# What running an AI-native engineering team actually cost us

**Written by** Ariella Marchuk
**Edited by** Claude

**Summary**

bsu-tool was built by seven students over five months, with AI assistance required by the project sponsor from day one. Every developer used Claude and Claude Code almost daily. This document records how each person worked, what went wrong, what we would repeat, and what the team lead observed about running a team this way. We wrote it because the next people to touch this repository will almost certainly work the same way, and their first question will be what it was like.

A note on method. Each account below is condensed from what that person wrote themselves at the end of the project. The observations were not coordinated. Several people arrived at the same practice independently which is great news!

---

## How each person worked

### Anonymous Engineer A

Anonymous Engineer A used Claude as a senior engineer partner. Each session started by giving Claude the state of the repo and the shape of the project, then a design discussion where both sides proposed and critiqued until the plan held up. The refined design went to the team before implementation. He built out one item at a time in small chunks, specifically so the implementation structure stayed trackable while Claude generated most of the code, then tested and made manual tweaks before opening a pull request.

He came away thinking Claude is an easy force multiplier and good for moving fast, but that you have to go slow enough to keep track of the work and test it. Working with Claude is a skill, and with practice you build larger and more precise things while using fewer tokens.

## Alicia Curths

Alicia used the desktop app for planning and research, and the VS Code plugin and the CLI for coding. She built a project workspace in the desktop app with linked local files, custom instructions, and scoped memory. She kept context shared at the workspace level but not across chats, to keep each session focused and save tokens. She wrote a script to convert PDFs to markdown before feeding them in, because markdown costs far less to parse.

She moved to the CLI and the VS Code plugin once implementation started. The CLI is faster and more token efficient. The plugin was worth keeping because it can hold several session tabs open at once and refer back to older sessions.

Pairing Claude Code with the GitHub CLI was the efficiency change she called out. Pulling pull requests, reviews and comments into a session gave Claude the context it needed, and she used it for the standard git workflow and for posting PR descriptions and reviews.

Her most effective pattern was to have Claude review the requirements and write a plan before it writes any code. Without that it starts coding on assumptions it never surfaces. Two prompt rules changed her outcomes: "ask when uncertain" and "ask clarifying questions before assuming". Both kept her in decisions that Claude would otherwise make silently.

She kept sessions scoped to one goal. Too little context and the model fills the gaps with wrong assumptions. Too much and it slows down and drifts. If a session's context filled up with earlier prompts and dead ends before the coding started, she closed it and opened a fresh one, carrying over only what was useful.

She also did not hand over unfettered control. She kept the setting that pauses for approval on every edit. It is slower, but it let her correct each change as the model made it instead of reading hundreds of finished lines at the end.

She summed it up by saying the effective patterns were the same ones that work without an LLM. Research, plan, build incrementally.

## Anonymous Engineer B

Anonymous Engineer B worked mostly in Claude Desktop rather than the CLI, using the CLI only to test the MCP connection workflow. Desktop showed screenshot previews, where the CLI displays an image number and makes it harder to confirm the right image was attached.

He found that with the prerequisite knowledge in place, AI made the project much more manageable, and that the developer's role moves closer to project manager. He valued the speed of fixes. He often asked Claude to review its own work from a project-wide angle, looking for missed cross-file effects.

The limitation he ran into is that repeated reviews are influenced by leading questions and can contradict earlier conclusions. Sound evaluative decisions still need engineering judgment.

## Anonymous Engineer C

Anonymous Engineer C worked iteratively rather than generating everything at once. He went back and forth reviewing each section or decision, pushed back on output that did not match the actual implementation, and updated from there. Design decisions were discussed before anything was committed, and he redirected the model when suggestions did not fit the project constraints.

Small focused exchanges kept him in control and made it easier to catch when something did not fit. Having an outlet to think out loud made the work more structured for him than working alone.

## Anonymous Engineer D

Anonymous Engineer D used Claude as a development and debugging partner for understanding existing code, troubleshooting, and running checks on the files he was working on. When he hit an error or was unsure about an implementation, he worked through possible causes with Claude and then tested the solutions himself.

He found it sped up debugging and helped him approach problems from different angles, but that you cannot blindly accept the suggestions. He still reviewed the code, ran the checks, and made sure he understood what was changing. The experience reinforced for him that you need enough technical understanding to evaluate and verify what the model gives you.

## Dylan Mills

Dylan used Claude and Codex as a development partner for understanding technical topics, organizing project ideas, and reviewing documentation before it went to the team. AI let him get a clear explanation quickly instead of working through scattered forum posts, then follow up on whatever was still unclear.

