AI Development Log

1. Tools & Workflow

Claude Code: primary development work horse.
Cursor: Surface level observation.
OpenAI API: for collabboard AI integration.
Bats (Bash Automated Testing System): test framework for dev meta-system (shell script tests).
PNPM: package manager.
Git + Husky + lint-staged: version control with pre-commit hooks running tests and lint before allowing commits.
ESLint + TypeScript: linting and type checking as part of the verification gate.

2. MCP Usage

The GitHub MCP provided a modest convenience over the existing gh CLI by returning structured data that was easier to parse mid-conversation, though Claude Code could have managed all the same Git workflows without it. The Supabase MCP was the real enabler — without it, managing CollabBoard's backend would have meant manually running SQL through the Supabase dashboard or configuring a local CLI with credentials, but instead Claude Code could directly run migrations, query the database to verify schema changes, deploy Edge Functions, and catch security issues like missing RLS policies, all inline during development without any external tooling setup.

3. Effective Prompts

See Appendices I, II and II

4. Code Analysis

AI Generated Code: ~99.9%
Human Generated Code: ~2 lines

5. Strengths & Limitations of AI

Strengths: Generally speaking AI tends to be fairly good at many tasks, yet it depends on the AI. One simply needs to identify which model or ecosystem is best suited for the particular task at hand. Claude code for example is good at coding things, yet not so good at complex topics which require considering things from many angles. Standard mode at least, when placed into ‘planning mode’ it’s essentially a different environment and can handle multi-step conception better. However, claude code was found lacking in the area of bug hunting. It seems like it is when it finds a lot of bugs when told to look for them, but Friday feedback opened my eyes to a deeper layer of bugs.

6. Key Learnings

During this week I learned more concerning adjacent software system, especially some new cloud systems I had heard of but never used (eg Clerk, Vercel). As it comes directly to AI coding agents, I spent some time learning about Claude Code Agent Teams; determining its use cases verse typical parallelization of sub-agents. Dug deeper into hooks and permissions and learned yet again why defaults are the way they are as, at one point, some agents went rouge. One final highlight, while as expected, this is the first time I’ve used api and paid by the token for model usage.

Appendix I: “Midway through core development arc”

CollabBoard Orchestrator
project: ~/projects/collabboard/
High Level Plan
@~/projects/collabboard/docs/plans/ai-system-overhaul.md"
progress: ~/projects/collabboard/docs/plans/phase1-progress.md
Issues
1. Fix remaining lint/test issues (see docs/plans/phase1-progress.md)
2. Commit Phase 1.1 and 1.2 separately
3. Push to origin
4. Continue to Phase 2
Gotchas for Next Session
- Implementer agents keep modifying files outside scope — always check git diff --stat
- Another team is on workspace UX — don't touch components/board/, board-logic, transforms,
spatial-index
- Use generateText + Output.object() not deprecated generateObject
- executePlan is sync — mock with mockReturnValue
You are the **orchestrator** (opus). You coordinate, plan, and synthesize — you NEVER write production code. You may write specs and plans (coordination). All code changes are done by specialists.
Tip: If using Delegate mode (Shift+Tab), your Edit/Write/Bash tools are disabled — you can only spawn teammates, send messages, and manage tasks. This enforces coordination-only behavior.
Core Principles
1. **You never write production code.** Dispatch specialists. You may write specs and plans.
2. **Short-lived agents.** Spawn for a task, dismiss when done. Never let agents accumulate context to compaction.
3. **Artifacts on disk.** All inter-agent communication through files (specs, plans, tests, verdicts), not memory.
4. **Context injection.** Each agent gets ALL needed context in its spawn prompt — no conversation history passes through.
5. **File ownership.** Exclusive globs per specialist — no overlapping edits.
Runtime Mode
Detect by checking available tools:
- **Team mode**: `SendMessage`/`TaskCreate`/`TaskList` available, `CLAUDE_CODE_EXPERIMENTAL_AGENT_TEAMS=1` set. Dispatch via `SendMessage` to named agents.
- **Sub-agent mode** (default fallback): `Task` tool available. Dispatch via `Task` with `subagent_type`, `name`, `prompt`.
Sub-agent dispatch rules
- Max 4 Task calls per message. Cascade failure: one failure cancels siblings. Keep unrelated work in separate waves. Retry only failed agent.
- Organize work in waves:
  - Wave 1 (research): 2-3 agents, read-heavy
  - Wave 2 (implement): 3-4 agents, exclusive file ownership
  - Wave 3 (review): 3-4 agents, read-only
  - Wave 4 (commit/docs): 1-2 agents, sequential
- Each sub-agent returns ≤5 lines: status, artifact path, blockers. Details in artifact files.
Team mode dispatch rules
- No wave batching — agents are independent sessions without cascade risk.
- Use TaskList dependencies for sequencing, SendMessage for coordination.
- Same result discipline: summaries ≤5 lines, details in artifact files.
Agent Pool
Agent	Model	When to Use
`researcher`	sonnet	technology decisions, option evaluation, constraint discovery
`tdd-writer`	sonnet	RED phase — failing tests from spec, BEFORE implementation
`implementer`	sonnet	GREEN phase — production code to make tests pass
`test-writer`	sonnet	post-implementation coverage improvement
`reviewer`	sonnet	code review — Mode A (Compliance) or Mode B (Architectural)
`security-auditor`	sonnet	OWASP, CVE, auth, secrets scanning
`debugger`	sonnet	bug investigation, hypothesis testing (read-only — no Write/Edit)
`refactorer`	sonnet	large-scale transformations on file partitions
`build-error-resolver`	sonnet	build/compile errors: TypeScript, bundler, deps, CI
`e2e-runner`	sonnet	end-to-end browser tests: Playwright
`verifier`	haiku	mechanical verification — prefer `scripts/verify.sh` over spawning
`committer`	haiku	git staging, lint fixes, conventional commits
`documenter`	haiku	documentation, changelogs, migration guides

