CollabBoard

Pre-Search Checklist

Generated: February 16, 2026

Solo Developer (ML Engineer / Sr DS) | React + Firebase + OpenAI | 7-Day Sprint

Complete this before writing code. Save your AI conversation as a reference document. The goal is to make an informed decision about all relevant aspects of your project. Understand tradeoffs, strengths and weaknesses, and make a decision that you can defend. You don't have to be right, but you do have to show your thought process.

Phase 1: Define Your Constraints

1. Scale & Load Profile

- **Users at launch? In 6 months?**

  At launch: 5-10 concurrent users (evaluation/demo context). In 6 months: potentially 50-100 if extended beyond the course project.

- **Traffic pattern: steady, spiky, or unpredictable?**

  Spiky. Usage will cluster around demos, testing sessions, and evaluation periods. Expect bursts of 5+ concurrent users during grading, with long idle periods between.

- **Real-time requirements (websockets, live updates)?**

  Critical. WebSocket-level real-time sync is the core feature. Cursor positions need <50ms latency, object sync <100ms. This is the hardest requirement and the reason the entire stack decision revolves around real-time infrastructure.

- **Cold start tolerance?**

  Low tolerance for the frontend (must load fast for demos). Backend cold starts are acceptable if kept under 2-3 seconds since Firebase/Firestore connections are persistent once established.

✅ **DECISION:** Design for 5-10 concurrent users with headroom to 50. No need to architect for 10K+ scale.

Rationale: *This is a 7-day sprint for a course project. Over-engineering for scale would steal time from the MVP hard gates (multiplayer sync, auth, deployment). Firebase's free/Blaze tier handles this range easily.*

2. Budget & Cost Ceiling

- **Monthly spend limit?**

  $450/week available, so roughly $1,800/month total budget. However, actual infrastructure costs should be far below this since most spend goes to AI API costs and Claude subscription.

- **Pay-per-use acceptable or need fixed costs?**

  Pay-per-use is fine. Firebase Blaze (pay-as-you-go) and OpenAI API are both usage-based. Claude Pro/Max subscription is a fixed monthly cost. This hybrid model works well for a project with spiky usage patterns.

- **Where will you trade money for time?**

Everywhere possible. Firebase eliminates server management, auth implementation, and real-time sync infrastructure. Claude Pro/Max accelerates development. OpenAI API costs for the AI agent are a fraction of the time it would take to build heuristic-based board commands.

Expected cost breakdown:

Category	Estimated Cost	Notes
Claude Pro (current)	$20/month	5x usage vs free tier; sufficient for moderate daily use
Claude Max 5x (if needed)	$100/month	5x more usage than Pro; includes Claude Code + Opus access
Claude Max 20x (if needed)	$200/month	20x more usage than Pro; for heavy all-day coding sessions
Firebase (Blaze plan)	$0-5/month	Free tier covers project scale; Blaze pay-as-you-go for overages
Firebase Hosting	$0/month	Free tier: 10GB transfer/month, generous for project scale
GCP Cloud Run (FastAPI)	$0-5/month	Free tier: 2M requests/month, 360K vCPU-seconds; more than enough for project scale
OpenAI API (dev)	$20-50/week	GPT-4 function calling during development and testing
OpenAI API (production)	$5-20/month	Actual user-facing AI agent commands at project scale
Domain (optional)	$0-12/year	Firebase provides free .web.app subdomain

✅ **DECISION:** Claude Pro ($20/mo) + Firebase Blaze + GCP Cloud Run + OpenAI API. Total expected: $45-90/month on Pro, $125-170/month if upgrading to Max 5x.

Rationale: *Well within $450/week budget. Start with Claude Pro — upgrade to Max 5x only if you consistently hit usage limits during heavy coding sessions. Entire backend infrastructure stays within GCP, single billing account, single ecosystem.*

3. Time to Ship

- **MVP timeline?**

  24 hours (hard gate). The MVP must include infinite canvas, sticky notes, shapes, real-time sync, multiplayer cursors, presence, auth, and deployment.

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

  Speed wins decisively. This is a 7-day sprint. Every architectural choice should optimize for shipping speed. Technical debt is acceptable if the core features work reliably.

