CollabBoard

Pre-Search Architecture Document

Building Real-Time Collaborative Whiteboard Tools with AI-First Development

Project	CollabBoard — Collaborative Whiteboard with AI Agent
Developer	Solo developer
Timeline	7-day sprint (MVP in 24 hours)
Date	February 2026

Recommended Technology Stack

Layer	Choice	Key Reason
Real-time Sync	Yjs + Hocuspocus	CRDTs are the correct primitive for collaborative editing
Canvas Rendering	Fabric.js	Built-in selection, transforms, grouping, serialization
Frontend	React + TypeScript + Vite	Ecosystem, tooling, type safety across subsystems
Authentication	Firebase Auth	Managed, fast setup, Google OAuth + email/password
Persistence	SQLite on Fly.io volume	Integrated with Hocuspocus, zero additional services
AI Model	Claude Sonnet 4.5 (tool use)	Fast, capable, cost-effective at ~$0.004/command

AI Backend	Express (in-process with Hocuspocus)	Direct Yjs document access, AI as "just another user"
Frontend Hosting	Vercel	Edge CDN, auto-deploy on git push
Backend Hosting	Fly.io	Always-on container, WebSocket support, persistent volumes

Phase 1: Define Your Constraints

1. Scale & Load Profile

Users at launch? In 6 months?

At launch: 5–10 concurrent users. The primary audience is Gauntlet evaluators testing during grading windows. The spec requires testing with 5+ concurrent users without degradation. In 6 months: approximately 100 users if the project becomes a portfolio piece. The architecture should not contain decisions that prevent scaling, but over-engineering for scale that won't materialize is avoided.

Traffic pattern: steady, spiky, or unpredictable?

Spiky. Evaluators will test in concentrated windows, potentially all hitting the app within the same hour. During off-hours, traffic drops to zero. Peak concurrent load is what matters, not average throughput. A system that handles 5 concurrent users but degrades at 6 fails the evaluation.

Real-time requirements?

Non-negotiable. The spec requires cursor sync at <50ms latency and object sync at <100ms. This rules out polling. WebSockets are mandatory. Two real-time data channels are needed: high-frequency ephemeral data (cursor positions, ~20–30 updates/second per user) and lower-frequency persistent data (object CRUD, ~1–5 updates/second during active editing). Any hiccups in the real-time experience are unacceptable—smooth collaboration is the core product requirement.

Cold start tolerance?

Low. The board should be interactive within 2–3 seconds of opening. A 5–10 second cold start on a serverless backend would cause WebSocket connection timeouts and create a poor first impression. The real-time server must be always-on or near-instant startup, as long as cold start mitigation doesn't compromise the real-time experience where users spend 99% of their time.

Decision: Design for 10–20 concurrent users per board. Always-on WebSocket server (no serverless for real-time). Optimize for peak concurrent load over average throughput.

Section Summary: 5–10 concurrent users at launch, ~100 in 6 months. Spiky traffic pattern with evaluators hitting simultaneously. Smooth real-time experience is non-negotiable with <50ms cursor sync and <100ms object sync. Minimize cold start issues with always-on infrastructure for the WebSocket layer.

2. Budget & Cost Ceiling

Monthly spend limit?

Target $0/month infrastructure using free tiers. Real-time server hosting on Fly.io free tier ($0–3/month), Vercel free tier for frontend ($0), Firebase Auth free tier ($0), SQLite persistence included with Fly.io VM. AI API costs are the sole variable: Claude Sonnet 4.5 at ~$0.004 per command, projecting $5–15 total during development and evaluation. Total project lifecycle cost: $5–20.

Pay-per-use acceptable or need fixed costs?

Pay-per-use is ideal. Spiky traffic with long idle periods means paying for always-on capacity during weeks of zero traffic would be wasteful. AI API calls are inherently pay-per-use. The only always-on cost is the free-tier WebSocket VM.

Where will you trade money for time?

Three key trades: (1) Managed auth (Firebase Auth) saves 4–8 hours at $0 cost. (2) Hocuspocus framework over custom WebSocket server saves 4–7 hours net after learning curve. (3) Claude API for AI commands ($5–15) replaces weeks of custom NLP work.

