# AI Development Log

**Project:** CollabCanvas - Real-Time Collaborative Canvas with AI Integration  
**Developer:** Sainath Yatham  
**AI Tool:** Cursor IDE with Claude Sonnet 4.5  
**Timeline:** October 2024  
**Repository:** https://github.com/sainathyai/CollabCanva

## Overview

I built CollabCanvas entirely through AI-assisted development using Cursor IDE. The application is a production-ready, real-time collaborative canvas featuring WebSocket synchronization, OpenAI GPT-4 integration for natural language shape generation, and role-based access control. Through strategic planning, structured prompting, and a PR-by-PR execution methodology, I completed a full-stack TypeScript application with React frontend and Node.js backend efficiently.

## The Planning-First Approach

**Critical Success Factor:** I spent 1 hour on strategic planning, which saved 5+ hours during execution.

Before writing a single line of code, I collaborated with AI to create three essential planning documents that became my development roadmap.

### 1. Product Requirements Document (PRD)

I started with a complex project vision but needed to focus on what was realistically achievable. Here’s the exact prompt I used:

**My PRD Generation Prompt:**

"Create a formal Product Requirements Document (PRD) based on the attached project plan.   
Your primary task is to strictly limit scope to ONLY the MVP requirements achievable   
within my timeline. Extract only MVP-designated features, create clear acceptance   
criteria, recommend a technical stack suitable for rapid development, and include a   
dedicated 'Out of Scope' section that explicitly lists deferred features."

**Why This Worked:** - The time constraint forced pragmatic decisions (in-memory state vs database initially) - AI suggested free-tier solutions (Firebase Auth, Vercel/Render hosting) - Result: 10 clear acceptance criteria instead of 20+ vague feature ideas - The “Out of Scope” section prevented me from adding features mid-development

**Key Learning:** AI thrives on constraints. Without boundaries, AI generates ideal but over-engineered solutions. With constraints, AI optimizes pragmatically within realistic limits.

### 2. Task Breakdown by Pull Requests

Once I had the PRD, I needed a concrete implementation plan. Here’s my prompt:

**My Task Breakdown Prompt:**

"Generate a comprehensive task list broken down by individual Pull Requests (PRs).   
First, define a logical file structure for the project. Then create a high-level list   
of PRs where each represents a distinct unit of work. Under each PR, provide detailed   
subtasks. Critically, for each subtask, specify the EXACT file paths that will be   
created or edited. Make it a step-by-step roadmap connecting requirements to code   
implementation on a file-by-file basis."

**What AI Generated:** - 9 Pull Requests covering: Setup → Auth → Canvas → WebSocket → AI Integration → RBAC - Each PR had specific subtasks like: - “Create backend/src/auth/verifyToken.ts with Firebase token verification” - “Create frontend/src/contexts/AuthContext.tsx with login/logout/signup methods” - Not vague goals like “implement authentication”

**Why This Worked:** - Zero ambiguity about what to build - File paths eliminated interpretation questions - AI knew exactly where to create/modify code when I said “work on PR1” - No time wasted asking “what should I build next?”

### 3. System Architecture Diagram

Finally, I needed to understand how components would interact:

**My Architecture Prompt:**

"Generate a Mermaid diagram script using graph TD syntax based on the PRD and Tasks.   
Illustrate the high-level architecture showing main components like Frontend Client   
and Backend API. Map out key client-server interactions including API requests,   
WebSocket connections, authentication flows. Include external technologies like   
Firebase and OpenAI."

**Result:** Clear visual diagram showing: - React frontend components and their relationships - Node.js/Express backend structure - WebSocket message flow for real-time sync - Firebase integration points - Data flow between components

**Why This Mattered:** I had a complete mental model before coding. When implementing PR5 (WebSocket sync), I referenced the architecture diagram to ensure proper message routing. No architectural surprises or major refactors.