One pattern he found useful was comparing answers between Claude and Codex. Asking the second model for an opinion on the first one's answer caught unclear wording and checked whether an idea held up before he brought it to the group. He also used it to turn rough thoughts into organized writing for issue comments, PR descriptions, and architecture docs.

In a group setting the strength was speed. It made drafting specifications, identifying edge cases, and explaining technical ideas to teammates easier, and it let him participate more confidently because he could get the background before a meeting.

The downside he wrote about is the sharpest observation in any of these accounts. AI makes it easy to move ahead before the group has agreed on the direction. On the analysis engine specification, the team would have been better served by meeting to discuss what the document should look like before so much of it was written. It became the foundation for later work, and teammates had a lot of good corrections. A shared discussion first would have reduced rework and given everyone the same mental model. He disliked that AI can create a false sense that a document is done before the team has reviewed the assumptions behind it.

He concluded that AI is most useful when it helps prepare ideas for group discussion, and least useful when it replaces the discussion.

---

## What everyone found independently

**Everyone planned before generating.** Anonymous Engineer A, Alicia and Anonymous Engineer C arrived at this separately. Design discussion or a written plan came first, code second. Alicia's "ask clarifying questions before assuming" prompt exists to force the assumptions into the open before they end up in the code.

**Everyone reviewed the work in small pieces as it happened.** Alicia kept per-edit approval on and corrected each change as the model made it. Anonymous Engineer A worked in small chunks so the structure stayed trackable while the model wrote most of the code. Anonymous Engineer C worked in small focused exchanges. Anonymous Engineer D reviewed and re-ran the checks himself. Nobody who wrote about this handed over the whole task and reviewed a large finished diff at the end.

**The model tends to agree with whoever is asking.** Anonymous Engineer B found that repeated reviews shift with how the question is asked and can contradict earlier conclusions. Dylan cross-checked Claude against a second model for the same reason. Anonymous Engineer D's version is that you need enough technical understanding to verify what you are given.

**AI let one person move faster than the group had agreed to move.** Dylan's spec is the concrete case. The document was thorough, fast, and became the foundation for later work before the team had agreed on its shape. Everyone's corrections were good ones, and they would have been cheaper as a conversation held before the drafting.

**Everyone managed context deliberately.** Alicia scoped every session to one goal, converted PDFs to markdown to cut parsing cost, and closed sessions whose context had filled up with material that was no longer useful. Anonymous Engineer B chose the interface that let him verify what he was actually attaching.

One practice died and deserves its record. Generating whole files in one shot was how some of us started, and it did not survive contact with real work. The engineers who tried it moved to incremental additions with constant testing, and nobody moved back.

After the project we read Anthropic's own published guidance on working with Claude Code, writing tools for agents, and context engineering. The practices above match it closely. Nobody on the team had read those documents during development. Seven students, told only to use the tool daily, rediscovered the vendor's documented best practices by running into the failures those practices exist to prevent.

---

## The team lead's account (Ariella Marchuk)

### The context window was the central constraint

A model session does not remember the project. It remembers the current conversation, and the conversation itself has a size limit. When it gets long enough, the oldest messages are dropped first to make room for new ones, so even things said earlier in the same session can be lost. On a project that runs five months, that is the central constraint of working this way. Every session starts from nothing, and whatever was figured out last week is gone unless it was written down somewhere outside the chat.

I dealt with this a few ways. The first and largest was having Claude write handoff documents to its own next session. My working folder contains files literally named HANDOFF-FOR-NEXT-CLAUDE-SESSION. They exist because a session ended and the next one had to start knowing what the last one learned. This project invented that document type out of necessity.

The second was writing skills, which are instruction files Claude Code loads automatically when a session starts in a given folder. I wrote a team-lead skill that loads in every new session of my working folder and carries the full context of the project: what was done, when it was done, and why it was done, with dates. I also wrote a skill holding my writing rules, so documents come out readable instead of in the default AI style, and a skill encoding our specification standard. I have come to realize throughout the project, that Claude knows what it needs. You, the human, have to know what YOU need, and it has to be explicit and in writing. It cannot read your mind. Whatever is out there in actual written characters is what is fed into the context process.

The rest of the system:

- The repository became the memory instead of the chat. Decisions had to land in specs, issue bodies, pull request descriptions, or review threads, because a decision recorded only in a conversation was lost as soon as that session ended.
- What goes into a session was chosen deliberately. You cannot dump a whole repository in and expect anything good. You feed it the two files that matter, and choosing those two files is a real skill.
- I watched for long-session drift. In a long session the model starts re-arguing things that were settled two hours ago, because the settling has been dropped from its context. The answer is to end the session and write a handoff, and never to argue.

