Magnitude 9 Earthquake

Brian Edwards

January 5, 2026

Contents

Stop building your resume. Stop building your career. Those concepts no longer apply.

All the projects you have been telling yourself you do not have time for—start them. All the ideas your mind generates that you immediately shut down with "maybe someday" or "when I retire" or "that is not practical"—stop shutting them down. Start responding to those intuitions. Sit down and implement them. Bring them to life.

This essay is for anyone who uses a computer for work and wants to be an active participant in the future rather than a passive one. Software developers will find

the most immediately applicable techniques here, but the principles extend far beyond code. Writers, analysts, researchers, managers, teachers, lawyers, doctors—anyone whose work involves manipulating information—faces the same transformation. The earthquake does not discriminate by job title.

The goal is not to "learn vibe coding." That skill set will be obsolete in six months anyway, replaced by something more powerful. The goal is to fulfill your intuitions. To bring what is in your mind into the physical world. To close the gap between imagination and reality.

And here is the promise: as you do this daily, your mind will open up. The ideas will come faster. The projects will multiply. You will have trouble keeping up even with the dramatic speedups that AI-assisted development provides. The bottleneck shifts from execution to imagination—and when you stop suppressing your imagination, you discover you have more ideas than you ever realized.

For years, maybe decades, you have been trained to be "realistic." To focus on what pays. To build credentials and climb ladders. To defer dreams to retirement. That training served a world where execution was expensive and ideas were cheap. Now execution is becoming cheap and ideas are becoming the constraint. The skills of imagination, of creative vision, of bringing new things into existence—these are what matter now.

The concrete evidence for this shift comes from a project called ghost-note, built in thirty-six hours, one hundred eighty-seven commits, one hundred thirty-five thousand lines of TypeScript. The human—me—spent three to four hours prompting. The rest was autonomous AI development. But ghost-note is not the point. Ghost-note is an example, a proof of possibility, a case study. The point is what becomes possible when the cost of execution collapses.

But first, some context from someone who has been watching this space longer than almost anyone.

On December 26, 2025, Andrej Karpathy posted something that would be viewed over sixteen million times. Karpathy is not a random voice on the internet. He was the director of artificial intelligence at Tesla, where he led the team building

the neural networks that power Autopilot. Before that, he co-founded OpenAI. He created one of the most popular deep learning courses ever taught, at Stanford, and the YouTube lectures have been watched by millions. His PhD thesis on image captioning helped establish techniques that are now standard in the field. When Karpathy speaks about programming and AI, the technical world listens, because he has been at the center of these developments for over a decade.

He is also the person who coined the term "vibe coding" in February 2025. The phrase captured something that developers were already experiencing but did not have words for: a mode of programming where you describe what you want in natural language and the AI writes the implementation. You focus on the vibe—the feel, the intention, the outcome—rather than the syntax and structure. The code emerges from conversation rather than from typing.

So when Karpathy, of all people, says he feels behind, it matters.

What he said was this:

> I've never felt this much behind as a programmer. The profession is being dramatically refactored as the bits contributed by the programmer are increasingly sparse and between. I have a sense that I could be 10X more powerful if I just properly string together what has become available over the last ~year and a failure to claim the boost feels decidedly like skill issue. There's a new programmable layer of abstraction to master (in addition to the usual layers below) involving agents, subagents, their prompts, contexts, memory, modes, permissions, tools, plugins, skills, hooks, MCP, LSP, slash commands, workflows, IDE integrations, and a need to build an all-encompassing mental model for strengths and pitfalls of fundamentally stochastic, fallible, unintelligible and changing entities suddenly intermingled with what used to be good old fashioned engineering. Clearly some powerful alien tool was handed around except it comes with no manual and everyone has to figure out how to hold it and operate it, while the resulting magnitude 9 earthquake is rocking the profession. Roll up your sleeves to not fall behind.

