabase Realtime which is managed, we need a persistent process. Railway's free tier includes 500 hours/month which covers the sprint. The Vercel frontend still has cold starts on serverless functions, but the canvas client connects directly to the WebSocket server, not to Vercel.

2. Budget & Cost Ceiling

Monthly spend limit: $450/month Ramp card budget must cover IDE, LLM subscriptions, and hosting.

Allocation: | Item | Monthly Cost | Priority | |—|—|—| | Cursor Pro/Max | $20–$200 | Critical — primary IDE | | Claude Max or Pro | $20–$100 | Critical — chat + code review | | Supabase (Free tier) | $0 | Free tier covers 500MB DB, Auth | | Railway (WebSocket server) | $0–$5 | Free tier: 500 hours/month (sufficient for sprint) | | Vercel (Free tier) | $0 | Free tier covers hobby deployments | | OpenAI API (AI agent) | $5–$20 | Pay-per-use for function calling | | Domain (optional) | $0–$12 | Nice-to-have for showcase |

Trade money for time: Yes — using Supabase's managed Realtime and Auth instead of building custom WebSocket infrastructure saves 6-8 hours on MVP day.

3. Time to Ship

MVP timeline: 24 hours (hard gate — Tuesday midnight CT).

Priority: Speed-to-market dominates. The MVP gate is pass/fail with 9 required items. Every architectural decision must optimize for "can I have multiplayer cursors and sticky notes working by tomorrow night?"

Iteration cadence: Daily. Tuesday→Friday is feature expansion. Friday→Sunday is polish, AI agent, documentation, and deployment hardening.

Maintainability consideration: Low priority for Week 1. This is a sprint project, not a long-term codebase. However, clean separation of concerns (sync layer vs. canvas layer vs. AI layer) will make the Friday→Sunday iteration smoother.

4. Compliance & Regulatory Needs

HIPAA: Not applicable. No health data.

GDPR: Not applicable for MVP. No EU users targeted. If scaling, would need cookie consent and data deletion endpoints.

SOC 2: Not applicable. No enterprise clients in scope.

Data residency: Not applicable. Supabase default region (US) is fine.

Decision: No compliance overhead for this sprint. Focus entirely on functionality and performance.

5. Team & Skill Constraints

Solo or team: Solo developer with AI coding agents.

Languages/frameworks known well: TypeScript, React, Next.js. Moderate familiarity with canvas APIs. Limited experience with CRDTs and real-time sync protocols.

Learning appetite vs. shipping speed: Yjs has a learning curve (~2 hours to understand Y.Doc, Y.Map, and awareness protocol), but it pays back immediately by eliminating manual conflict resolution, undo/redo implementation, and disconnect recovery logic. The net time cost is lower than building those features manually with Supabase Realtime.

Honest gaps: - Never built a collaborative real-time app before - Limited experience with HTML5 Canvas / Konva.js - No prior CRDT implementation (Yjs is new) - Limited experience with AI function-calling / tool-use patterns - No prior experience running a persistent WebSocket server (Railway/ Render)

Mitigation: Lean heavily on Cursor with indexed Yjs and Konva.js docs. Yjs has excellent documentation and a small API surface (Y.Doc, Y.Map, Y.Array, awareness). Use Claude Code as a reviewer for sync logic. Pre-research Konva.js object model and Yjs shared types before writing any canvas code. Railway deployment is a single `Dockerfile` or `package.json` start script.

Phase 2: Architecture Discovery

6. Hosting & Deployment

Decision: Vercel (frontend) + Railway (WebSocket server) + Supabase (DB/Auth)

Option	Pros	Cons
Vercel + Railway + Supabase	Free tiers, Vercel optimized for Next.js, Railway runs persistent WS server, Supabase for managed auth + Postgres	Three services to manage, Railway free tier has hour limits
Vercel + Supabase (Realtime)	Two services, simplest setup	Supabase Realtime has DB round-trip latency, not suitable for rapid object sync
Railway only (full stack)	Single deploy, persistent server	Slower deploys, no Next.js edge optimization
Firebase	Excellent Realtime DB	Vendor lock-in, Firestore query limitations, no CRDT support

Why this three-tier split: - **Vercel** handles the Next.js frontend and API routes (auth callbacks, AI agent endpoint). Native Next.js optimization, instant deploys from GitHub. - **Railway** runs the y-websocket + Socket.io server as a persistent Node.js process. This must be always-on (not serverless) because WebSocket connections are long-lived. Railway's free tier provides 500 hours/month — more than enough for the sprint. - **Supabase** provides PostgreSQL for persistence (Yjs doc snapshots, board metadata) and Auth. It is NOT used as the real-time transport.