Decision: ~$0/month infrastructure via free tiers. $5–15 total AI API spend. Invest money in AI API calls (highest ROI per dollar). Use free managed services for auth and hosting.

Section Summary: ~$0/month infrastructure (free tiers across Fly.io, Vercel, Firebase Auth), $5–15 total AI API spend. Pay-per-use model for AI calls, always-on free-tier VM for real-time server. Key money-for-time trades: managed auth ($0, saves 4–8 hrs), Hocuspocus framework (free, saves 4–7 hrs net), Claude API ($5–15, replaces weeks of custom NLP).

3. Time to Ship

MVP timeline?

24-hour hard gate. The MVP checklist includes: infinite board with pan/zoom, sticky notes, one shape type, object CRUD, real-time sync between 2+ users, multiplayer cursors, presence awareness, authentication, and deployed publicly. Critical path is real-time sync—build order front-loads sync infrastructure. Estimated ~16 hours of implementation with 8 hours of buffer for debugging Yjs/Hocuspocus integration and deployment issues.

Speed-to-market vs. long-term maintainability priority?

Speed to market, decisively. This is a one-week sprint with an admission gate. Invest in structural decisions that prevent time-consuming bugs (TypeScript types for shared models, clean sync/rendering layer separation). Skip polish and abstraction that doesn't affect functionality. A setTimeout hack that resolves a race condition is acceptable over a proper mutex implementation this week.

Iteration cadence after launch?

Monday: Pre-Search. Tuesday (24hrs): MVP hard gate. Wednesday–Friday: full feature buildout (shapes, frames, connectors, transforms, selection, AI agent). Saturday–Sunday: polish, documentation, demo video, deployment hardening. Daily feature iteration Wed–Fri. No post-submission iteration planned.

Decision: 24-hour MVP, speed-to-market over maintainability. Sync first, features second. ~16 hours implementation + 8 hours buffer. Daily iteration Wed–Fri, polish Sat–Sun.

Section Summary: 24-hour MVP hard gate, speed-to-market is the priority. Build order: sync first, features second. Invest in structural correctness (types, layer separation) that prevents expensive debugging; skip abstraction and polish until final weekend. Full submission Sunday 10:59 PM CT.

4. Compliance & Regulatory Needs

No regulatory requirements apply. This is a demo/evaluation project, not a production SaaS product. No HIPAA (no health data), no GDPR (minimal PII, US-based evaluators), no SOC 2 (no enterprise clients), no data residency constraints. Minimal good practice: avoid logging PII, deploy to US-East regions (Fly.io IAD), and acknowledge data storage location in documentation.

Decision: No compliance implementation. Focus entirely on building a working product. Deploy to US-East, note data storage location in README.

Section Summary: No regulatory requirements (HIPAA, GDPR, SOC 2, data residency). Demo project—focus on functionality. Deploy to US-East, document data storage location.

5. Team & Skill Constraints

Solo or team?

Solo developer across a 7-day sprint. No parallelization—critical path is linear. Every technology choice evaluated against one question: does this reduce total friction from zero to deployed?

Languages/frameworks and shipping speed preference?

React + TypeScript frontend, Node.js + TypeScript backend. Single language across the full stack with shared type definitions for board objects. The npm ecosystem for collaborative canvas tools is overwhelmingly React-oriented. Shipping speed is the overriding priority. Accept targeted learning investments in Yjs/Hocuspocus and Fabric.js because both net significant time savings by eliminating custom sync and interaction code. All remaining tooling choices optimized for minimal setup friction: Firebase Auth, Express, Vercel, Fly.io.

Decision: Solo. React + TypeScript full-stack. Ship fast, accept learning only where it nets time savings. Best frameworks for least implementation friction.

Section Summary: Solo developer. React + TypeScript frontend, Node.js + TypeScript backend—single language, shared types. Shipping speed priority. Accept learning in Yjs and Fabric.js (both net time savings). All other tooling: minimal friction.

Phase 2: Architecture Discovery

6. Hosting & Deployment

Serverless vs. containers vs. edge vs. VPS?