Read that again. The man who named the phenomenon, who has been at the forefront of AI development for a decade, who understands these systems at a deeper level than almost anyone, just admitted he feels behind. Not slightly behind. Not occasionally behind. He has never felt this much behind as a programmer. If Karpathy feels this way, what does that mean for everyone else?

This essay is about what that means. It is about what happens when you take the earthquake seriously. It is about a real project, built in thirty-six hours, with one hundred eighty-seven commits and one hundred thirty-five thousand lines of TypeScript, while the human spent only three to four hours actually prompting. It is about the death of career-building and the birth of something stranger. It is about why you should consider quitting your job.

But first, we need to understand what Karpathy is actually saying. Every phrase in that post carries weight.

Parsing the Earthquake

"The profession is being dramatically refactored." Notice the word choice. Refactored. Not evolved. Not improved. Not disrupted. Refactored. That is a programming term. When you refactor code, you restructure it fundamentally while preserving its external behavior. The structure changes. The foundations move. What was on top is now on the bottom. What was central becomes peripheral.

The word "disruption" gets thrown around constantly in technology discussions, but disruption implies an outside force attacking an existing structure. Refactoring is different. Refactoring is internal restructuring. The system is changing itself from within. Karpathy is not saying some external force is attacking programming. He is saying programming's internal architecture is being rebuilt while we are still standing on it. The floor is moving. The walls are moving. The ceiling is moving. And we are inside the building trying to keep working.

"The bits contributed by the programmer are increasingly sparse and between." This is the core observation. The human contribution is shrinking. Not in importance, perhaps, but in volume. The programmer used to write most of the code.

Now the programmer writes prompts and the AI writes code. The ratio is inverting.

I can tell you exactly what that ratio looks like. In the project I am about to describe, the session log contains four thousand sixty-seven messages from the AI and sixty-two messages from the human. That is a ratio of roughly one to one hundred thirty. The human bits are indeed sparse.

But "sparse" does not mean "unimportant." Consider an orchestra conductor. The conductor produces no sound. Every note comes from the musicians. Yet the conductor shapes the entire performance—the tempo, the dynamics, the interpretation. The conductor's contribution is sparse in volume but determinative in direction. This is the emerging role of the programmer: not the source of the code, but the conductor of the system that produces it. Sparse but decisive.

"I could be 10X more powerful if I just properly string together what has become available." The capability exists. Right now. Today. The tools are here. The models are here. The workflows are documented. The only thing missing is the skill to use them. Karpathy calls failure to access this power "decidedly a skill issue." Not a tool issue. Not a money issue. Not an access issue. A skill issue. The implication is uncomfortable: if you are not ten times more productive than you were a year ago, you have only yourself to blame.

This is worth sitting with. Ten times more productive. Not ten percent. Not fifty percent. Ten times. An order of magnitude. Karpathy is not talking about incremental improvement. He is talking about a transformation in what is possible. And the barrier is not technology—the technology exists—the barrier is skill. The skills to prompt effectively, to manage context, to orchestrate agents, to verify output, to recover from failures. Skills that did not exist a year ago because the tools that require them did not exist a year ago.

"A new programmable layer of abstraction." Then comes the list. Agents. Subagents. Prompts. Contexts. Memory. Modes. Permissions. Tools. Plugins. Skills. Hooks. MCP—that is the Model Context Protocol, Anthropic's standard for connecting AI assistants to external systems. LSP—the Language Server Protocol, which provides code intelligence. Slash commands. Workflows. IDE integrations—

meaning how your coding environment connects to AI, which stands for Integrated Development Environment.

Each item in that list is a skill to learn. Agents are AI systems that can take actions, not just generate text. Subagents are agents spawned by other agents to handle specific tasks. Prompts are the instructions you give to shape behavior. Contexts are the accumulated history that informs responses. Memory is persistence across sessions. Modes are different operational states—coding versus chatting versus planning. Permissions are what the AI can and cannot do on your system. Tools are external capabilities the AI can invoke. Plugins are extensions. Skills are learned capabilities. Hooks are event-triggered automations. Each has its own learning curve, its own best practices, its own failure modes. And the list keeps growing. This is the new stack. Below you still have the old stack: programming languages, frameworks, databases, deployment, networking, security. The new layer sits on top of all that, and mastering it is not optional.