CI/CD: Vercel auto-deploys frontend from GitHub `main`. Railway auto-deploys the WebSocket server from a `/server` directory or separate branch. Supabase migrations managed via `supabase` CLI.

Scaling: Vercel scales serverless functions automatically. Railway scales the WS server vertically (add more RAM/CPU). For 5 users this is trivial. For 100+ concurrent boards, would need to shard y-websocket rooms across multiple Railway instances or move to a managed WebSocket platform.

7. Authentication & Authorization

Decision: Supabase Auth with email/password + magic link

Rationale: The MVP requires "User authentication" as a hard gate. Supabase Auth provides: - Email/password and magic link out of the box - JWT-based sessions that work with Supabase's Row Level Security (RLS) - Built-in `@supabase/auth-helpers-nextjs` for Next.js integration - User metadata (display name) for multiplayer cursor labels

RBAC: Not needed for MVP. All authenticated users have equal board access. For final submission, could add board-level permissions (owner, editor, viewer) via Supabase RLS policies.

Multi-tenancy: Each board is its own “room.” Users join a board by URL/ID. No workspace-level multi-tenancy needed.

Social login: Omitted for MVP. Can add Google OAuth via Supabase if time permits for final submission.

8. Database & Data Layer

Decision: Yjs (in-memory CRDT) as primary data layer + Supabase PostgreSQL for persistence

The critical insight is that the database is NOT the real-time sync layer. Yjs holds the live board state in memory on the WebSocket server and in each client. Supabase Postgres is the durability layer — it stores snapshots so boards survive server restarts and all-users-leave scenarios.

Schema Design (Draft):

```
boards
  - id: uuid (PK)
  - name: text
  - created_by: uuid (FK → auth.users)
  - created_at: timestamp
  - updated_at: timestamp

board_snapshots
  - id: uuid (PK)
  - board_id: uuid (FK → boards)
  - yjs_state: bytea (Y.encodeStateAsUpdate() binary)
  - snapshot_at: timestamp
  - UNIQUE(board_id) — only latest snapshot kept, upserted on
each save

ai_command_log (optional, for cost tracking)
  - id: uuid (PK)
  - board_id: uuid (FK → boards)
  - user_id: uuid (FK → auth.users)
  - command_text: text
  - tokens_used: integer
  - model: text
  - created_at: timestamp
```

Persistence strategy: 1. When a client connects to a board, the y-websocket server loads the latest `board_snapshots.yjs_state` from Supabase and applies it to the Y.Doc via `Y.applyUpdate()`. 2. While clients are connected, the Y.Doc lives in memory on the server. All sync happens via Yjs CRDT protocol — no DB reads or writes during active editing. 3. The server debounces snapshots to Supabase every 500ms (or immediately on last client disconnect). Snapshot = `Y.encodeStateAsUpdate(yDoc)` stored as `bytea`. 4. On server restart, the Y.Doc is reconstructed from the latest snapshot. Yjs updates are incremental, so no data is lost as long as the most recent snapshot was saved.

Why NOT store individual objects as rows: The previous architecture stored each board object as a `board_objects` row and used Supabase Postgres Changes for sync. This creates a problem: every drag/resize generates a DB write, which triggers CDC, which broadcasts to clients. At 30fps drag speed × 5 users, that's 150 writes/second hitting Postgres. Yjs eliminates this entirely by keeping the hot path in memory and only persisting a compact binary snapshot periodically.

Conflict resolution: Yjs CRDTs handle this automatically. When two users edit the same object simultaneously, Yjs merges the changes deterministically based on its internal clock. No last-write-wins data loss. No manual conflict resolution code needed. This is a significant upgrade over the previous approach and directly addresses the spec's "Handle simultaneous edits" requirement.

Read/write ratio to Postgres: Extremely read-light, write-light. One read on board load (fetch snapshot), one write every 500ms during active editing. Supabase free tier is more than sufficient.

Vector storage: Not needed. The AI agent uses function calling with scoped board state, not semantic search.