All of this is why documentation stopped being a chore on this team. We wrote specs because they were the only memory the project had, and that turned out to be a better reason than any grader could supply. One correction to how that story usually gets told however. The discipline of documentation that used to just be up to human engineers was not the AI's. The AI will happily let a decision evaporate. The discipline was the human habit of asking for the documentation every time, and then holding what came back to a readability standard, because AI-written notes that make no sense to a future human with no context are not documentation.

There is a side effect to this however that I believe would be extremely beneficial to any future team curious about the AI-native team workflow. Because everything had to be written down to survive, this project produced a complete written record: every issue, pull request, review thread, specification, even a few meeting logs. I think it would be good to go back into the repository and study in technical depth how a team adopted AI in 2026..

## Running a team through it

Claude helped me run a tight ship on the small constant work of leading: planning, tracking, review prep, and drafting the writing a team lead produces every week. But I had to act as the source of truth for everything. The AI loves to assert its opinions, push back to defend them, and introduce scope creep, and it is very talkative by default. Mental and informational overload comes fast in a role like mine, leading a whole team while helping with PR reviews and keeping the project on its path. The folder-and-skills system above is what made it survivable. It let me compartmentalize, keep what mattered in front of me, and keep the path unstrayed.

## The writing problem

The most frustrating thing I found was the writing itself. AI writing style has changed over the years these assistants have been public, but it stays nonsensical in a specific way when what you want is plain, human-readable text. It leans on implied context the reader does not have, on punchy compressed phrasing, and on volume. I refined this all term, and eventually wrote my writing rules into a skill and into Claude's memory so its replies come out in a form that works for me. Informational overload was the most common failure.

The same problem showed up across the team. People used the AI to write their PR descriptions and documentation, and in a few cases the result was frustrating, because reading it as a human it did not make sense. But if I gave the text to Claude, Claude understood it fine. The AI was acting as a middleman, writing for itself to help itself rather than writing for humans to understand.

## Reviewing with subagents, and choosing models

My review process used subagents. A subagent is a separate Claude instance the main session launches for one task. It starts with a fresh context and sees only the instructions and files the main session hands it, none of the conversation history. That fresh start is exactly what makes it useful for review. It behaves like an outside reviewer who has not absorbed the session's accumulated assumptions, and the main session even described its own subagents to me that way. Without that separation, a model reviewing its own work in the same session tends to form a self-validating feedback loop. In practice this meant things like a specification audited by four subagents that each read the real repository, and features built in one pass and then reviewed by a separate high-effort pass before I looked at the result myself.

Model choice mattered more than I expected, and learning when to use which model was its own education. What I found on this project:

- Haiku, the smallest model, was not usable for this work.
- Sonnet on a low reasoning setting was right for daily back-and-forth. Given too much room to think, the model loops and confuses itself. Human thinking gets interrupted constantly, in a brainstorming session someone cuts in and the thought resolves. The low setting gives the model that same stopping point, and the results were better for it. I think this generalizes past model settings. Constraint has produced human breakthroughs for as long as there have been humans working under pressure with worse resources than ours. These systems are trained on what we made under those constraints, so it should not surprise anyone that removing all constraint does not automatically produce better output. As the tools get more capable, a forced stopping point may be a feature a harness keeps on purpose, and how much autonomy to grant should be a decision, made per task, and never a default.
- Opus, with its large context window, was for review sessions that had to hold a lot of code at once.
- The frontier model was for orchestrated audits of the entire repository, the times I felt something was off but could not find it myself.

One more habit that paid off was asking Claude why it did something in incremental steps. I only adopted this way of working after I noticed another engineers way of using Claude Code. I saw them ask Claude something, read through its output, and ask Claude “Why did you do X?” which forced itself to actually think through why it did something on a certain assumptive basis. A long prompt with many steps works in specific cases, but when building a feature, asking why

trains it to stop assuming things silently and to stop producing output built on assumptions I never knew about. It also taught me the project's internals in depth.

## The coordination question

I want to record an honest thought I had in the final weeks, because I think it is one of the most important yet existential observation in this document. Watching people online build large projects alone on a single AI subscription, I realized this project could plausibly have been finished in days by one person with a model running continuously. There were afternoons when delegating a task, waiting for it to be finished, finding the small problems in the PR, waiting for the fixes, and re-reviewing cost more time than the work itself would have.