Two-tier deployment. Frontend (React SPA): static assets deployed to Vercel's edge CDN. No server, no cold starts, instant loads globally. Backend (Hocuspocus + Express): always-on container on Fly.io. Serverless is ruled out because WebSocket connections must persist for the duration of a user's session—Lambda/Cloud Functions can't hold open WebSocket connections. Edge functions have the same limitation plus restricted Node.js APIs. Render's free tier ruled out due to cold starts after 15 minutes of inactivity. No VPS needed—Fly.io provides always-on containers without OS management.

CI/CD requirements?

Minimal. Vercel's GitHub integration auto-deploys on push to main (entire frontend CI/CD). Backend deploys manually via 'fly deploy' CLI. No CI pipeline, no staging environment, no branch previews. Solo sprint—ship directly to production.

Scaling characteristics?

Single Fly.io instance (256MB RAM) handles 50+ concurrent WebSocket connections comfortably. Yjs documents for 500-object whiteboards are a few hundred KB in memory. Frontend already on CDN—scales infinitely. Horizontal backend scaling possible via Hocuspocus Redis adapter if needed, but not implemented for the sprint.

Decision: Vercel (edge CDN) for frontend. Fly.io (always-on container) for backend. No CI pipeline—auto-deploy frontend, manual deploy backend. Single instance handles target scale.

Section Summary: Vercel (edge CDN) for React SPA with auto-deploy on git push. Fly.io (always-on container) for Hocuspocus WebSocket server and AI endpoint. Serverless ruled out for WebSocket persistence. No CI/CD pipeline. Single Fly.io instance handles 50+ concurrent connections; horizontal scaling via Redis adapter available but not implemented.

7. Authentication & Authorization

Auth approach?

Google social login as primary (one click, auto-populates display name and profile photo for cursor labels and presence indicators). Email/password as fallback via Firebase Auth. Both methods use the same SDK. Skip email verification—adds friction, protects nothing meaningful in a demo. Magic links and SSO out of scope.

RBAC needed?

No. All authenticated users have full edit permissions on any board they access. Board access controlled by shareable URL ("anyone with the link can edit" model, mirroring how evaluators will share board links). No invite system, no viewer-only roles, no admin controls.

Multi-tenancy considerations?

Natural multi-tenancy via Yjs document-per-board. Each board is a separate Yjs document with isolated state, users, and persistence. Hocuspocus handles this natively—each board ID maps to a separate document room. No cross-board data leakage by design. Optional "my boards" dashboard as stretch goal.

Decision: Google OAuth primary + email/password fallback via Firebase Auth. No RBAC. Shareable URL access model. Natural multi-tenancy via Yjs documents.

Section Summary: Google social login (one-click, auto-populates cursor labels) + email/password fallback, both via Firebase Auth. No RBAC—full edit access via shareable URL. Multi-tenancy is natural: each board is a separate Yjs document. Optional "my boards" index as stretch goal.

8. Database & Data Layer

Database type?

Three-tier storage model. (1) Board object state: Yjs documents persisted to SQLite on Fly.io persistent volume via Hocuspocus. Stored as serialized binary blobs keyed by board ID—not individually queryable. The entire board state is one blob that Yjs manages. This is effectively a key-value store. (2) User identity: Firebase Auth, fully managed. (3) Board metadata (stretch): deferred—MVP uses direct URL access with no dashboard.

Real-time sync, full-text search, vector storage, caching needs?

Real-time sync handled entirely by Yjs at the application layer, not the database layer. The database is only involved for cold-start persistence. No full-text search (AI agent uses in-memory board state via getBoardState()). No vector storage (no RAG, no embeddings). No external caching—Hocuspocus manages in-memory caching of active Yjs documents natively with write-through to SQLite on disconnect.

Read/write ratio?

Write-dominant during active sessions. 5 active users generate ~100–150 Yjs mutations per second (cursor moves + edits). Reads are implicit—peers receive changes via WebSocket push, not database queries. Database reads only on cold board loads. Yjs CRDTs are purpose-built for this high-frequency concurrent write pattern.

Decision: Yjs documents persisted to SQLite (key-value blob storage). Firebase Auth for identity. No search, vector, or caching infrastructure. Yjs handles real-time sync at the application layer.