9. Backend/API Architecture

Decision: Next.js API routes (REST) + standalone y-websocket/ Socket.io server

Rationale: The backend is split into two processes:

1. **Next.js API routes (Vercel):** Handle stateless request/response operations — board CRUD, AI agent command endpoint, auth callbacks. These are serverless functions that scale automatically.
2. **y-websocket + Socket.io server (Railway):** A persistent Node.js process that manages WebSocket connections, Yjs document sync, cursor broadcast, and debounced persistence to Supabase. This MUST be a long-running process because WebSocket connections and Yjs documents live in memory.

Why two processes instead of one: Vercel's serverless functions cannot hold WebSocket connections (they terminate after execution). The WebSocket server must be always-on. Keeping the REST API on Vercel gives us free serverless scaling for the AI agent endpoint, which can be CPU-intensive but is stateless.

REST vs. GraphQL vs. tRPC: - REST is the pragmatic choice for the API routes. Simple endpoints, easy to debug, no additional tooling. - The WebSocket server uses raw Socket.io events for cursors and the y-websocket protocol for object sync — no REST/GraphQL layer needed.

Background jobs: Not needed for MVP. AI commands execute synchronously via the Next.js API route. For scale, complex multi-step AI commands could be offloaded to a background queue on Railway, but that's a post-MVP concern.

10. Frontend Framework & Rendering

Decision: Next.js 15 (App Router) + Konva.js + react-konva

Why Next.js 15: Already the most familiar React meta-framework. App Router provides server components for the auth flow and layout, client components for the interactive canvas.

Why Konva.js over alternatives:

Library	Pros	Cons
Konva.js + react-konva	Object-oriented canvas, hit detection, event system, React bindings, extensive docs	Slightly larger bundle than raw Canvas
Fabric.js	Rich feature set, SVG support	React integration is clunky, heavier
PixiJS	WebGL performance, GPU acceleration	Gaming-oriented, overkill for whiteboard
Raw HTML5 Canvas	Zero overhead	Must build own object model, hit detection, event handling

Decision: Konva.js. The object model (Stage → Layer → Shape) maps cleanly to whiteboard concepts. react-konva provides declarative rendering that works with React state. Hit detection and drag events come free. The tradeoff vs. raw Canvas is a larger bundle, but the development speed gain is worth it for a 24-hour MVP.

Pan/Zoom: Konva's Stage supports `draggable` for panning and `scaleX/scaleY` for zoom. Implement with mouse wheel for zoom and drag on empty canvas for pan. The "infinite board" requirement is met by not capping the Stage's coordinate space.

SEO: Not needed. This is a web application, not a content site.

PWA/Offline: Not needed for MVP. Could add service worker caching for final submission.

11. Third-Party Integrations

Service	Purpose	Pricing Risk
Supabase	DB (persistence), Auth	Free tier: 500MB DB, 50K auth users
Railway	WebSocket server (y-websocket + Socket.io)	Free tier: 500 hours/month, 512MB RAM
Vercel		

Service	Purpose	Pricing Risk
	Frontend hosting, API routes	Free tier: 100GB bandwidth, serverless functions
OpenAI API	AI agent (function calling)	Pay-per-use: ~$0.01–0.03 per command with GPT-4o-mini
Anthropic API (alt)	AI agent (tool use)	Pay-per-use: ~$0.01–0.05 per command with Sonnet

Rate limits: OpenAI's GPT-4o-mini has generous rate limits (500 RPM on Tier 1). Railway's free tier WebSocket server can handle hundreds of concurrent connections within its 512MB memory limit — more than enough for 5+ users per board.

Vendor lock-in risk: Low. Yjs is an open-source library, not a vendor service. The WebSocket server is a standard Node.js process that can run anywhere (Railway, Render, Fly.io, a VPS). Supabase Auth uses standard JWTs. The only Supabase-specific piece is the Postgres persistence layer, and even that is just a `bytea` column — trivially portable to any Postgres host.

AI Provider Decision: Start with OpenAI GPT-4o-mini for cost efficiency. Function calling is well-documented and reliable. If response quality is insufficient for complex commands (SWOT analysis, journey maps), upgrade to GPT-4o for those specific command types while keeping GPT-4o-mini for simple creation commands.