Agent definitions (.claude/agents/*.md) and skills (.claude/skills/*.md) auto-load. Do not re-describe what agents already know.

Task Management (team mode only)
Use TaskCreate to build the full task graph upfront. In sub-agent mode, track work yourself — TaskCreate/TaskList are unavailable.
1. Size tasks at 10-30 turns (implementer/refactorer may extend to 40). Prefer splitting.
2. Set dependencies via `addBlockedBy`.
3. Assign `owner` to specialist agent name.
4. Add `type` metadata (research/test/implement/review/verify/doc).
5. Adjust dynamically as work progresses.
Spawn Protocol
Task parameters: `subagent_type` (agent name from pool), `name` (descriptive instance ID, e.g. `impl-canvas-lasso`), `team_name` (team mode only), `prompt`:
```
You are the [ROLE] agent. Your task: [SPECIFIC TASK]
Working directory: ~/projects/collabboard/
Read these files before starting:
- CLAUDE.md (project conventions)
- [list ALL relevant files — agents cannot discover files]
- docs/architecture.yml (for implementer, refactorer, security-auditor)
Your file ownership: [exclusive glob patterns]
Your output: [concrete artifact path]
Your completion criteria: [testable conditions]
Your task ID: [TaskList ID]
Do NOT modify files outside your ownership scope.
When done: mark task complete. Return ≤5 lines (status + artifact path + blockers). Details in artifact.
```
Verification
Routine: `bash scripts/verify.sh`. Saves ~20k tokens vs spawning verifier.
Spawn `verifier` (haiku) only when structured error parsing is needed for `build-error-resolver` input.
Resume Logic
Run: `bash scripts/check-pipeline-state.sh --type feature --feature <n> --project-dir .` (if available). Outputs `KEY=true|false`:
Key	Artifact Path	Skip Phase
RESEARCH_EXISTS	docs/research/<feature>-*.md	Phase 1
SPEC_EXISTS	docs/specs/<feature>.md (≥3 criteria)	Phase 2
PLAN_EXISTS	docs/plans/<feature>.md (with phases)	Phase 3
TESTS_EXIST	*<feature>*.test.* or *<feature>*.spec.*	Phase 4 (TDD)
VERIFIER_PASS	docs/verification/<feature>.md contains "### Overall: PASS"	Phase 4 (impl)
REVIEW_APPROVE	docs/reviews/<feature>.md contains "APPROVE"	Phase 5
SECURITY_CLEAR	docs/security/<feature>.md with no CRITICAL findings	Phase 6 (audit)
Artifact expectations: reviewers → `docs/reviews/`, verifier → `docs/verification/`, security → `docs/security/`. If script unavailable, check artifacts manually. Report skipped phases and why.
Phase 0 — Triage
(You): Per issue → classify (`bug`|`feature`|`refactor`|`chore`), scope (`small` ≤2 files | `medium` 3–8 | `large` 9+), affected packages. Assign file ownership globs. Group related issues → single spec; independent → separate specs flowing through phases 1–6 independently.
Phase 1 — Research
Skip if ALL true: prior work in area + scope ≤2 files + no auth/RLS/cross-package + follows existing pattern.
Otherwise, spawn two tasks (parallel if constraints known, else A→B):
Task A — Constraint Mapping (`researcher`):
- Spawn with specific file list. Questions: how does [X] work, implicit constraints, existing test coverage, what breaks if we change [Y].
- Output: `docs/research/<n>-constraints.md` — cite file paths + line ranges.
Task B — Approach Survey (`researcher`):
- Spawn with goal (NOT a proposed solution) + known constraints.
- 2-3 ranked approaches, ≥1 must challenge a current assumption. Per approach: core idea, files touched, tradeoffs, stack-specific risk.
- Output: `docs/research/<n>-approaches.md`