- **Iteration cadence after launch?**

  Daily. Day 1 = MVP, Days 2-4 = full feature set, Days 5-7 = polish, AI agent, documentation, deployment hardening.

  ✅ **DECISION:** Optimize every decision for shipping speed. Use managed services over custom infrastructure.

  Rationale: *With a 24-hour MVP gate and solo development, any time spent on infrastructure setup is time stolen from feature development. Firebase eliminates server management, auth implementation, and real-time sync infrastructure.*

4. Compliance & Regulatory Needs

- **Health data (HIPAA)?**

  No. No health data involved.

- **EU users (GDPR)?**

  No. Not targeting EU users for this project.

- **Enterprise clients (SOC 2)?**

  No. Not an enterprise product.

- **Data residency requirements?**

  None. US-based hosting is fine.

  ✅ **DECISION:** No compliance requirements. Standard security best practices only.

  Rationale: *Course project with no sensitive data. Firebase Auth handles secure authentication. Firestore security rules handle authorization. No additional compliance overhead needed.*

5. Team & Skill Constraints

- **Solo or team?**

  Solo developer.

- **Languages/frameworks you know well?**

  Strong: Python (professional ML Engineer and Senior Data Scientist background). Moderate: GCP ecosystem, general web development. Junior: JavaScript/React — functional but not fluent. Gap: No canvas library experience (Konva.js, Fabric.js, etc.).

The JS/React gap combined with no canvas experience means heavy reliance on AI coding tools for frontend code.

- **Learning appetite vs. shipping speed preference?**

  Shipping speed wins, but both React and canvas are unavoidable for a whiteboard. The ML/Python background is a major asset for the AI agent integration (OpenAI API, function calling, prompt engineering) — that part will feel natural. The frontend is where AI coding tools earn their keep. Need the canvas library with the gentlest learning curve and best AI-assisted coding support.

✅ **DECISION:** Use Konva.js with react-konva for the canvas layer. Lean heavily on AI coding assistants for all frontend and canvas-specific code.

Rationale: *Konva.js has the best React integration (react-konva), the most approachable API for someone new to canvas, and excellent documentation. Fabric.js is more feature-rich but has a steeper learning curve and worse React integration. The junior JS level actually makes react-konva's declarative JSX approach even more important — it reads closer to HTML than imperative canvas drawing. AI coding tools (Claude Code, Cursor) will generate the React/Konva boilerplate, letting the developer focus on logic and integration where Python-honed problem-solving skills transfer directly.*

Phase 2: Architecture Discovery

6. Hosting & Deployment

- **Serverless vs. containers vs. edge vs. VPS?**

  Hybrid, all within GCP. Firebase Hosting serves the static React SPA via CDN. A Python FastAPI backend runs on GCP Cloud Run for the AI agent endpoint. Same ecosystem, same billing, same IAM — no cross-platform complexity.

- **CI/CD requirements?**

  Minimal. Firebase CLI deploys frontend. Cloud Run deploys via 'gcloud run deploy' from local machine or auto-deploys from GitHub via Cloud Build. All within GCP tooling. GitHub Actions can be added post-MVP if time allows.

- **Scaling characteristics?**

  Firebase auto-scales. No manual scaling configuration needed at project scale. Firestore handles concurrent connections automatically.

Alternatives considered:

Option	Pros	Cons
Firebase Hosting + Cloud Functions	Tight Firestore integration, CDN, SSL, single deploy	Cloud Functions cold starts, Node.js only, no Python support
Firebase Hosting + GCP Cloud Run (FastAPI)	All-GCP ecosystem, Python backend, auto-scaling, containerized, same billing	Requires Docker container, slight setup overhead vs Cloud Functions
Firebase Hosting + Render (FastAPI)	Simple deploy, familiar platform, free tier	Cross-platform (GCP + Render), separate billing, free tier sleeps after inactivity
Vercel + Firebase backend	Great React DX, instant deploys, preview URLs	Two platforms to manage, potential CORS issues

✅ **DECISION:** Firebase Hosting for frontend + GCP Cloud Run for Python (FastAPI) AI agent backend. All within GCP.