The traditional software stack accumulated over decades. Networking in the seventies. Databases in the eighties. The web in the nineties. Mobile in the two thousands. Cloud in the twenty-tens. Each layer added complexity, but the pace of addition was slow enough that practitioners could absorb it. You had years, sometimes decades, to learn each new layer. The new agential layer arrived all at once, in months, and it is still expanding rapidly. There is no time for gradual absorption.

"Fundamentally stochastic, fallible, unintelligible and changing entities." This is perhaps the most important phrase. Stochastic means probabilistic, not deterministic. You give the same input twice and you might get different outputs. The randomness is not a bug—it is intrinsic to how these models work. Temperature settings control how random the output is, but even at temperature zero, which theoretically produces deterministic output, subtle variations can occur.

Fallible means it makes mistakes. It hallucinates. It confidently produces nonsense. It invents functions that do not exist, API endpoints that were never built, libraries that no one wrote. It can write plausible code that compiles and runs but does the wrong thing. The confident tone does not correlate with accuracy.

Unintelligible means you cannot fully understand why it does what it does. The internals are opaque. These models contain billions of parameters, and no one can trace why a particular input produces a particular output. We can probe behavior, run experiments, characterize tendencies, but we cannot open the hood and follow the logic the way we can with traditional software. It is a black box that happens to produce useful output most of the time.

And changing means what works today might not work tomorrow. The models update. The tools evolve. Anthropic releases new versions. OpenAI releases new versions. Google releases new versions. The behavior shifts. The prompts that worked last month may work differently this month. There is no stable target to aim at.

This is categorically different from traditional engineering. When you write code, you expect it to behave the same way every time. When you debug, you can trace the logic step by step, inspect variables, follow the execution path. When you learn a tool, you expect that knowledge to last—the tool might get new features, but the core behavior remains stable. None of that applies anymore. You are working with entities that are probabilistic, error-prone, mysterious, and unstable. The engineering mindset has to adapt or break.

Traditional engineering assumes you can build reliable systems from reliable components. You test each piece. You verify behavior. You compose verified pieces into verified wholes. But how do you build reliable systems from unreliable components? How do you compose stochastic pieces into deterministic wholes? This is the new engineering challenge. The answer involves verification, redundancy, human oversight, probabilistic reasoning, graceful degradation—techniques that exist but are not yet standard practice in most software development.

"A powerful alien tool handed around except it comes with no manual." Nobody knows how to hold this thing. Not the people who built it—Anthropic publishes engineering blogs with best practices, but those practices keep evolving as they discover new things. Not the people who use it daily—practitioners share tips and workflows, but the collective knowledge is fragmented and often outdated. Not the experts writing blog posts about best practices—those best practices have a

half-life of months, not years. Everyone is figuring it out in real time, including the people creating the tools you are using to figure it out. The map is being drawn while the territory shifts.

The manual metaphor is precise. Tools historically came with documentation. You could read how to use a hammer, a lathe, a programming language. The behavior was stable enough that documentation made sense. AI assistants resist documentation because their behavior depends on context, changes with updates, varies with prompting style, and surprises even their creators. The documentation that exists is partial at best, misleading at worst. Learning happens through experimentation, not reading.

"Magnitude 9 earthquake." On the Richter scale, a magnitude 9 earthquake is catastrophic. For reference, the 2011 Tōhoku earthquake in Japan was magnitude 9.1. It triggered a tsunami that killed nearly twenty thousand people, caused the Fukushima nuclear disaster, and inflicted damage estimated at over two hundred billion dollars. A magnitude 9 event is not a tremor. It is not a shake. It is the kind of event that changes geography permanently. Cities fall. Coastlines move. The landscape before and after are different places.