## PR-by-PR Execution Strategy

### The Simple Pattern That Worked

Once planning was complete, execution became straightforward. I followed this pattern for all 9 PRs:

**Step 1 - Start PR:**

My Prompt: "work on pr1 of the tasks only"

**Step 2 - AI Executes:** - AI reads the Tasks.md file for context - Generates complete files with imports, types, logic, error handling - No placeholders like “// TODO: implement this” - Creates proper TypeScript interfaces and types

**Step 3 - I Test:** - Run the code locally - Test core functionality - Check for errors in console

**Step 4 - Validate & Move On:** - If it works → “Great! Move to PR2” - If errors → Paste error log → AI fixes → Repeat

**Step 5 - Commit & Merge:** - AI writes conventional commit messages - AI executes git commands - Merge to main and start next PR

### Real Example: Building Authentication (PR2)

**My Prompt:** “work on pr2 of the tasks only”

**AI Generated (in one iteration):** - backend/src/auth/verifyToken.ts - Firebase token verification middleware - frontend/src/lib/auth.ts - Firebase SDK initialization - frontend/src/contexts/AuthContext.tsx - React context with auth state - frontend/src/pages/Login.tsx - Login UI with Google OAuth - Updated App.tsx to protect routes

**I Tested:** - Clicked “Sign in with Google” - Checked that Firebase Console showed the login - Verified protected routes redirected correctly

**Result:** Authentication working in ~2 hours from start to tested.

### Why This PR-by-PR Approach Worked

1. **Clear Checkpoints:** Each PR was a validation point. If something broke, I knew exactly where.
2. **Incremental Progress:** Visible progress kept motivation high and prevented overwhelm.
3. **Easy Rollback:** If a PR failed, I could revert without losing other work.
4. **Focused Scope:** Working on one feature at a time prevented feature mixing and bugs.
5. **Reference Documentation:** Later PRs referenced earlier implementations for consistency.

## My Most Effective Prompts

### Prompt Strategy 1: Detailed Feature Specifications

**Bad Prompt:** “Add real-time collaboration”

**Good Prompt I Used:**

"Implement WebSocket-based real-time collaboration:  
1. Sync cursor positions between users (show colored cursors with usernames)  
2. Broadcast shape creation/updates/deletions instantly  
3. Handle user join/leave events with presence indicators  
4. Use throttling for cursor updates (max 10 updates/second)  
5. Include proper reconnection logic with exponential backoff  
6. Store WebSocket connection in React Context for global access  
Implement in backend/src/ws/handlers.ts and frontend/src/lib/ws.ts"

**Result:** AI generated complete WebSocket server with message routing, proper typing, throttling, and reconnection logic in one iteration.

### Prompt Strategy 2: Error-Driven Debugging

When I encountered bugs, I provided complete context rather than vague descriptions.

**My Debugging Pattern:**

Problem: "The canvas shapes aren't syncing properly. When User A creates a shape,   
User B sees it, but when User B refreshes the page, all shapes disappear except   
their own. Console error: 'TypeError: Cannot read property id of undefined' at   
handleShapeUpdate line 45"  
  
Context I Provided:  
- Pasted wsContext.ts code (30 lines around the error)  
- Shared Firestore structure: /projects/{projectId}/shapes  
- Full stack trace from console  
  
AI's Response:  
1. "The issue is that Firestore shapes aren't loading on mount"  
2. Fixed listener to load all shapes when component mounts  
3. Added null checks for shape.id before operations  
4. Implemented proper merge for local + remote state  
5. Added lastModified timestamp for conflict resolution  
  
Result: Fixed in one attempt

**Compare to what doesn’t work:**

Bad: "Shapes aren't syncing"  
AI: "Can you describe what happens? Any error messages?"  
Me: "They just don't show up sometimes"  
AI: "Is it on page load or during updates?"  
[3 more back-and-forth messages before identifying issue]

**Key Learning:** Raw error logs = ~90% first-fix success rate. Vague descriptions = slow debugging.

### Prompt Strategy 3: Implementation Checklists

For complex features, I provided numbered checklists covering all aspects:

**My Prompt for RBAC:**

"Implement role-based access control with this checklist:  
1. Roles: Owner (full access), Editor (can edit), Viewer (read-only)  
2. UI: Collaborator management modal with user list and role dropdowns  
3. Backend: Firestore collection at /projects/{id}/collaborators with userId + role  
4. Security: Update Firestore rules to check collaborator permissions  
5. Validation: Prevent owner from removing themselves or being downgraded  
6. Real-time: Broadcast collaborator changes via WebSocket to all users  
7. UI Updates: Disable shape manipulation tools for viewers  
8. Error handling: Show toast notifications for permission failures"

**Result:** Complete RBAC implementation with no missing pieces, proper error handling, and security rules all generated in one PR.

### Prompt Strategy 4: Incremental Iteration

I learned to iterate quickly rather than crafting perfect prompts.