Rationale: *Everything stays in one ecosystem: Firebase Hosting for the React SPA, Cloud Run for the Python backend, Firestore for real-time data. Single GCP billing account, single set of IAM credentials, familiar GCP tooling. Cloud Run supports Python natively via Docker, auto-scales to zero when idle (cost-efficient), and has a generous free tier. No cross-platform CORS headaches since Firebase and Cloud Run share GCP networking.*

7. Authentication & Authorization

- **Auth approach: social login, magic links, email/password, SSO?**

  Google sign-in as primary (2 lines of code with Firebase Auth), email/password as fallback. No magic links, no SSO — unnecessary complexity for a course project.

- **RBAC needed?**

  No. All authenticated users have equal board access. No admin roles, no permission levels.

- **Multi-tenancy considerations?**

  Lightweight. Each board is a separate Firestore document path, providing natural isolation. Users can create/join boards. No org-level multi-tenancy needed.

Alternatives considered:

Option	Pros	Cons
Firebase Auth (Google sign-in)	2 lines of code, instant setup, handles tokens	Google-only limits user base
Firebase Auth (email/password + Google)	Broader access, still simple	Need to build email/password UI
Supabase Auth	Good DX, supports many providers	Adds a second platform, not needed if using Firebase
Custom JWT + bcrypt	Full control	Days of work for auth alone, security risks

✅ **DECISION:** Firebase Auth with Google sign-in as primary, email/password as fallback.

Rationale: *Google sign-in is literally 2 lines of code with Firebase. For MVP, Google-only is sufficient. Email/password can be added in day 2-3 with Firebase's built-in UI components (FirebaseUI).*

8. Database & Data Layer

- **Database type: relational, document, key-value, graph?**

  Document database (Cloud Firestore). The whiteboard data model maps naturally to documents: each board is a document, each object is a sub-document. No need for relational joins or graph traversals.

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

  Real-time sync is critical — Firestore's onSnapshot listeners provide this natively. No full-text search needed. No vector storage needed (AI agent uses OpenAI directly, not RAG). No caching layer needed at project scale.

- **Read/write ratio?**

  Roughly 70/30. Real-time listeners generate many reads (especially cursor positions). Writes are less frequent (object creation/modification). Cursor updates should be throttled to 10-15/second to manage Firestore read costs.

Data model sketch:

boards/{boardId} → { name, createdBy, createdAt }

boards/{boardId}/objects/{objectId} → { type, x, y, width, height, text, color, rotation, zIndex, updatedBy, updatedAt }

boards/{boardId}/presence/{userId} → { displayName, cursor: {x, y}, color, lastSeen }

Alternatives considered:

Option	Pros	Cons
Firestore (document DB)	Real-time listeners, offline support, auto-scaling, generous free tier	Complex queries limited, 1MB doc size limit, eventual consistency by default
Firebase Realtime Database	Lower latency than Firestore, simpler data model	Less scalable, no offline support, single region
Supabase (Postgres + Realtime)	SQL power, real-time via websockets, row-level security	Different platform from auth, more setup
Custom WebSocket + Redis/Postgres	Full control, lowest possible latency	Massive implementation effort, must handle reconnection, scaling, persistence

✅ **DECISION:** Firestore as primary data store with real-time listeners for sync.

Rationale: *Firestore's onSnapshot listeners give us real-time sync with zero WebSocket infrastructure. The 1MB document limit is irrelevant since each object is its own document. Read costs are managed by throttling cursor updates.*

9. Backend/API Architecture

- **Monolith or microservices?**

  Two-service split, all within GCP. Firebase (Firestore) handles the real-time data layer. A FastAPI service on Cloud Run handles the AI agent endpoint. This is not a microservice architecture — it's a simple frontend + backend + database split where the backend only does one thing (AI commands).

- **REST vs. GraphQL vs. tRPC vs. gRPC?**

No traditional API needed for the core whiteboard — clients read/write directly to Firestore through the SDK. The AI agent endpoint is a FastAPI REST endpoint on Cloud Run (POST /api/ai-command). Simple, one route, no GraphQL or gRPC overhead.

- **Background job and queue requirements?**

  None for MVP. AI commands execute synchronously (user sends command → FastAPI calls OpenAI → returns structured result → frontend writes to Firestore → all clients see updates via listeners). If complex multi-step AI commands need async processing, FastAPI BackgroundTasks or Celery can be added later.