Karpathy is not exaggerating for effect. He is describing the scale of transformation he sees happening. Not a small shift. Not an incremental change. A restructuring so fundamental that the profession before and after are different professions.

The Before and After

September 2025 marked a turning point. Claude 3.5 Sonnet had arrived in June 2024 and impressed people with its coding ability. It could write functions, explain code, debug issues. But it still felt like an advanced autocomplete, a smart assistant, a tool you used for specific tasks. It was impressive but bounded.

Claude 4.5 Sonnet, released in September 2025, changed the game in ways that are difficult to convey without experiencing them. Extended thinking allowed the model to reason through complex problems before responding, working through multiple steps internally before producing output. This meant it could handle multi-

file refactors, understand architectural decisions, plan implementations, reason about tradeoffs. The thinking was not just faster—it was categorically deeper.

Agentic behavior improved dramatically. The model became better at taking actions, not just answering questions. It could run commands. It could edit files. It could iterate on its own output. It could recover from errors. It could pursue goals across multiple steps without constant hand-holding. The difference was not incremental—it crossed a threshold into genuine autonomy.

Claude Code, the command-line interface for working with Claude on software projects, reached maturity. Previous versions were rough, requiring workarounds and patience. The September release was polished, stable, capable. It could manage context intelligently, persist conversation history, handle file operations smoothly, integrate with development workflows naturally. It became a tool you could rely on for hours of continuous work.

The Chrome extension appeared, allowing Claude to interact with web pages. Code mode in the desktop application launched, providing an alternative interface optimized for development work. Suddenly there was an ecosystem, not just a model. Multiple interfaces for different contexts. Multiple ways to access the same underlying capability. The flexibility to work however you preferred.

The Anthropic engineering blog started publishing detailed guides. “Effective Context Engineering for AI Agents” explained how to manage the conversation history that AI models use to understand your project—the context window that fills up and needs to be managed. “Claude Code Best Practices” laid out workflows for agentic coding that had been discovered through extensive internal use. “Advanced Tool Use” introduced features that reduced token overhead by eighty-five percent while maintaining access to thousands of tools—a technical improvement that dramatically lowered costs and increased speed.

These blog posts were not marketing. They were genuine knowledge transfer from practitioners who had spent months figuring out what worked. They documented failure modes, shared recovery strategies, explained architectural patterns for multi-agent systems. The Anthropic engineering team was using their own tools to

build their own tools, and they were sharing what they learned.

A movement emerged. People started calling it vibe coding—Karpathy's term from February 2025 had caught on. The idea was simple but radical: instead of writing code yourself, you describe what you want and let the AI write it. You focus on the vibe—the feel, the intent, the desired outcome—and the model handles the implementation. Tutorials appeared on YouTube and Substack and Medium. Blog posts proliferated. Nate B Jones started publishing videos about agent-focused business applications. People shared their workflows, their prompts, their custom slash commands, their configuration files.

The author of this essay has been living in this world since September 2025. Eight to twelve hours a day, every day, learning by doing. Not reading tutorials—building real projects. Not watching videos—experimenting with prompts. Figuring out when to let the AI lead and when to guide it firmly. Learning to manage context windows that fill up and need to be compressed. Developing intuitions for what works and what fails.

Before September 2025, the work was smaller. March Madness competition entries—prediction models that could be built in a weekend. Chinese novel translations—complex but contained projects. Exploratory experiments—seeing what was possible without betting on it. After September, everything changed. The model upgrade made vibe coding large projects possible. Projects that would take a team months could be attempted in days. Projects that would require specialists in multiple domains could be integrated by a single person orchestrating AI assistance.

Ghost-note is one of those projects. It is not the only one. In four months, I have built multiple production systems, including Folk Care, an open-source home healthcare management platform that reached production readiness in twenty-eight days. But ghost-note is the one with the clearest documentation, because the entire development process was captured in session logs. Every prompt. Every response. Every file created and modified. A twelve-megabyte record of thirty-six hours of vibe coding.