I do not say that to diminish what the team built, because learning to work with others and the entire process from start to finish, of leading and coordinating with a team of engineers, is extremely important skills to have. But I say it because it changes what a team project has to be. When one person and a model can produce the artifact, a multi-person project has to be justified by something other than needing many hands, and the expensive part stops being writing the code and becomes coordinating the people. bsu-tool was a great project to work on because despite me having these thoughts at times, its more advanced premise of helping other engineers in the future helped us keep our eyes on the prize.

We watched this happen to our own process. We started the term with standard advice: daily standups, report what you are blocked on. Daily standups did not end up working on this team, and the reasons are worth recording. First, the AI unblocked people before the standup arrived. The blocked state that standups exist to catch mostly stopped existing. Second, AI-assisted code moves so fast that a daily cadence reported on work that was already merged. The weekly meeting survived because its job was different, in which it was not for unblocking, but for team alignment, making sure the whole team held the same picture of the goals. My conclusion for future teams is that the rituals that manage scarcity of hands can go, and the rituals that manage shared understanding become the whole job. A team working this way should act, as a unit, like a living orchestrator that agrees on the goals, get them done well, verify that everyone understands them, and re-apply strict scrutiny to those three points every time something new is brought up. The shape I picture is one organism with separate tentacles and branches, each off doing its own work, converging back into one body at every sync meeting.

The review loop itself is the remaining time sink, and I think its future is automation. There is a project called Rolling Alice, by John Andersen, that describes a living threat model: automated CI dataflows that check work continuously instead of at review time. Five months inside this workflow convinced me that is the missing piece. If a two-person open source project asked me for the minimum process for safely taking AI-written PRs, my answer is our four automated gates plus a human reader today, and something like that living CI as soon as it exists.

## What I believe a repository needs now: a written-for-humans standard

Every document in a repository now has two audiences, humans and models, permanently. Those audiences have different tolerances. Every human engineer I speak to hold the same frustration over PRs, and how they are all written by AI models and that they dread reviewing code that was mashed together by a model, and is practically unreadable. The model reads AI-written text fine. The human becomes easily overwhelmed by the amount of writing, and the style of writing in it. This project ran into that exactly: PR descriptions and docs that made sense to Claude and not to people, with the AI acting as a middleman that writes for itself.

So I now think a repository needs a written-for-humans standard the way it needs a license. I ended up writing one for myself, as the skill holding my writing rules, and I think the general version belongs in any project's contributing guide. The enforcement that fits it is a self-assessment before posting a PR. Can you explain what this change does and why, in your own words? If you did the work and understand it, that costs nothing and is not a blocker. If the AI did it and you were just the person hitting push, the question blocks you, and becomes irritating. And this also becomes the honest version of the rule underneath all of this, which is for you to assess, truthfully, what you are using the AI for. Your answer to this as a team member, and your ability to be truthful with yourself is directly relevant to how successful and fulfilling a project that uses AI in everything could be.

## The generational question

An engineer who graduated in 2017 was forced to learn the hard, big, and annoyingly small things first, and then these tools arrived. Their judgment plus the tools is what makes them so effective now. Students forming today are not dumber and are not skipping the hard things, but the deep-rooted concrete understanding those earlier engineers built is harder to develop when this much help is available from the first day, and when there feels like there is no time to keep up with how quickly everything is evolving. So sitting down at your desk and learning the hard annoying things before asking for help from the AI feels 10x more difficult. And the speed of the field is its own pressure. You are behind the moment you stop keeping up, which is a frightening thing to carry as a new graduate.

Daily use cut the other way for us. Because we used Claude almost every day of development, we were never stale on the state of the tools. And this project was not buildable by seven part-time students without it. I hold both of those alongside the honest cost of the logic and review passes we did ourselves are not the same as writing everything from scratch, and we cannot fully know what we did not learn.

Where our own baseline came from is important to mention, because it is the thing educators should protect. It came from previous group projects, and from the start of the first term, when we spent weeks researching USB before writing anything. What I would add for the next team is even more of that, done by hand, two meetings researching how USB reverse engineering is

actually practiced, one meeting attempting to reverse engineer something ourselves with no tool, a session on what we learned, and a discussion of which workflow would help a real user most. The manual attempt is the point. You cannot judge what a tool automates until you have felt the work it automates.

## Leading a team this way, and whether I would again

The thing AI helps with most is doing the writing, and as a team it was very easy for things to become unorganized if I did not keep them in check. I wrote most, if not all, of the issues the team made PRs for, and I believe that if I had not, the project could easily have gone out of scope. We learned quickly how important documentation discipline is, most sharply around the milestone-three spec and again as the project reached its most important milestones. I had to keep detailed documentation on the state of things while balancing what the repository actually said online against what my own working folder held.