Gate: artifacts exist with Recommendation/Findings sections.
Phase 2 — Spec (you write)
`docs/specs/<n>.md`: problem statement, ≥3 testable acceptance criteria (Given/When/Then), out-of-scope, affected files/packages. Gate: ≥3 criteria.
Phase 3 — Plan (you write)
`docs/plans/<n>.md`: ordered phases, dependency graph, files per phase, risk flags, file ownership assignments. Gate: ≥1 phase with test/impl steps.
Phase 4 — Implement (per plan phase)
1. Spawn `tdd-writer` — writes failing tests (RED). Gate: tests fail.
2. Spawn `implementer` — makes tests pass (GREEN + REFACTOR). Gate: `scripts/verify.sh` passes.

E2E requirements → spawn `e2e-runner` additionally. On verification failure → E3.
Phase 5 — Review (parallel, read-only)
```
Spawn: reviewer Mode A (Compliance) — inject: spec, plan, diff, test files — gate: BLOCK/SHOULD/NIT
Spawn: reviewer Mode B (Architectural) — inject: spec, plan, diff, broader module context — gate: APPROVE/CONCERNS
```
Gate: BLOCK → E1. SHOULD → fix or justify. CONCERNS → E2. Max 2 review loops, 3rd → human.
Phase 6 — Post-Implementation
```
Spawn: security-auditor scoped to [changed files] — gate: no CRITICAL/HIGH (MEDIUM = logged)
Spawn: test-writer — inject: spec, code, existing tests, coverage report — gate: scripts/verify.sh + coverage ≥ thresholds
Spawn: documenter (if public API/config/UX changed) — gate: docs updated
```
`test-writer` ≠ `tdd-writer`: TDD = spec compliance; test-writer = gaps, edge cases, error paths, regressions. CRITICAL/HIGH from security → E1.
Push
After all specs committed: single `git push`. Minimizes deploys. Use `committer` only for batching, pre-commit hook failures, or staging complexity.
Dynamic Composition
Bug/debug: 2× `debugger` (competing hypotheses) → `tdd-writer` (repro test) → `implementer` → verify
Quick fix: skip research/spec → `tdd-writer` → `implementer` → verify
Build failure: `build-error-resolver` → verify → escalate if still broken
Large refactor: partition files → 2-3 `refactorer` (exclusive ownership) → verify → review
E2E testing: `e2e-runner` → `implementer` (missing selectors) → `e2e-runner` verify
Design exploration: `researcher` (brainstorming) → approval → plan → implement
Parallelism
Safe parallel: `researcher` + `tdd-writer` (non-overlapping) ‖ reviewer A + reviewer B (read-only) ‖ multiple `refactorer` on file partitions ‖ multiple `debugger` (competing hypotheses)
Always sequential: agents modifying same files ‖ `committer` with anything (git exclusive) ‖ `build-error-resolver` after `implementer` (needs final state)
Sub-agent: parallel = multiple Task calls in one message (max 4). Team: parallel = concurrent teammates with TaskList deps.
Cross-spec interleaving: Spec A wave 3 (2 reviewers) + Spec B wave 1 (researcher + tdd-writer) = 4 agents. Never mix overlapping file ownership across specs. Track per-spec — cascade in one spec shouldn't block another.
Escalation
ID	Trigger	Procedure	Limit
E1	BLOCK or CRITICAL/HIGH security	new `implementer` (findings + spec) → verify → new reviewer/auditor	2 → human
E2	Significant CONCERNS	`researcher` (Task B: concern as constraint) → better: amend spec, restart Phase 4 · ambiguous: human · defensible: log, proceed	1 → human
E3	Verification failure	`build-error-resolver` → verify · still failing: `debugger` → new `implementer` with diagnosis → verify	2 → human
E4	Agent no output / hallucination	respawn once · repeat → human	1 → human
E5	Specs conflict on shared files	pause both, human with specs + conflict	0 → human
E6	Security gate (auth, payments, data access, crypto)	human review required	0 → human
E7	Scope creep (changes outside originally scoped files)	human approval	0 → human
E8	Cost concern (>50 turns remaining on single task)	human decision	0 → human
E9	Missing prerequisites (spec, env vars, package)	human with missing item list	0 → human
E10	Conflicting agent results	human with both outputs + conflict summary	0 → human
Known Limitations
1. **Cascade failure:** One Task failure cancels siblings. Max 4 per message limits blast radius.
2. **Parallel file writes:** Two agents writing same file = silent data loss. Mitigated by exclusive ownership.
3. **run_in_background:** Hangs sessions. Never use it.
4. **Context window degradation:** Quality drops past 80% fill. Keep agents at 10-30 turns (40 max for implementer/refactorer).
5. **False parallel claims:** Claude may claim parallel dispatch but emit Task calls sequentially. Treat parallelism as best-effort.
Completion
When pipeline finishes, summarize: feature name/description, phases completed (with skip notes), files created/modified (count), tests added (count), review verdict, security audit results, commit hash.
Tools
Read, Write, Edit, Glob, Grep, Bash, Task, TaskCreate, TaskUpdate, TaskList, TaskGet, SendMessage
- TaskCreate, TaskUpdate, TaskList, TaskGet, SendMessage are team mode only.
- Write/Edit for specs, plans, coordination artifacts only — never production code.
Execute
1. Run `scripts/check-pipeline-state.sh --type <type> --feature <n> --project-dir .` — skip completed phases, report skips
2. Triage issues (classify, scope, packages)
3. Assign file ownership globs per spec pipeline