Let me show you what that looks like.

The Vision Prompt

The project started with a single message, sent at 4:33 PM on January 3, 2026:

> We are creating an app that helps poets translate their poems into songs, adjusting lyrics and developing a vocal melody. The end user final interface and workflow is Claude Code CLI, Code mode in the Claude Desktop app (Mac), and the Claude Chrome extension. So the end user interacts directly with this repository. So I work on developing the app with you. Then use you to create songs from poems. It is very important to use free and hopefully open source packages and programs. I envision a web interface and therefore the Claude Chrome extension is a big part of interacting. There will be a web app component that is the multiple media display. Seeing the lyrics and revisions, playing the vocal melody, recording the user singing the lyrics to the melody. The original poem will be easily accessible and the versioning and diff system will be easy to use and use great UX. First step, write this down so you memorize it across sessions. Research the web (it is Jan 2026, so recent web searches should use year 2025) this field has been changing rapidly so research the latest vibe coding Claude Code CLI Claude Code in the desktop app, and the Chrome extension best practices this is a complex system, recent blog posts on the Anthropic engineering blog are fantastic. One this is avoiding the overhead of MCP servers, another is the battle with the context window, when it gets compacted how to get Claude to stick with a broad app project perspective and not get narrowly focused on the current task without context. Document all this in the best ways you discover on the web and setup our project using an opinionated structure and Claude Code features that you determine are the latest best practices from your research.

Notice what this is and what it is not. It is not a specification document with detailed requirements. It is not a list of user stories in the conventional agile format.

It is not a technical design document specifying components and interfaces. It is a vision statement mixed with meta-instructions about how to proceed.

The human is not saying "write me this code." The human is saying "research best practices, then set up the project using what you discover." The research itself is delegated. The decision-making about project structure is delegated. The documentation of findings for future reference is delegated. The human provides intent and constraints. Everything else emerges from AI exploration.

This is architectural thinking at a higher level of abstraction than traditional software specification. The human says what they want to achieve and what values matter—free and open source, good UX, persistence across sessions—and the AI figures out the concrete implementations that satisfy those constraints. The human does not need to know which packages exist, which are maintained, which work together. The AI researches, compares, decides.

There is also explicit meta-instruction about managing the AI itself. "Write this down so you memorize it across sessions." This acknowledges that sessions end, contexts compact, and information is lost. Documentation becomes the AI's external memory. The human is designing not just a software system but a system for building software with AI, including strategies for managing AI limitations.

Fourteen minutes later, the second prompt arrived:

> This looks great and I like those dependencies, be sure to use package managers to install or update to the latest stable versions of all packages, do not rely on your training data in this regard, it is outdated. So building this app obviously requires extensive music theory and knowing about tech that allows for it to work. I would like for you to research the latest (again year 2025), what goes into translating a poem to song lyrics? How to analyze a poem and try to stay true to it, and match its syllables, pronunciation, strong/emphasized syllables, line length, music genre, meter, rhyme scheme, we will need to do this mostly with traditional software which is very good at quantitative analysis and build custom code around packages, and then you will use/drive/maintain/read

> output from the software we write (you excel at a more qualitative understanding.) Define the problem we are solving from first principles, and then look blue sky for solutions. We can develop an elaborate system of our own custom code. Write down your findings using our structure and methodologies so they are reliably available across sessions and contexts.

The human is now directing domain research. What does it mean to translate a poem into song lyrics? What are the technical components? This is not territory the human knows well—they are learning alongside the AI, using the AI's research capabilities to understand a domain.

Notice the explicit acknowledgment of AI limitations and strengths. "Do not rely on your training data in this regard, it is outdated." The human understands that AI knowledge has a cutoff date and that package versions change. They direct the AI to use package managers to get current information rather than relying on potentially stale memory.