Section Summary: Three-tier storage: Yjs documents in SQLite (binary blobs keyed by board ID), Firebase Auth for identity, board metadata deferred. Real-time sync via Yjs application layer, not database. Write-dominant workload (100–150 mutations/sec with 5 users). Database reads only on cold board loads.

9. Backend/API Architecture

Monolith or microservices?

Monolith. The backend has two responsibilities: Yjs/Hocuspocus WebSocket server and AI command handler. Critically, the AI handler needs direct in-process access to Yjs documents—when Claude returns tool calls, the handler writes to the Yjs document in memory and all connected clients receive updates instantly. Splitting this into microservices would require the AI service to connect as a WebSocket client to

Hocuspocus, adding fragility and latency for zero benefit.

REST vs. GraphQL vs. tRPC vs. gRPC?

REST via Express. The HTTP surface is two endpoints: POST /api/ai-command (receives natural language, calls Claude, executes tool calls against Yjs) and GET /api/health. Everything else is WebSocket traffic through Hocuspocus. GraphQL, tRPC, and gRPC are unjustifiable overhead for a 2-endpoint surface.

Background job and queue requirements?

None. AI commands execute synchronously within the HTTP request (2–4 seconds for complex commands). Yjs document updates propagate to clients in real-time as each tool call executes—users see the AI "building" results step by step. No Redis, no job queue. Concurrent AI commands from multiple users merge naturally via CRDTs.

Decision: Single Node.js monolith (Hocuspocus + Express). REST with 2 endpoints. No background jobs—synchronous AI execution with real-time Yjs propagation.

Section Summary: Single Node.js monolith: Hocuspocus (WebSocket) + Express (HTTP). AI handler in-process with direct Yjs document access—Claude's tool calls write to Yjs, propagating to all clients instantly. REST API with 2 endpoints. No job queue—AI commands execute synchronously with streaming Yjs updates.

10. Frontend Framework & Rendering

SEO requirements?

None. Collaborative whiteboard is authenticated and canvas-rendered—no public content to index. Eliminates SSR frameworks (Next.js, Remix) from consideration entirely. Their primary value propositions (SSR, static generation, file-based routing) provide zero benefit here while adding hydration complexity.

Offline support/PWA?

No. Real-time collaboration is inherently online. Yjs supports offline editing technically, but implementing a proper offline experience (service workers, IndexedDB, offline UI indicators) is 6–8 hours for an unspecified feature. Yjs does provide native disconnect/reconnect resilience—buffered local changes sync on reconnect—which satisfies the spec's disconnect recovery testing without any PWA infrastructure.

SPA vs. SSR vs. static vs. hybrid?

SPA. The app has at most three views (login, optional board list, whiteboard canvas). Users log in, land on a board, and stay for the entire session. No page navigations during active use. React with Vite: fast HMR for development, optimized production builds, TypeScript out of the box, no framework opinions. Vite over CRA (deprecated). Client-side routing via React Router for minimal route structure.

Decision: React SPA with Vite + TypeScript. No SSR, no PWA. React Router for client-side routing. Deploys as static assets to Vercel CDN.

Section Summary: React SPA built with Vite + TypeScript. No SEO (canvas-rendered, authenticated), no SSR, no PWA. Yjs provides native disconnect/reconnect resilience without PWA infrastructure. Three routes max. Deploys as static assets to Vercel CDN.

11. Third-Party Integrations

External services needed?

Two required services only. (1) Anthropic Claude API: Claude Sonnet 4.5 with tool use for AI agent command parsing and multi-step execution. ~$0.004 per command. (2) Firebase Auth: Google OAuth + email/password. Free tier. No payments, email service, analytics, file storage, or other integrations needed.

Pricing cliffs and rate limits?

All services within free tier limits. Claude: 50 req/min at Tier 1 (5 concurrent users = 5 simultaneous requests, well within limits). Linear per-token pricing, no sudden jumps. Firebase Auth: unlimited Google sign-ins. Fly.io: 3 free VMs, 3GB persistent storage. Vercel: 100GB bandwidth. Server-side rate limiting (10 commands/user/minute) is the primary cost protection mechanism.