Alternatives considered for AI agent backend:

Option	Pros	Cons
FastAPI on GCP Cloud Run (Python)	Strong Python skills, same GCP ecosystem, auto-scales, generous free tier, Docker-based	Requires Dockerfile, slight cold start on scale-to-zero
Firebase Cloud Functions (Node.js)	Same platform as Firestore, easy Firestore access, auto-scaling	Cold starts (3-5s), Node.js only — junior JS skills make this slower to build and debug
FastAPI on Render (Python)	Simple deploy, Python backend, familiar platform	Cross-platform billing, free tier sleeps after 15min inactivity
Flask on Cloud Run (Python)	Even simpler than FastAPI, strong Python skills apply	No async support, no auto-generated docs, less modern

✅ **DECISION:** FastAPI on GCP Cloud Run for the AI agent endpoint. Python backend stays within GCP ecosystem.

Rationale: *Writing the AI agent in Python is significantly faster and more natural for an ML Engineer than writing it in Node.js via Cloud Functions. FastAPI provides auto-generated docs (useful for debugging), async support (good for OpenAI streaming), and type hints that feel natural from Python. Cloud Run's free tier handles project scale, and staying within GCP means single billing, shared IAM, and no cross-platform networking issues. The frontend writes AI results to Firestore directly after receiving the structured response from FastAPI, keeping the real-time sync architecture intact.*

10. Frontend Framework & Rendering

- **SEO requirements (SSR/static)?**

  None. A whiteboard app has no SEO requirements. No SSR needed.

- **Offline support/PWA?**

  Not needed for MVP. Firestore has built-in offline persistence that can be enabled later if desired, but it's not a requirement.

- **SPA vs. SSR vs. static vs. hybrid?**

  SPA. React 18 with Vite as the build tool. Pure client-side rendering. No server-side rendering, no static generation.

Canvas rendering library comparison:

Option	Pros	Cons
Konva.js + react-konva	React-native integration, declarative API, good docs, strong AI training data	Performance ceiling lower than raw Canvas/WebGL
Fabric.js	Feature-rich (built-in selection, grouping), battle-tested	Imperative API, poor React integration, steeper learning curve
PixiJS	WebGL performance, great for complex graphics	Game-engine complexity, overkill for a whiteboard
HTML DOM (divs + CSS transforms)	Simplest to implement, no canvas learning curve	Performance degrades badly at 100+ objects, limited transforms

✅ **DECISION:** React 18 + Vite SPA. Konva.js with react-konva for canvas rendering.

Rationale: *react-konva lets us write canvas elements as JSX components (<Rect>, <Text>, <Group>), which maps directly to React mental models. For someone with no canvas experience, this is the fastest path to a working whiteboard. Performance ceiling (60fps with 500+ objects) matches project requirements.*

11. Third-Party Integrations

- **External services needed (payments, email, analytics, AI APIs)?**

  OpenAI API (GPT-4 with function calling): Core requirement for the AI agent. Function calling maps directly to the tool schema (createStickyNote, moveObject, etc.). Estimated cost: ~$0.03-0.10 per AI command. No payments, email, or analytics needed for MVP.

- **Pricing cliffs and rate limits?**

  Firebase Firestore: 50K reads/day free, then $0.06/100K. With real-time listeners on cursor positions, this can add up. Mitigation: throttle cursor updates to 10-15/second. OpenAI API: GPT-4 is ~$30/1M input tokens, ~$60/1M output tokens. At project scale this is negligible.

- **Vendor lock-in risk?**

  Moderate. Firebase is the biggest lock-in risk — Firestore's real-time listener API is proprietary. Mitigated by the fact that this is a course project, not a long-term product. If it were, an abstraction layer over the data access would be worthwhile.

✅ **DECISION:** OpenAI GPT-4 with function calling for the AI agent. Firebase services for everything else. No other third-party dependencies.

Rationale: *Minimal dependency surface. Two external services (Firebase, OpenAI) — both well-documented, reliable, and within budget.*

Phase 3: Post-Stack Refinement

12. Security Vulnerabilities

- **Known pitfalls for your stack?**

  Firestore security rules: The #1 risk. Default rules are either fully open or fully closed. Must write rules that allow authenticated users to read/write their boards but prevent unauthorized access. Common mistake: forgetting to restrict writes on the presence collection, allowing users to impersonate others.