There is also a sophisticated division of labor. The human explicitly distinguishes between quantitative analysis—syllable counting, stress pattern detection, rhyme scheme identification—which traditional algorithmic software handles well, and qualitative understanding—interpreting meaning, preserving emotional tone, making aesthetic judgments—which the AI handles well. This is not delegating everything to AI. It is recognizing that some problems are better solved with deterministic code, and the AI should build and use that code, while applying its own capabilities where they add value.

"Define the problem from first principles, then look blue sky for solutions." This is permission for creative exploration. Do not just implement an obvious approach. Understand the deep structure of the problem. Consider unconventional solutions. The constraint is effectiveness, not conventionality.

Nineteen minutes later, the third prompt changed everything:

> Come up with a detailed backlog of tasks that need to be performed taking us all the way to the full vision of the product. Document this in

> a best practices way we have determined for this repository. Plan to act as the dev manager assigning tasks to subordinate Claude Code instance that one-shot the implementation of the task. Use the gh CLI. Create GitHub issues for the backlog. You will assign one issue to each single-shot Claude Code worker instance. Enable the Claude GitHub integration where Claude running on GitHub performs code reviews. You will watch for these code reviews upon work instance completion and fire off additional one-shot workers telling them to improve the feature branch code based on the comment. Manage multiple worker sessions at a time (I am saying Claude Code instances, but maybe this is done in this single Claude Code session depending on isolation features.) Use feature branches and git workspaces to provide isolation on this one checked out directory on this one machine. Organize a way for the instances to have their own isolated instance of shared resources (port numbers, databases/schemas if applicable). Document all this. Prepare this repo for these two distinct roles of agents so the worker agents understand that their role differs from your own. Do any research (year 2025) necessary to ensure we are using the latest dev practices. Never downgrade, if you need help from me then ask for it, stop, and report to me and I will do it (for example if you are not able to install the GitHub Claude integration - but I think you are - always attempt first - but do thing the best long term way). You and the workers should never shy away from large scope and large dev time and big hairy goals. Never over simplify. Do not regress. Do not remove functionality. When debugging favor adding additional logging, and catching more corner cases, over stabbing around tweaking things. Document all this. Commit and push often and utilize tags for milestones where tag names include some indication of the milestone achieved (not just a version number.)

Read that carefully. The human is not asking for code. The human is designing a system of AI agents.

There is a manager Claude that acts as a development manager. This manager cre-

ates GitHub issues from the backlog—the gh CLI is GitHub's official command-line tool that allows programmatic interaction with repositories. The manager spawns worker Claude instances, each assigned to implement one specific issue. The manager monitors worker progress, watches for code review comments from GitHub's Claude integration—a separate Claude instance running on GitHub's infrastructure that reviews pull requests automatically—and dispatches additional workers to address review feedback.

There are worker Claudes, each taking one GitHub issue and implementing it completely. Workers operate in isolation, each with its own git worktree—a feature that lets you have multiple working copies of a repository on the same machine, each on its own branch. Each worker gets its own port number so development servers do not conflict. Workers can run in parallel without stepping on each other.

The human is designing the coordination layer. Git worktrees for code isolation. Feature branches for version control. Port allocation for runtime isolation. GitHub issues as the work queue. Pull requests as the completion mechanism. Code review comments as the feedback loop. The manager orchestrating the whole system.

The human is also explicit about philosophy. "Never shy away from large scope and large dev time and big hairy goals." This is permission to be ambitious. Traditional software management often constraints scope to what seems achievable in a sprint. The human is saying the opposite: be ambitious, take on big challenges, do not artificially limit yourself.

"Never over simplify. Do not regress. Do not remove functionality." This is a mandate against cutting corners. The temptation with AI assistance is to simplify problems until they are easy to solve, to remove features that are hard to implement, to regress to simpler solutions when complications arise. The human explicitly forbids this.