Vendor lock-in risk?

Low to moderate. Yjs/Hocuspocus: open-source, no vendor. Firebase Auth: standard JWTs, swappable with Supabase Auth or Clerk in a few hours. Claude API: tool schemas portable to OpenAI with minor reformatting. Vercel/Fly.io: infrastructure-agnostic (static files and Docker containers deploy anywhere). No high lock-in risks in the stack.

Decision: Two services—Anthropic Claude API and Firebase Auth. All within free tiers. Rate limit AI endpoint at 10 req/min/user. Low vendor lock-in across the stack.

Section Summary: Two third-party services: Anthropic Claude API (~$0.004/command) and Firebase Auth (free). All services within free tier limits. Server-side rate limiting protects AI costs. Low to moderate vendor lock-in—core infrastructure is open-source, auth uses standard JWTs, deployments are portable.

Phase 3: Post-Stack Refinement

12. Security Vulnerabilities

Known pitfalls for your stack?

Yjs/Hocuspocus: unauthenticated WebSocket connections are the #1 risk—Hocuspocus accepts all connections by default. Without the onAuthenticate hook, anyone discovering the server URL can access any board. Firebase Auth: JWT verification must be cryptographic (Firebase Admin SDK), never just decoding the token payload. Tokens expire after 1 hour; authenticate at WebSocket connection time and trust the session. Express AI endpoint: API key must stay server-side (Fly.io secrets, never in code/git). Prompt injection constrained by tool use schema (Claude can only call predefined tools). React/Fabric.js: canvas rendering is inherently XSS-safe (draws pixels, not DOM), but any UI rendering user content outside canvas must sanitize input.

Common misconfigurations?

Five priority items: (1) Hocuspocus without onAuthenticate—open access to all boards. (2) Firebase service account JSON committed to repo—credential leak. (3) CORS set to wildcard on Express—any origin can call the AI endpoint. (4) No rate limiting on AI endpoint—enables cost abuse. (5) Fly.io persistent volume not backed up—single point of failure for board data (acceptable for sprint).

Dependency risks?

All major dependencies are actively maintained. Yjs has a single core maintainer (bus-factor risk) but wide adoption. Hocuspocus maintained by Tiptap (funded company). Fabric.js recently completed v6 rewrite—pin version to avoid API instability. Firebase Auth: Google-maintained, zero abandonment risk. Anthropic SDK: stable tool use API, pin model string to avoid behavioral changes.

Decision: Four priority security implementations: (1) Hocuspocus onAuthenticate with Firebase JWT verification, (2) API key in Fly.io secrets only, (3) express-rate-limit at 10 req/min/IP, (4) CORS locked to Vercel domain. Pin all dependency versions.

Section Summary: Critical security: authenticate all WebSocket connections (Hocuspocus onAuthenticate + Firebase JWT), protect Anthropic API key (Fly.io secrets), rate limit AI endpoint (10 req/min/IP), lock CORS to frontend domain. Prompt injection mitigated by tool use schema + 20-call cap per request. UUID board IDs as unguessable access tokens. All dependencies actively maintained; pin versions.

13. File Structure & Project Organization

Monorepo vs. polyrepo?

Monorepo with npm/pnpm workspaces. Three packages: client (React SPA via Vite), server (Hocuspocus + Express), and shared (TypeScript type definitions). The shared package is the key driver—BoardObject type definitions must be identical between client (renders objects, sends Yjs mutations) and server (AI agent writes Yjs mutations). Workspaces resolve @collabboard/shared to the local directory. One type change propagates instantly to both consumers.

Feature/module organization?

Vertical slices by domain. Frontend components organized by feature area: Board/, Objects/, Toolbar/, Cursors/, Presence/, AIAgent/, Auth/. Custom React hooks serve as the integration layer between Yjs, Fabric.js, and Firebase Auth. Server code mirrors Hocuspocus lifecycle hooks (onAuthenticate, onLoadDocument, onChange). AI agent code separated into tool definitions, executor, handler, and prompts for clean extensibility.

