Pre-Search: Architecture Decision Record

CollabBoard -- Real-Time Collaborative Whiteboard

Phase 1: Constraints

Scale

- Target: 5+ concurrent users per board
- Traffic pattern: Spiky (demo-driven, presentation bursts)
- No expectation of sustained high load at MVP stage

Budget

- Zero-cost infrastructure requirement
- Fly.io free tier (3 shared VMs, 256MB RAM each)
- No paid third-party services
- Anthropic API is the only variable cost (Claude Haiku 4.5, minimal during dev)

Timeline

- Total sprint: 1 week
- MVP target: 24 hours (core canvas + real-time sync)
- Remaining days: AI agent, polish, deployment, documentation

Team

- Solo developer
- Strong JavaScript/TypeScript background
- Prior multiplayer experience from world-builder project (PartyKit-based)

Phase 2: Architecture Discovery

Hosting: Fly.io with Docker (Bun Runtime)

- **Selected:** Fly.io with auto-stop/start machines
- Dockerfile builds with Bun base image
- Machines auto-stop after inactivity, cold-start on first request
- Keeps costs at $0 during low/no traffic periods
- Single region deployment sufficient for MVP scale

Auth: Name-Based Auth

- **Selected:** Simple name-based auth for MVP
- Users enter a display name to join a board -- no passwords, no OAuth
- Firebase/Supabase auth is overkill for the gate requirement
- Can layer on real auth later without changing the core architecture

Database: In-Memory on Server

- **Selected:** In-memory state within Bun.serve() process
- Board state (shapes, positions, properties) lives in server memory
- State persists as long as the server process runs
- No external database dependency for MVP
- Tradeoff: state lost on deploy/restart (acceptable for sprint scope)

Backend: Bun.serve() Monolith

- **Selected:** Single Bun.serve() process handling everything
- HTTP routes for API endpoints and health checks
- WebSocket handler for real-time communication
- HTML imports for bundling frontend assets (no Vite needed)
- Single process = simple deployment, simple debugging

Frontend: React + Konva.js + Zustand

- **Selected:** React for UI, Konva.js for canvas, Zustand for state
- Konva.js provides a performant 2D canvas abstraction with React bindings (react-konva)
- Zustand is lightweight, minimal boilerplate, works well with external sync
- Canvas renders shapes; React renders toolbar/UI chrome

Real-Time: Native WebSocket via Bun.serve()

- **Selected:** Raw WebSocket using Bun's built-in websocket handler
- Room-based broadcast: each board is a room, messages fan out to all connected clients
- Last-write-wins conflict resolution (simplest correct approach for this scale)
- No external pub/sub layer needed at MVP scale
- Message types: shape create, update, delete, cursor position, board state sync

AI: Anthropic Claude Haiku 4.5 with Tool Use

- **Selected:** Claude Haiku 4.5 via Anthropic API, server-side execution
- 9 tools defined for canvas manipulation (create shapes, add text, change colors, arrange, clear, etc.)
- User sends natural language command, server calls Claude with tool definitions
- Claude returns tool calls, server executes them and broadcasts results
- API key stored in Fly.io secrets, never exposed to client

Phase 3: Post-Stack Decisions

Security

- WebSocket messages validated server-side before broadcast
- Shape data sanitized (no script injection via text fields)
- Anthropic API key stored in environment secrets on Fly.io
- No client-side API keys

Project Structure

```
collabboard/
  src/           -- Frontend (React, Konva, Zustand)
  server/        -- Backend (Bun.serve, WebSocket, AI handler)
  shared/        -- Shared TypeScript types (shapes, messages)
  public/        -- Static assets
  Dockerfile     -- Bun-based container
  fly.toml       -- Fly.io configuration
```

Testing Strategy

- Manual multi-window testing (open 3+ browser tabs to same board)

- Health endpoint (`/health`) for uptime verification
- Console logging for WebSocket message tracing
- No automated test suite for MVP (manual verification sufficient for 1-week sprint)

Tooling

- Bun for everything: runtime, package manager, bundler, test runner
- Single tool chain eliminates version conflicts and config sprawl
- `bun install` , `bun run dev` , `bun run build` -- consistent interface

Tradeoffs

Canvas Library: Konva.js vs tldraw vs Fabric.js

Criteria	Konva.js	tldraw	Fabric.js
React integration	react-konva (mature)	Built-in (React-only)	Community wrapper
Licensing	MIT	Custom (tldraw license)	MIT
Learning curve	Moderate	Low (opinionated)	Moderate
Customization	High	Low (opinionated UI)	High
Bundle size	~150KB	~500KB+	~300KB

Decision: Konva.js. Best balance of features, customization, and clean MIT licensing. tldraw's license introduces uncertainty for a portfolio project. Fabric.js is heavier and less React-friendly.

Real-Time Layer: Firebase vs Supabase vs Custom WebSocket

Criteria	Firebase Realtime DB	Supabase Realtime	Custom WS (Bun)
Setup complexity	Medium (SDK config)	Medium (client config)	Low (built into Bun)
Vendor lock-in	High	Medium	None
Cost at free tier	Generous	Generous	$0 (Fly.io)
Latency control	None (managed)	None (managed)	Full
Learning value	Low	Low	High

Decision: Custom WebSocket via Bun.serve(). Zero vendor lock-in, full control over message format and broadcast logic, and no external dependency. The Bun websocket handler makes this nearly as simple as using a managed service.

Sync Framework: Liveblocks vs PartyKit vs Raw WebSocket

Criteria	Liveblocks	PartyKit	Raw WebSocket
CRDT support	Built-in	Optional	Manual
Pricing	Free tier limited	Free tier	Free

Complexity	SDK overhead	Moderate	Minimal
Prior experience	None	Yes (world-builder)	Yes

Decision: Raw WebSocket. Already had PartyKit experience from the world-builder project, which informed the architecture. For last-write-wins at 5 users, CRDTs are unnecessary complexity. Raw WS keeps the stack minimal.

Persistence: In-Memory vs Postgres

Criteria	In-Memory	Postgres (Fly.io)
Setup time	0	1-2 hours
Query complexity	None (JS objects)	SQL/ORM
Data durability	Process lifetime	Persistent
MVP speed	Fastest	Slower

Decision: In-memory for MVP. Board state lives in a JavaScript Map on the server. This is the fastest path to a working demo. Persistence can be added later by serializing state to Postgres or SQLite without changing the WebSocket architecture.