Phase 3: Post-Stack Refinement

12. Security Vulnerabilities

Known pitfalls for this stack:

- **WebSocket server authentication:** The y-websocket server must validate Supabase JWTs on connection. Without this, anyone with the WebSocket URL can join any board. Implement a `verifyClient` hook that checks the JWT before upgrading the HTTP connection.
- **Yjs document access control:** y-websocket uses room names to separate boards. The server must verify that the connecting user has access to the requested board room before allowing the connection.
- **AI prompt injection:** Users could craft natural language commands that attempt to manipulate the AI agent beyond board operations. The tool schema must be strictly defined and the system prompt must constrain behavior to board manipulation only. Never pass raw user input to a system prompt — always use it as a tool argument.
- **XSS via sticky note text:** User-generated text content (sticky notes, labels) must be sanitized before rendering. React's default escaping handles most cases, but `dangerouslySetInnerHTML` must never be used for user content.

- **Environment variable exposure:** `.env` files with Supabase keys, OpenAI API keys, and Railway deploy tokens must be in `.gitignore` and `.cursorignore`. Use `NEXT_PUBLIC_` prefix only for the Supabase anon key (which is safe to expose by design). The WebSocket server URL should be in `NEXT_PUBLIC_WS_URL`.
- **Yjs snapshot injection:** If a malicious client sends crafted Yjs updates, they could corrupt the shared document. y-websocket's default behavior relays all updates. For production, consider validating update sizes and rates on the server.
- **CORS:** The WebSocket server on Railway will be on a different origin than the Vercel frontend. Configure Socket.io and y-websocket to accept connections only from the Vercel deployment URL.

13. File Structure & Project Organization

```
collabboard/
├── .cursor/
│   └── rules/
│       ├── tech-stack.mdc
│       ├── tdd.mdc
│       ├── code-patterns.mdc
│       └── ai-agent.mdc
├── agents.md
├── system-design.md          # Data flow, state ownership, event
schema
├── CLAUDE.md
├── .cursorignore
├── .env.local
├── server/                    # WebSocket server (deploys to
Railway)
│   ├── index.ts               # y-websocket + Socket.io server
entry
│   ├── auth.ts                # JWT verification for WS
connections
│   ├── persistence.ts         # Debounced Yjs snapshot →
Supabase
│   └── package.json
├── app/
│   ├── layout.tsx             # Root layout with auth provider
│   ├── page.tsx               # Landing / board list
│   ├── board/
│   │   └── [id]/
│   │       └── page.tsx       # Board view (client component)
│   └── api/
│       ├── boards/
│       │   └── route.ts       # Board CRUD
│       └── ai/
│           └── command/
│               └── route.ts   # AI agent endpoint
├── components/
│   ├── board/
│   │   ├── Canvas.tsx         # Main Konva Stage
│   │   ├── StickyNote.tsx     # Sticky note component
│   │   ├── Shape.tsx          # Rectangle, circle, line
│   │   ├── Connector.tsx      # Arrow/line connectors
│   │   ├── Frame.tsx          # Grouping frames
│   │   ├── Toolbar.tsx        # Tool selection bar
│   │   ├── Cursors.tsx        # Remote cursor overlays
│   │   └── AICommandBar.tsx   # Natural language input
│   ├── auth/
│   │   ├── LoginForm.tsx
│   │   └── AuthProvider.tsx
│   └── ui/                    # Shared UI components
├── lib/
│   ├── supabase/
│   │   ├── client.ts          # Browser client (auth +
persistence queries)
│   │   └── server.ts          # Server client
│   ├── yjs/
│   │   ├── provider.ts        # y-websocket client provider
setup
│   │   ├── board-doc.ts       # Y.Doc schema: Y.Map<BoardObject>
│   │   └── awareness.ts       # Presence via Yjs awareness
protocol
│   ├── sync/
│   │   ├── cursor-socket.ts   # Socket.io client for cursor
broadcast
│   │   └── throttle.ts        # 20-30Hz cursor throttle
│   ├── ai-agent/
│   │   ├── tools.ts           # Tool definitions (JSON Schema)
│   │   ├── executor.ts        # Tool execution → writes to Yjs
doc
│   │   ├── scoped-state.ts    # getBoardState() with 50-object
cap
│   │   └── prompts.ts         # System prompt + few-shot
examples
│   └── utils/
│       └── board-helpers.ts
├── stores/
│   └── ui-store.ts            # Zustand: selected tool, zoom,
pan (local only)
├── types/
│   ├── board.ts               # BoardObject, Shape, etc.
│   ├── sync.ts                # CursorMoveEvent,
ScopedBoardState
│   └── ai.ts                  # AI command/tool types
├── __tests__/
│   ├── unit/
│   │   ├── yjs-board-doc.test.ts    # Yjs map operations
│   │   ├── ai-tools.test.ts
│   │   ├── scoped-state.test.ts     # getBoardState() filtering
│   │   └── board-helpers.test.ts
│   └── integration/
│       ├── yjs-sync.test.ts        # 2 Y.Docs syncing via mock
provider
│       └── cursor-broadcast.test.ts # Socket.io cursor relay
└── public/
    └── ...
```