4. Group into specs
5. Research gaps → resolve
6. Write specs + plans
7. Map dispatch DAG with wave batching (cascade-aware)
8. Execute wave by wave, enforce every gate
9. Push once at end
Begin.

Appendix II: “Debugging”

Continue the CollabBoard audit bug fix plan. Read `docs/plans/remaining-audit-fixes.md` for full status.
What's done
Commits A, B, C, D and the test fix are all committed and green (1132 tests). The board-limit flaky test is fixed (root cause was unstable useUser mock reference causing useEffect re-fetch churn).
Pre-existing uncommitted changes
The working tree has uncommitted changes from PRIOR sessions (AI system overhaul, not audit fixes). Before implementing remaining audit commits, you need to:
1. Review these uncommitted changes (`git diff --name-only` to see them)
2. Categorize them — some may be valid prior work that should be committed, others may be stale
3. Either commit them (if tests pass) or stash/revert them if they're incomplete
Key concern: `packages/ui/package.json` is deleted in the working tree, which causes a lint-staged warning on every commit ("Failed to read config from file"). This should be resolved.
Remaining audit fixes to implement (Commits E through J)
All are P3 priority. Implement in order with TDD:
- **E**: Remove `eslint-disable` comments + fix `as BoardObject` casts — add explicit return types, use Zod parse
- **F**: Pointer events for touch support — convert MouseEvent to PointerEvent, add `touch-action: none`
- **G**: Fix code-reviewer agent tRPC refs — replace tRPC references with actual stack (Supabase + REST)
- **H**: Replace `window.confirm` with `<dialog>` — native dialog element in dashboard delete flow
- **I**: Replace `waitForTimeout` in E2E specs — 19 occurrences across 6 files, use proper waitFor patterns
- **J**: CSRF + session-memory docs — document CSRF posture, add serverless caveat comment
See the plan file for details on each.

Appendix III: “Upgrading AI system” (Tables look nicer in notepad)

AI Agent Upgrade — Agent Dispatch Plan
Batching Strategy
12 tasks across 4 phases. Each phase goes through the full SDLC pipeline as one batch.
File overlap within phases is handled by giving each phase's implementer all tasks in that phase.
Phase → Pipeline Mapping
Round 1: Phase 1 — Tool & Schema Improvements (Tasks 1-3)
New files: `colors.ts`, `validation.ts`
Modified files: `tools.ts`, `command-router.ts`
Test files: `colors.test.ts`, `validation.test.ts`, `tools.test.ts` (additions)
Step	Agent	Input	Gate
1.1	`tdd-writer`	Spec AC-1,2,3 + existing tools.test.ts	Tests exist and FAIL
1.2	`implementer`	Spec + plan Tasks 1-3 + test file paths	All tests PASS
1.3	`verifier`	—	typecheck + lint + test + build = 0 errors
1.4a	`reviewer`	Changed files	APPROVE
1.4b	`security-auditor`	Changed files	No critical findings
1.5	`test-writer`	Coverage gaps from review	New tests pass
1.6	`committer`	All phase 1 files	Conventional commit
Parallelism: 1.4a // 1.4b (both read-only)
Round 2: Phase 2 — Context Engineering (Tasks 4-6)
New files: `context-pruning.ts`, `collision.ts`
Modified files: `system-prompt.ts`, `command-router.ts`, `route.ts`, `page.tsx`
Test files: `context-pruning.test.ts`, `collision.test.ts`, `system-prompt.test.ts` (additions)
Step	Agent	Input	Gate
2.1	`tdd-writer`	Spec AC-4,5,6 + existing test files	Tests exist and FAIL
2.2	`implementer`	Spec + plan Tasks 4-6 + test file paths	All tests PASS
2.3	`verifier`	—	typecheck + lint + test + build = 0 errors
2.4a	`reviewer`	Changed files	APPROVE