- **Common misconfigurations?**

  OpenAI API key exposure: API keys must never reach the client. The FastAPI backend on Cloud Run acts as a proxy. Verify the endpoint checks Firebase Auth tokens before processing requests. Firebase config exposure: Firebase client config (apiKey, projectId) is safe to expose — security comes from Firestore rules, not the config. Cloud Run IAM: Can restrict invocations to authenticated requests only via GCP IAM, adding an extra security layer.

- **Dependency risks?**

  Minimal. Firebase SDK, react-konva, and OpenAI SDK are all well-maintained. Pin versions in package.json. XSS via sticky note text: Konva.js renders text to canvas, which is inherently XSS-safe since canvas doesn't execute HTML/JS. Safe by default.

13. File Structure & Project Organization

- **Standard folder structure for your framework?**

```
collabboard/
├── frontend/
│   ├── public/
│   ├── src/
│   │   ├── components/
│   │   │   ├── Board/          # Canvas, Stage, Layer components
│   │   │   ├── Objects/         # StickyNote, Shape, Connector, Frame
│   │   │   ├── Toolbar/         # Tool selection, color picker
│   │   │   ├── Presence/        # Cursors, user list
│   │   │   └── AI/              # Chat input, command history
│   │   ├── hooks/
│   │   │   ├── useFirestore.js  # Real-time listeners
│   │   │   ├── usePresence.js   # Cursor + presence sync
│   │   │   └── useBoard.js      # Board state management
│   │   ├── services/
│   │   │   ├── firebase.js      # Firebase config + init
│   │   │   ├── ai.js            # Calls to FastAPI backend
│   │   │   └── board.js         # CRUD operations for board objects
│   │   ├── utils/
│   │   ├── App.jsx
```

```
│   │   └── main.jsx
│   ├── firestore.rules
│   ├── firebase.json
│   └── package.json
├── backend/
│   ├── app/
│   │   ├── main.py              # FastAPI app + routes
│   │   ├── ai_agent.py          # OpenAI function calling logic
│   │   ├── schemas.py           # Pydantic models for request/response
│   │   ├── auth.py              # Firebase token verification
│   │   └── config.py            # Environment variables
│   ├── requirements.txt
│   └── Dockerfile                   # Cloud Run deployment
└── README.md
```

- **Monorepo vs. polyrepo?**

  Monorepo with two top-level directories: frontend/ (React + Firebase) and backend/ (FastAPI + OpenAI). Single GitHub repo, two deploy targets (Firebase Hosting + GCP Cloud Run), all within GCP.

- **Feature/module organization?**

  Feature-based folders under components/ (Board, Objects, Toolbar, Presence, AI). Shared logic in hooks/ and services/. This maps to how the PRD organizes requirements.

14. Naming Conventions & Code Style

- **Naming patterns for your language/framework?**

  Components: PascalCase (StickyNote.jsx, BoardCanvas.jsx). Hooks: camelCase with 'use' prefix (useFirestore.js, usePresence.js). Services/utils: camelCase (firebase.js, board.js). Firestore collections: lowercase plural (boards, objects, presence). Firestore fields: camelCase (createdAt, updatedBy, zIndex).

- **Linter and formatter configs?**

  ESLint with React recommended config. Prettier for formatting. Tailwind CSS for styling (utility classes for speed, no CSS modules or styled-components). Configure once at project init, don't revisit.

15. Testing Strategy

- **Unit, integration, e2e tools?**

  MVP (Day 1): Manual testing only. Open 2+ browser windows, test sync constantly. Use Chrome DevTools network throttling to simulate poor connections. Days 2-4: Continue manual multi-window testing. Consider Playwright E2E tests for sync scenarios if time allows. Days 5-7: Basic unit tests for utility functions if polish time is available.

- **Coverage target for MVP?**

0% for MVP. Aspirational 20-30% for final submission. Manual multi-browser testing catches real collaboration bugs better than unit tests at this stage.

- **Mocking patterns?**

  Firebase Emulator Suite for local development (mocks Firestore and Auth locally). No need for additional mocking libraries for MVP. If Playwright tests are added, use Firebase emulators as the test backend.