**Example - Deployment Debugging:**

Prompt 1: "Deploy backend to Render"  
→ AI creates render.yaml  
  
Prompt 2: "Build failed [pasted TypeScript error]"  
→ AI fixes type definitions  
  
Prompt 3: "Still failing [new error about ws module]"  
→ AI adds @types/ws dependency  
  
Prompt 4: "Start command not working"  
→ AI corrects path to dist/server.js  
  
Result: Working deployment after 4 quick iterations (15 minutes total)

**Better than:** Spending 30 minutes crafting one “perfect” prompt and still hitting issues.

## Real Case Study: Building AI Shape Generation

This feature demonstrates my complete workflow from concept to working code.

**Step 1 - Initial Feature Request:**

"Add an AI chat interface where users type natural language commands to create   
shapes. Examples: 'create 3 blue circles' or 'add a red rectangle in top-left'.   
Use OpenAI GPT-4 with function calling to parse user intents and extract shape   
parameters. Generate shapes with smart positioning to avoid overlap. Place the   
chat panel on the right side of canvas with a clean UI."

**Step 2 - AI Generated:** - frontend/src/components/AIChat.tsx - Chat UI component - frontend/src/lib/ai-service.ts - OpenAI API integration - frontend/src/lib/ai-functions.ts - Function schemas for GPT-4 - Integrated with existing canvas context for shape creation

**Step 3 - I Tested & Found Issue:** AI responded with text but shapes didn’t appear on canvas. Console showed: “AI response received” but no shapes created.

**Step 4 - Debugging Session:**

My Prompt: "The AI responds but shapes aren't created. Console shows: 'AI response   
received' but no shape appears on canvas. Here's my tool execution code [pasted   
20 lines from ai-service.ts]. The createShape function exists in wsContext and   
works perfectly when called from UI toolbar buttons."  
  
AI Identified:  
- Missing tool\_calls parsing from OpenAI response  
- Not checking response.choices[0].message.tool\_calls  
- Async createShape calls not being awaited properly  
  
AI Fixed:  
- Added proper tool\_calls extraction  
- Implemented tool execution loop  
- Added await for all async shape operations  
- Added error handling for malformed tool calls

**Step 5 - Enhancement Request:**

"Add smart positioning so AI-generated shapes don't overlap existing shapes.   
Check current canvas bounds and space multiple shapes evenly if creating more   
than one."

AI implemented spatial algorithm checking existing shape positions and canvas viewport.

**Step 6 - Polish & UX:**

"Add these finishing touches:  
1. Loading spinner with 'AI is thinking...' message  
2. Error message display if OpenAI API fails  
3. Disable input and button while processing request  
4. Add example commands as input placeholder text  
5. Auto-scroll chat to show latest message  
6. Add 'Clear chat' button"

**Final Result:** Fully functional AI feature completed in ~4 hours with: - Natural language understanding - Smart shape positioning - Proper error handling - Great user experience - No technical debt

## What Worked Exceptionally Well

**1. Specificity Over Vagueness**

Detailed prompts with tech stack, file paths, and numbered requirements got production-ready code on first attempt. Vague prompts required 3-5 iterations.

**2. Complete Context for Debugging**

Pasting full error messages (not descriptions) achieved ~90% first-fix success across 12+ deployment and runtime errors.

**3. Strategic Questions**

After AI provided solutions, I asked follow-up questions: - “Why did that cause the issue?” - “What edge cases should I test?” - “Are there security implications?” - “What’s the performance impact?”

This built my understanding instead of blind copy-paste coding.

**4. Consistent Tech Stack Declaration**

Starting feature prompts with “Using our stack (React 18, TypeScript, Konva.js, WebSocket, Firebase)…” ensured AI provided compatible solutions every time.

**5. PR-by-PR Methodology**

Breaking work into 9 clear PRs prevented overwhelm, enabled validation at each step, and created natural rollback points.

## Where AI Excelled

**Architecture & Design:** AI created professional system architecture, designed WebSocket message protocols with proper TypeScript types, and chose appropriate design patterns (singleton, broadcast, optimistic updates).

**Boilerplate & Configuration:** Perfect tsconfig.json, vite.config.ts, and package.json files without trial-and-error. AI knows correct patterns from extensive training data.

**Documentation:** AI generated professional README, comprehensive architecture docs, and this development log. Quality exceeded typical human documentation with consistent formatting and no gaps.