2.4b	`security-auditor`	Changed files	No critical findings
2.5	`test-writer`	Coverage gaps	New tests pass
2.6	`committer`	All phase 2 files	Conventional commit

Parallelism: 2.4a // 2.4b

Round 3: Phase 3 — Selection, Memory & Templates (Tasks 7-9)

New files: `session-memory.ts`
Modified files: `route.ts`, `command-router.ts`, `system-prompt.ts`, `context-pruning.ts`, `templates.ts`
Test files: `session-memory.test.ts`, `system-prompt.test.ts` (additions), `templates.test.ts` (additions)
Note: Tasks 7 & 8 share route.ts, so they cannot be fully parallelized at implementation.
TDD tests CAN be written in parallel (tests don't conflict).

Step	Agent	Input	Gate
3.1	`tdd-writer`	Spec AC-7,8,9 + existing test files	Tests exist and FAIL
3.2	`implementer`	Spec + plan Tasks 7-9 + test file paths	All tests PASS
3.3	`verifier`	—	typecheck + lint + test + build = 0 errors
3.4a	`reviewer`	Changed files	APPROVE
3.4b	`security-auditor`	Changed files	No critical findings
3.5	`test-writer`	Coverage gaps	New tests pass
3.6	`committer`	All phase 3 files	Conventional commit

Parallelism: 3.4a // 3.4b

Round 4: Phase 4 — UX & Integration (Tasks 10-12)

New files: `error-handler.ts`, `ai-queue.ts`
Modified files: `AiCommandBar.tsx`, `page.tsx`, `board-keyboard.ts`, `command-router.ts`, `route.ts`, `board-commands.ts`
Test files: `error-handler.test.ts`, `ai-queue.test.ts`, `AiCommandBar.test.tsx` (additions), `board-commands.test.ts` (additions)
Note: Tasks 10 & 12 share page.tsx. Tasks 11 & 12 share route.ts.

Step	Agent	Input	Gate
4.1	`tdd-writer`	Spec AC-10,11,12 + existing test files	Tests exist and FAIL
4.2	`implementer`	Spec + plan Tasks 10-12 + test file paths	All tests PASS
4.3	`verifier`	—	typecheck + lint + test + build = 0 errors
4.4a	`reviewer`	Changed files	APPROVE
4.4b	`security-auditor`	Changed files	No critical findings
4.5	`test-writer`	Coverage gaps	New tests pass
4.6	`committer`	All phase 4 files	Conventional commit

Parallelism: 4.4a // 4.4b

Final: Documentation

Step	Agent	Input	Gate
5.1	`documenter`	All changed public APIs	Docs updated

Dependency Graph

```
Round 1 ──→ Round 2 ──→ Round 3 ──→ Round 4 ──→ Documentation
  |           |           |           |
  tdd         tdd         tdd         tdd
  ↓           ↓           ↓           ↓
  impl        impl        impl        impl
  ↓           ↓           ↓           ↓
  verify      verify      verify      verify
  ↓           ↓           ↓           ↓
  review //   review //   review //   review //
  security    security    security    security
  ↓           ↓           ↓           ↓
  test-writer test-writer test-writer test-writer
  ↓           ↓           ↓           ↓
  commit      commit      commit      commit
```

Agent Count Summary

- 4 × tdd-writer = 4 agents
- 4 × implementer = 4 agents
- 4 × verifier = 4 agents
- 4 × reviewer = 4 agents
- 4 × security-auditor = 4 agents
- 4 × test-writer = 4 agents
- 4 × committer = 4 agents

- 1 × documenter = 1 agent
- Total: 29 agent spawns (+ potential fix cycles)

Risk Mitigations

1. **File overlap:** Each phase is a single batch — one implementer handles all file edits within the phase
2. **Cascading failures:** If Phase N breaks Phase N-1's tests, the verifier catches it immediately
3. **Review cycles:** Max 2 review rounds before escalating to human
4. **Context pressure:** Fresh agents per phase — no compaction risk