I was surprised by how much can get done with this help, and it gave me and the team breathing room to refine and to dream bigger about what the project could become in the very end. Leading was still a lot, and I will not say it was easy, because it was real responsibility and real work even before the AI was in the picture to keep everything in check, and I will not claim I did the best job, only the best job possible of me.

One thing I keep thinking about too, is being told from the start that this was an AI-native project shaped my posture toward it. If we had just been handed the tools without the framing, I think I would have felt freer to lead from my own human judgment. I did use that judgment, but at a remove. A team lead with twenty years of experience who picks these tools up gets a great addition to a workflow that already exists. A student lead learning both at once has the tools shaping the workflow while it forms, and my team may not always have gotten the best version of me because of it.

If I set up a team like this again, three things go in on day one. First, a norms conversation before any work starts: what we agree is good AI use, bad AI use, and dishonest AI use, what we want a PR to look like, and whether anyone on the team quietly resents the tools, because plenty of people do and it surfaces eventually. Mandating AI changes a team differently than allowing it, and a mandate without agreed norms leaves everyone guessing at the line. Second, a person who records every meeting and produces detailed minutes with actionable points. The AI is each person's context keeper inside their own sessions, but nothing keeps the context between people unless a human owns it. Third, a person who owns how the project looks to the outside: the README, the visuals, the customer-facing surface, kept human-made and only AI-assisted. A designer's eye is hard to train and the audience for those surfaces is entirely human. Related to that, I would ask that PR descriptions be written by people from now on. Reviewing a jumbled mix of AI code and AI prose is a headache twice over, and the description is where the human proves they understand their own change.

My honest verdict is that from this point on it will not be possible to lead a software team without AI in the picture. It is here to stay, and the only way through is to keep learning it. What I want next is more reps on my own front, as well as my teammates. To spend time on more teams, more leading, and more of the advice I have been collecting from seasoned engineers about the human parts of this job, which are still, as our sponsor put it, most of the job.

### Building a tool for an AI kept us honest

We were using AI to build the tool, and at the same time building a tool for an AI to use. The second thing disciplined the first. Every tool returns structured data. Every claim traces to a specific packet range in a specific file. The agent cannot go around the tools and read the capture itself, so it cannot hand-wave, which meant we could not hand-wave either. The constraint improved the product. I did not expect to be able to say that.

---

## Review, safety, and what almost got through

- Nothing AI-generated merged without a human reading it. Every change went through review by a person who did not write it, plus four automated gates on every push.
- The thing that catches AI-written code specifically is that it is plausible. It looks right. The two bugs that cost us the most time this term both passed their own tests. What caught them was a person going and testing the actual claim rather than trusting the test.
- USB captures contain real data from real devices. Our reference captures come from a fingerprint reader. Every capture in the repository is sanitized before it goes in. We also keep device serial numbers out of device identifiers, because an identifier propagates into every packet record and every serialized session, and a serial number is a per-unit fingerprint.

---

## Lessons learned

The useful pattern everybody landed on independently was to plan before generating anything. Ask for a design or an implementation plan, argue with it, and only then let it write code. When you skip that step it starts coding against assumptions it never told you about.

Review each edit as the model makes it. Approving changes one at a time catches things that reading five hundred finished lines at the end does not.

The model tends to agree with whoever is asking. Asking it to re-review its own work gives you answers shaped by how you asked, sometimes contradicting what it said an hour earlier. You still need enough engineering judgment to know which answer was right.

The honest downside is that AI lets one person get ahead of the group. Our analysis engine spec was written fast and thoroughly, and it became the foundation for a lot of later work, but the team had a pile of good corrections that would have been cheaper as a conversation held before the drafting.

Review did more work than we expected. The two most expensive bugs of the term were both found by a person testing something the tests claimed to already cover.

If we did it again, we would go spec-first from week one instead of week six, and we would talk about the shape of a document before somebody drafts the whole thing. Nobody sat down together to think about what our load-bearing spec should contain before it was written, and that is the failure the shape conversation prevents.

And the lesson under everything else in this document is about honesty. The workflows this team discovered are, we believe, close to how much of this work will be done from now on. What made them work was never the model. It was each person's willingness to be honest with themselves about what they were using it for, to keep learning on purpose, and to hold their own discipline about understanding what they shipped. No playbook can install that. A person with no real interest in the work, who puts the strongest model on the project and walks away, learns nothing and contributes nothing a model alone would not have, and there is no rule anyone can write into a contributing guide that prevents it. The model is trained on what people made and goes where the person steers it. It is exactly as good as the values of the person using it. Teams that agree on those values out loud, before the work starts, will get the version of this workflow worth having.