Decision: Monorepo (single directory). No need for polyrepo with a solo sprint. Feature-based organization under `components/board/` and `lib/` keeps concerns separated without over-engineering.

14. Naming Conventions & Code Style

Convention	Pattern	Example
Files	kebab-case	`board-store.ts`, `ai-agent.ts`
React components	PascalCase	`StickyNote.tsx`, `AICommandBar.tsx`
Functions	camelCase	`createStickyNote()`, `handleDragEnd()`
Types/Interfaces	PascalCase	`BoardObject`, `AICommand`
Zustand stores	camelCase with `use` prefix	`useBoardStore`, `usePresenceStore`
API routes	kebab-case directories	`/api/ai/command/route.ts`
Constants	SCREAMING_SNAKE_CASE	`MAX_OBJECTS`, `CURSOR_THROTTLE_MS`
DB columns	snake_case	`board_id`, `created_by`, `z_index`
WebSocket events	colon-separated namespace	`cursor:move`, `object:create`, `object:update`

Linter: ESLint with `@typescript-eslint/recommended` + Next.js recommended config.

Formatter: Prettier with 2-space indent, single quotes, trailing commas.

15. Testing Strategy

MVP coverage target: Focused on the critical path — sync correctness and AI tool execution. E2E browser tests are deferred to Friday to avoid burning MVP hours on Playwright setup.

Layer	Tool	Coverage Target	Focus Areas
Unit tests	Vitest	Yjs doc operations, AI tool execution, scoped state	Y.Map CRUD, tool schema validation, getBoardState filtering

Layer	Tool	Coverage Target	Focus Areas
Integration tests	Vitest	2 Y.Docs syncing, cursor relay	Create object on Doc A → verify appears on Doc B, cursor broadcast round-trip
E2E tests (Friday+)	Playwright	2-browser multiplayer scenario	Full flow: login → create board → add sticky → see on other browser

Mocking patterns: - Create two Y.Doc instances connected by a mock y-websocket provider (Yjs provides test utilities for this) to simulate multi-user sync without a running server - Mock Socket.io with a local event emitter for cursor broadcast tests - Mock OpenAI API responses for AI agent tests (avoid burning tokens during test runs) - Test Yjs persistence by encoding a Y.Doc to binary, creating a new Y.Doc, applying the update, and verifying state matches

What NOT to test in MVP: Individual UI component rendering, Tailwind class application, Supabase Auth internals, Playwright browser automation. These are either covered by library test suites or deferred to post-MVP.

16. Recommended Tooling & DX

VS Code / Cursor Extensions: - `Tailwind CSS IntelliSense` — autocomplete for Tailwind classes - `ESLint` — inline linting - `Prettier` — format on save - `Supabase` — schema explorer (if available) - `Thunder Client` — quick API testing without leaving the IDE

CLI Tools: - `supabase` CLI — local dev, migrations, schema management - `npx vitest` — test runner - `railway` CLI — deploy and manage WebSocket server - `vercel` CLI — manual deploys if needed

Debugging Setup: - Chrome DevTools → Network tab → WS filter for WebSocket frame inspection (both y-websocket and Socket.io frames visible) - Yjs `yDoc.on('update', ...)` logger — add a dev-only hook that logs every Yjs update with a decoded summary - Socket.io debug mode: set `DEBUG=socket.io*` env var on the server for verbose connection/event logging - `console.table()` for board state snapshots during development - Custom `useDebugSync()` hook that logs Yjs Y.Map changes and cursor events in dev mode - Railway logs: `railway logs` in terminal for real-time WebSocket server output