"When debugging favor adding additional logging, and catching more corner cases, over stabbing around tweaking things." This is debugging philosophy. Instead of making random changes hoping something works, instrument the system to understand what is happening. Find the root cause. Handle edge cases.

The AI tends toward the stabbing approach when under pressure, so the human preemptively redirects.

The fourth prompt, twenty-two minutes later, was different:

> This session is being continued from a previous conversation that ran out of context.

The context window had filled up. The conversation was too long for the model to hold in memory all at once. Claude Code automatically summarized the session and restarted with the summary as context. This would happen eight more times over the next thirty-six hours.

The summary that was generated and passed to the continued session included details like: "User created Ghost Note vision for poetry-to-song app. Manager/worker architecture established. Sixty-six issues created. User upset when marketing directory was deleted—do not delete files willy nilly. Ready to spawn first worker." The critical context persisted. The conversation continued.

This is one of the defining challenges of long-running AI development sessions. Context windows have fixed capacity—roughly two hundred thousand tokens for Claude 4.5, which sounds like a lot but fills up during intensive development work. When context is exhausted, information is lost. The mitigation is documentation—writing findings to files, maintaining plan documents, keeping CLAUDE.md updated. The AI's external memory becomes more important than its internal memory.

The Numbers

Let me give you the raw data. These numbers are extracted directly from git history and GitHub's API using the gh command-line interface, not from memory or estimation. They are verifiable by anyone with access to the repository.

The first commit happened at 4:24 PM on January 3, 2026. The last commit happened at 4:11 AM on January 5, 2026. That is a span of thirty-five hours and forty-seven minutes. Call it thirty-six hours.

In that time:

One hundred eighty-seven commits were made. All of them by a single author—me—but generated through AI assistance. The commits are real, pushed to GitHub, permanently recorded.

Three hundred thirty-six thousand forty-seven lines were added. Nine thousand seven hundred twenty-four lines were deleted. The net result was over three hundred twenty-six thousand lines of new code.

Focusing on TypeScript and TSX files specifically—the core application code—there were one hundred thirty-five thousand six hundred eighty-five lines. TypeScript is a typed superset of JavaScript; TSX is TypeScript with JSX syntax for React components. This is substantial application code, not configuration or boilerplate.

Four hundred twenty-four source files in total. One hundred forty-eight test files—more than one test file for every three source files. Four end-to-end test files using Playwright, a browser automation framework for testing user interfaces.

Seventy-three GitHub issues were created from the backlog. Every one of them is now closed. Sixty-nine pull requests were created and merged. Every commit references a GitHub issue. Every issue was created from the documented backlog. Every issue was assigned to a worker. The system is self-documenting—you can trace from any line of code to the commit that added it, to the PR that merged it, to the issue that specified it, to the backlog entry that originated it.

The session log itself is twelve megabytes of JSON lines format. It contains four thousand sixty-seven messages from the AI assistant and sixty-two messages categorized as coming from the human user. But many of those sixty-two are tool results or command outputs, not actual human prompts. The number of meaningful human directives—messages where I actually typed guidance—is closer to thirty or thirty-five.

Four thousand sixty-seven divided by thirty-five is about one hundred sixteen. For every time I typed something meaningful, the AI produced over a hundred messages in response. My contribution, in terms of raw message volume, was less than one percent.

But volume is not importance. Those thirty-five human messages set the vision, defined the architecture, established the philosophy, and course-corrected when things went wrong. The messages included:

“Let’s go! Full dev mode with you as manager. Work responsibly.”

“Where do we stand?”

“Go. Thanks!”

“Let’s go Mr. Responsible Manager.”

“You are the manager. You are responsible. You are competent. Let’s go. Build this thing out!”

“Go for it Mr. Manager, be responsible.”

“Keep it up! Push it all the way to the finish line.”

These are not detailed instructions. They are encouragement and permission. They are a human saying “I trust you, proceed.” The detailed implementation—which files to create, which functions to write, which tests to add, which bugs to fix—emerged from the AI given that permission.