Decision: Monorepo with pnpm workspaces (client, server, shared). Vertical feature slices. Shared TypeScript types as the single source of truth for board objects.

Section Summary: Monorepo with three packages: client (React/Vite), server (Hocuspocus/Express), shared (TypeScript types). Components organized by domain (Board, Objects, Cursors, Presence, AIAgent). Hooks as integration layer. Server mirrors Hocuspocus lifecycle API. AI code split into tools, executor, handler, prompts.

14. Naming Conventions & Code Style

Naming patterns?

PascalCase for React components and TypeScript types/interfaces. camelCase for hooks, utilities, and files. UPPER_SNAKE_CASE for constants. Discriminated union pattern for board objects (switch on obj.type with exhaustive checking). Interfaces for object shapes, type aliases for unions. No 'any'—use 'unknown' with type guards. Explicit return types on hooks and utilities. Event handlers: 'handle' prefix internally, 'on' prefix in props.

Linter and formatter configs?

Three enforced tooling layers. ESLint with @typescript-eslint/strict-type-checked: catches type coercion, unsafe any propagation, missing promise awaits. react-hooks/exhaustive-deps as error (not warn)—missing hook dependencies cause stale closures that break Yjs sync. Prettier with standard defaults (semi, single quotes, trailing commas, 2-space indent). eslint-config-prettier prevents rule conflicts between tools. TypeScript compiler: strict: true plus noUncheckedIndexedAccess (forces undefined handling on Yjs map access). VS Code settings.json committed to repo with format-on-save and ESLint auto-fix.

Decision: Strict linting and formatting enforced at every level. ESLint strict-type-checked, Prettier format-on-save, TypeScript strict mode. Conventions enforce code quality during rapid development.

Section Summary: PascalCase components/types, camelCase hooks/files, UPPER_SNAKE_CASE constants. Discriminated unions for board objects. ESLint strict-type-checked (no any, exhaustive hook deps), Prettier (format-on-save), TypeScript strict + noUncheckedIndexedAccess. VS Code settings committed to repo.

15. Testing Strategy

Unit, integration, e2e tools?

Vitest for unit and integration tests (primary, written early and continuously alongside code). Playwright for automated e2e (stretch goal—manual testing covers the same ground faster during the sprint). Unit tests are the highest priority: test early and often to catch issues before they compound. Mock external boundaries, validate real logic.

Coverage target for MVP?

Targeted coverage, not percentage-based. 95%+ on the critical path: Yjs-to-Fabric binding functions (most complex, most bug-prone), AI tool executor functions (each tool is a testable unit against a real in-memory Yjs document), board object type guards and validators, and pan/zoom coordinate math. Integration tests for core sync scenarios: object create/update/delete sync, concurrent edit resolution, reconnection state recovery. Zero unit test investment in configuration, UI components, and setup code—these are tested implicitly by e2e.

Mocking patterns?

Mock at external boundaries only: Anthropic SDK (avoid real API calls in tests—slow, costly, flaky) and Firebase Admin SDK (avoid hitting Firebase servers for JWT verification). Never mock Yjs—create real in-memory Y.Doc instances in tests. Yjs has no external dependencies and runs entirely in-process. Test data transformations as pure functions separate from rendering. Use Chrome DevTools network throttling for manual resilience testing. ~3–5 hours total testing investment during the sprint.

Decision: Vitest for unit/integration (write early and often). Mock boundaries, never mock Yjs. 95%+ coverage on sync binding and AI executor. Manual e2e with multiple browsers throughout.

Section Summary: Vitest unit/integration tests written continuously. 95%+ coverage on critical path (Yjs binding, AI executor, validators). Mock external services only (Claude SDK, Firebase Admin). Never mock Yjs—use real in-memory documents. Manual e2e with multiple browsers throughout development. Automated Playwright e2e as stretch goal. ~3–5 hours testing budget.

16. Recommended Tooling & DX

Development workflow?

Claude Code as the primary development driver. Define tasks clearly, let Claude Code generate implementation and tests, review with ESLint/TypeScript in VS Code, verify sync in multi-browser testing. Claude Code excels at: scaffolding files, implementing well-defined functions, writing unit tests, debugging stack traces, and coordinated multi-file changes. It struggles with: real-time WebSocket debugging, visual canvas layout, and Fabric.js interaction edge cases—these require manual browser testing with DevTools.