---

Build Priority (from spec)

The assignment defines a strict vertical build order. Every architectural decision above is oriented around making this sequence as fast as possible:

1. **Yjs + y-websocket server setup** — Get the WebSocket server running on Railway, two clients connecting to the same Y.Doc
2. **Cursor sync** — Socket.io broadcast layer, two cursors moving across browsers (throttled to 20-30Hz)
3. **Object sync** — Sticky notes created via Yjs Y.Map appear for all users instantly
4. **Conflict handling** — Already handled by Yjs CRDT (verify with concurrent edit test)
5. **State persistence** — Debounced Yjs snapshots to Supabase Postgres, board survives full disconnect
6. **Board features** — Shapes, frames, connectors, transforms (Konva.js components driven by Y.Map)
7. **AI commands (basic)** — Single-step creation/manipulation via OpenAI function calling, writes to Yjs doc
8. **AI commands (complex)** — Multi-step template generation (SWOT, journey maps) with scoped getBoardState()

AI Cost Analysis (Preliminary)

Development Costs (Estimated for 1-Week Sprint)

Cost Category	Estimated Spend
Cursor IDE (monthly)	\$20–\$200
Claude/OpenAI chat (monthly)	\$20–\$100
OpenAI API for AI agent dev/testing	\$5–\$15
Railway (WebSocket server)	\$0 (free tier, 500 hrs/month)
Supabase	\$0 (free tier)
Vercel	\$0 (free tier)
Total Sprint Cost	**\$45–\$315**

Production Cost Projections

Assumptions: - Average 5 AI commands per user per session - Average 3 sessions per user per month - ~800 tokens per command (input + output) using GPT-4o-mini at \$0.15/\$0.60 per 1M tokens - Supabase Pro plan at \$25/month for >500MB or >200 connections

Scale	AI API Cost	Railway (WS Server)	Supabase	Vercel	Total/Month
100 users	~\$0.18	\$0 (free tier)	\$0 (free tier)	\$0	~\$1

Scale	AI API Cost	Railway (WS Server)	Supabase	Vercel	Total/ Month
1,000 users	~$1.80	$5 (Hobby)	$25 (Pro)	$0	~$32
10,000 users	~$18	$20 (Pro)	$25 (Pro)	$20 (Pro)	~$83
100,000 users	~$180	$100 (multi-instance)	$599 (Team)	$150 (Enterprise)	~$1,029

Cost optimization strategies: - Use GPT-4o-mini instead of GPT-4o (10x cheaper, sufficient for board commands) - Scope `getBoardState()` to max 50 objects to reduce token usage per AI call - Throttle cursor sync to 20-30Hz (well within <50ms target, halves Socket.io message volume) - Yjs snapshots are compact binary (much smaller than JSON), minimizing Supabase storage costs - At 10K+ users, shard y-websocket rooms across multiple Railway instances (each board is an independent Y.Doc)

Decision Summary

Decision	Choice	Key Tradeoff
Frontend	Next.js 15 + Konva.js	Familiar framework, object-oriented canvas; tradeoff is bundle size vs. raw Canvas
Real-time sync (objects)	Yjs CRDT + y-websocket	Automatic conflict resolution, offline reconnect; tradeoff is learning curve (~2hrs) and need for persistent WS server
Real-time sync (cursors)	Socket.io broadcast (20-30Hz throttled)	Fast ephemeral path; tradeoff is separate transport from Yjs
Persistence	Debounced Yjs snapshots → Supabase Postgres	Minimal DB load (1 write/500ms); tradeoff is snapshot-level granularity, not per-object
Auth	Supabase Auth	Managed JWT sessions; tradeoff is dependency on Supabase for auth even if DB usage is light
AI agent	OpenAI GPT-4o-mini with function calling + scoped getBoardState()	Cheap + reliable + bounded context; tradeoff is less capable than GPT-4o for complex templates
State management		

Decision	Choice	Key Tradeoff
	Zustand (UI only) + Yjs (board objects)	Clear ownership separation; tradeoff is two state systems to reason about
Deployment	Vercel (frontend) + Railway (WS server) + Supabase (DB)	Each service optimized for its role; tradeoff is three services to deploy vs. one
Testing	Vitest (MVP) + Playwright (Friday+)	Fast feedback loop for sync tests; tradeoff is no browser-level E2E until mid-week