✅ **DECISION:** Manual multi-browser testing throughout. Automated E2E tests only if time allows after Day 4.

Rationale: *For a solo 7-day sprint, manual testing is faster and catches real collaboration bugs better than unit tests. The PRD's testing scenarios (2 users editing simultaneously, refresh mid-edit, network throttling) are all manual/visual tests.*

16. Recommended Tooling & DX

- **VS Code extensions?**

  ES7+ React/Redux/React-Native snippets, Tailwind CSS IntelliSense, Firebase Explorer, Prettier, ESLint.

- **CLI tools?**

  Firebase CLI (deploy, emulators), Vite (dev server), npm.

- **Debugging setup?**

  React DevTools, Firebase Emulator Suite (local Firestore for offline development), Chrome DevTools Network tab for throttling tests. Key DX tip: Run Firebase emulators locally during development to avoid hitting Firestore quotas and to enable offline development.

AI coding tools (required by PRD):

Primary: Claude Code for architecture decisions, complex React/Konva component generation, and debugging frontend code — this is where AI assistance matters most given junior JS skills. Secondary: Cursor for inline code completion and rapid iteration on unfamiliar React patterns. The AI agent backend (FastAPI + OpenAI function calling) will rely less on AI assistance since Python/ML experience transfers directly to prompt engineering, API integration, and structured output handling.

Appendix 1: Architecture Summary

Final stack decision:

Layer	Technology
Frontend Framework	React 18 + Vite
Canvas Rendering	Konva.js + react-konva
Styling	Tailwind CSS
Authentication	Firebase Auth (Google + email/password)
Database / Real-time Sync	Cloud Firestore with onSnapshot listeners
Backend (AI Agent)	FastAPI (Python) on GCP Cloud Run
AI API	OpenAI GPT-4 with function calling
Hosting	Firebase Hosting
Version Control	GitHub
AI Dev Tools	Claude Code + Cursor

Why this stack wins for this project:

Single platform (Firebase) handles the hardest parts: real-time sync and auth. Firestore's real-time listeners provide WebSocket-level sync without writing a single line of WebSocket code. react-konva maps canvas rendering to React's component model, minimizing the learning curve. The Python FastAPI backend for the AI agent plays directly to ML Engineer strengths — OpenAI function calling, prompt engineering, and structured output handling are natural extensions of existing skills. GPT-4 function calling maps cleanly to the required tool schema.

Biggest risks:

1. Firestore read costs with real-time cursor listeners — mitigate by throttling cursor updates to 10-15/second and using a separate 'presence' collection with short TTLs.

2. Canvas performance at 500+ objects — mitigate by using Konva's built-in layer caching and only re-rendering dirty regions.

3. Cloud Run cold starts on scale-to-zero — mitigate by setting minimum instances to 1 ($0.05/hour, ~$36/month) or accepting 2-3s first-request delay. At project scale, cold starts are infrequent and acceptable since users expect AI response latency anyway.

4. Docker setup for Cloud Run — requires a Dockerfile for the FastAPI app. This is a one-time 15-minute setup. A simple Python Dockerfile is well-documented and AI coding tools can generate it instantly.

Appendix 2: Build Priority (Aligned with PRD)

Priority	Feature	Timeframe
P0	Firebase setup + Auth + basic React app deployed	Hours 0-3
P0	Infinite canvas with pan/zoom (react-konva Stage)	Hours 3-6
P0	Sticky notes (create, edit, move) + Firestore sync	Hours 6-12
P0	Multiplayer cursors + presence	Hours 12-18
P0	Shape support + polish for MVP gate	Hours 18-24
P1	Connectors, frames, text elements	Day 2-3
P1	Selection (single, multi, drag-to-select)	Day 2-3
P1	Transforms (resize, rotate)	Day 3
P1	AI agent (basic creation commands)	Day 3-4
P2	AI agent (complex commands, SWOT, templates)	Day 4-5
P2	Polish, animations, UX improvements	Day 5-6
P2	Documentation, demo video, cost analysis	Day 6-7

This document serves as the Pre-Search deliverable for the CollabBoard project. All decisions are defensible, tradeoffs are documented, and the architecture is optimized for a solo developer shipping a real-time collaborative whiteboard in 7 days.