VS Code extensions?

Core four: Prettier (format-on-save), ESLint (inline errors), Error Lens (inline diagnostics on the same line), TypeScript Importer (auto-imports). Thunder Client for API testing of the AI endpoint. Disable GitHub Copilot to avoid conflicts with Claude Code.

CLI tools?

pnpm for workspace-aware package management. Fly CLI for backend deployment and production log tailing. Vercel GitHub integration for automatic frontend deploys. Firebase CLI for initial auth project setup.

Debugging setup?

Three contexts. Frontend: Chrome DevTools with React DevTools extension, WebSocket frame monitoring (Network tab, WS filter), Fabric.js canvas inspection via console (window.__canvas), and network throttling for resilience testing. Backend: structured tagged console logging ([HOCUS], [AI] prefixes) with 'fly logs' for production tailing, plus VS Code Node.js debugger for step-through on complex AI executor issues. Yjs-specific: custom document dump utility (serializes Yjs shared types to readable JSON) accessible from browser console. Multiple browser profiles (Chrome, Firefox, Incognito) as the primary manual testing setup throughout the sprint.

Decision: Claude Code drives development. VS Code with Prettier/ESLint/Error Lens for review. pnpm + Fly CLI + Vercel auto-deploy. Multi-browser manual testing throughout.

Section Summary: Claude Code as primary dev tool—generate, review, test cycle. VS Code: Prettier, ESLint, Error Lens, TypeScript Importer. CLI: pnpm, Fly CLI, Vercel auto-deploy. Debugging: Chrome DevTools (React DevTools, WS monitoring, throttling), structured server logging + fly logs, Yjs document dump utility. Multiple browser profiles for continuous manual testing.

Architecture Summary

This Pre-Search document captures the architectural decisions for CollabBoard, a real-time collaborative whiteboard with an AI agent. Every decision was evaluated against the project's core constraints: solo developer, 7-day sprint, 24-hour MVP gate, and the requirement that smooth real-time collaboration is non-negotiable.

Key Architectural Insights

1. CRDTs are the correct primitive for collaborative editing. Yjs provides conflict resolution, disconnect resilience, and state persistence at the data structure level rather than the application level. This is fundamentally superior to last-write-wins approaches and eliminates an entire class of sync bugs.

2. The AI agent is "just another user." By running the AI handler in-process with Hocuspocus, Claude's tool calls write directly to the Yjs document. All users see AI results through the same real-time sync channel used for human edits. No special AI state management, no separate sync path.

3. Two services, zero infrastructure overhead. The only external services are Anthropic Claude API and Firebase Auth. Everything else—real-time sync, persistence, hosting—runs on open-source software deployed to free tiers. Total project cost: $5–20.

4. Type safety is a speed investment, not a speed tax. Shared TypeScript types for board objects prevent the category of bugs that are hardest to diagnose in real-time collaborative systems: silent data mismatches between the sync layer, the rendering layer, and the AI agent.

Build Priority Order

Priority	Task	Est. Hours	Milestone
1	Project scaffolding + Firebase Auth	1.5	
2	Yjs + Hocuspocus setup + deploy	3	
3	Cursor sync + presence	2	
4	Infinite canvas with pan/zoom (Fabric.js)	2	
5	Sticky notes with real-time sync	2.5	
6	One shape type + object manipulation	3	MVP (24hr)
7	Shapes, frames, connectors, transforms	4	
8	Selection + operations (multi-select, copy/paste)	4	
9	AI agent — basic commands (6+ types)	4	
10	AI agent — complex/multi-step commands	4	Early (Fri)
11	Polish, docs, video, deploy hardening	4	Final (Sun)

This document was generated through a structured AI-assisted Pre-Search conversation. All decisions reflect deliberate tradeoff analysis optimized for the project's constraints: solo developer, one-week timeline, and the principle that a simple, solid multiplayer whiteboard with a working AI agent beats any feature-rich board with broken collaboration.