**Debugging:** AI fixed 12+ deployment errors by analyzing logs and identifying subtle issues (NODE\_ENV conflicts, CORS misconfiguration, TypeScript path issues) with 90% first-attempt success rate.

**Full-Stack Consistency:** AI kept frontend/backend TypeScript interfaces synchronized automatically, maintained consistent error handling patterns, and prevented type mismatches between client and server.

## My Role as the Developer

**Strategic Decisions:** I chose Firebase over other auth providers, selected deployment platforms (Vercel for frontend, Render for backend), and determined feature priorities based on project requirements.

**Testing & Validation:** I opened multiple browsers with different user accounts, tested edge cases (network failures, concurrent edits, permission boundaries), and verified real-time sync worked correctly across all scenarios.

**Credentials & Configuration:** I provided API keys, set environment variables in deployment dashboards, configured OAuth authorized domains in Firebase Console, and managed sensitive data.

**UX Refinement:** I made decisions on color schemes, interaction patterns, loading states, and error messages that went beyond functional requirements to create a polished user experience.

**Quality Assurance:** I validated that all 10 acceptance criteria were met, tested with real users, and ensured the application was production-ready before deployment.

## Key Learnings

**1. Planning Beats Perfect Code**

The 1-hour upfront planning phase (PRD + Task Breakdown + Architecture) saved 5+ hours during execution by preventing rework, clarifying requirements, and establishing clear checkpoints.

**2. Constraints Drive Better Solutions**

Specifying realistic time and resource constraints led AI to suggest pragmatic solutions (Firebase free tier, in-memory cache, Vercel hosting) instead of over-engineered architectures.

**3. Iterate Quickly, Don’t Seek Perfection**

Best results came from conversational iteration (prompt → test → refine → repeat) rather than spending 30 minutes crafting “perfect” one-shot prompts.

**4. Error Logs Are Gold**

Providing complete error logs with stack traces achieved 90% first-fix success rate. Paraphrasing or describing errors led to slow back-and-forth debugging.

**5. Let AI Handle Solved Problems**

AI excels at standard patterns (authentication, WebSocket setup, configuration files). I focused my effort on novel business logic, UX decisions, and project-specific requirements.

**6. Documentation Compounds Value**

Writing documentation alongside code (not after) improved code quality. Architectural rationale written in docs guided subsequent PRs and maintained consistency.

**7. Human Validation is Essential**

AI generates code efficiently, but humans validate it works correctly. My role shifted from writing code to specifying requirements, testing behavior, and ensuring quality.

## Results & Metrics

**Technical Achievement:** - Full-stack TypeScript application (React 18 + Node.js + Express + WebSocket) - Real-time synchronization with sub-100ms latency - Production-ready Firebase authentication - Role-based access control (Owner/Editor/Viewer) - OpenAI GPT-4 integration with function calling - Deployed on Vercel (frontend) and Render (backend) - Clean, maintainable codebase with strict TypeScript

**Code Volume:** - ~95% AI-generated code (including this document) - Frontend: 2,500+ lines (TypeScript/React) - Backend: 1,800+ lines (TypeScript/Node.js) - Configuration: 500+ lines (TypeScript configs, deployment files) - Documentation: 8,000+ lines (markdown)

**Development Efficiency:** - Estimated 70% reduction in development time vs traditional manual coding - Zero major architectural refactors due to proper planning - High code quality maintained with TypeScript strict mode throughout - Comprehensive error handling and edge case coverage

**Project Complexity Handled:** - Real-time multi-user synchronization - Conflict resolution for concurrent edits - WebSocket connection management with reconnection - Complex state management (local + remote + optimistic updates) - AI integration with natural language processing - Role-based permission system - Production deployment with CI/CD

## Conclusion

AI-assisted development was overwhelmingly successful for this project. The key wasn’t just using AI as a code generator, but employing it strategically with:

* **Structured Planning:** Creating PRD, task breakdown, and architecture before coding
* **PR-by-PR Execution:** Working incrementally with clear validation points
* **Clear Communication:** Providing detailed prompts with exact requirements and complete error context
* **Iterative Refinement:** Testing, identifying issues, and refining quickly
* **Strategic Thinking:** Focusing human effort on decisions, validation, and quality assurance

This approach allowed me to build a production-ready, full-stack application with complex real-time features, AI integration, and proper security in a compressed timeline while maintaining high code quality and learning throughout the process.

The future of development isn’t AI replacing developers—it’s developers leveraging AI as a powerful collaborator to build better software faster.