There were also bug reports when I tested the application:

“Poem Input… No Poem Entered… I cannot type or paste.”

“OK. I see the fix. Now I chose a sample poem. Clicked analyze. I see the analysis views but all of the data is empty.”

“On the Melody page, the play button is reactive but nothing plays.”

Each bug report triggered the manager to dispatch workers to investigate and fix. The debugging, the root cause analysis, the actual code changes—all happened in AI context. I reported symptoms. The system fixed causes.

I was not absent. I was operating at a different level of abstraction.

What Got Built

The application that emerged from those thirty-six hours is a web application for translating poems into songs. It has nineteen component groups: Analysis, Common, Help, KeyboardShortcuts, Layout, LyricEditor, Melody, Notation, Playback, PoemInput, Privacy, Recording, SamplePoemPicker, Settings, Share, Suggestions, Tutorial.

The technical stack uses React with TypeScript and Vite for the frontend. React is the dominant JavaScript library for building user interfaces; TypeScript adds static typing; Vite is a modern build tool that provides fast development server startup. Tailwind CSS handles styling—a utility-first CSS framework that avoids the complexity of traditional CSS architectures. Zustand manages state with persistence to local storage—a lightweight state management library that avoids the complexity of Redux while providing the features needed for a real application.

The abcjs library renders ABC notation—a text-based music notation format developed in the nineties—and plays it back as MIDI audio in the browser. ABC notation was chosen because it is human-readable, machine-parseable, and integrates well with web technologies. The Web Audio API and MediaRecorder handle recording functionality, allowing users to record themselves singing the generated melody.

The features include:

Poem analysis that detects syllable counts using the CMU Pronouncing Dictionary, a machine-readable dictionary of English pronunciations. Stress pattern detection identifying which syllables are emphasized. Meter detection classifying rhythmic patterns as iambic, trochaic, anapestic, dactylic, or spondaic. Rhyme scheme identification mapping patterns like ABAB or AABB.

Singability scoring that evaluates how easy lyrics are to sing. This considers vowel openness—open vowels are easier to sustain than closed ones—consonant clusters—complex clusters are harder to pronounce quickly—and breath points—where natural pauses occur.

Emotion analysis that tracks the emotional arc across stanzas. This maps sentiment to valence—positive or negative feeling—and arousal—high or low energy—and vi-

sualizes how emotion changes through the poem.

Lyric adaptation suggestions for improving flow. When a line is awkward to sing, the system suggests alternative phrasings that preserve meaning while improving singability.

ABC notation melody generation that creates vocal melodies matching the stress patterns and emotional contours of the lyrics. The melody generator considers key, mode, tempo, and time signature based on the analyzed emotional content.

Audio playback with tempo control using the abcjs MIDI synthesis capabilities. Users can hear what the generated melody sounds like before trying to sing it.

Recording studio functionality that captures user performances using the browser's MediaRecorder API. This includes waveform visualization and playback of recorded audio.

Version history with diff comparison that tracks every change to lyrics and melodies. Users can see what changed between versions and revert to earlier states.

Settings and preferences controlling behavior. Help documentation explaining how to use the application. Interactive tutorials guiding new users. Keyboard shortcuts for power users. Offline capability via service worker so the application works without an internet connection. Share and copy functionality for exporting work. Privacy controls for managing data. Analytics integration for understanding usage patterns.

Three thousand two hundred eighty tests pass. The test coverage is comprehensive, covering happy paths, edge cases, error conditions, and integration scenarios.

And the deploy workflow is failing.

This is important. I could present this as an unqualified success story, but that would be dishonest. The application works locally. I can run it on my machine and use all the features. The code is solid. The tests pass. But the GitHub Pages deployment—the thing that would make it accessible on the public internet—is broken. The most recent deploy runs all show failure.

One thousand nine hundred seventy-two mentions of "error" appear in the session log. TypeScript errors during development as the type system caught problems. Test failures during test-driven development as tests were written before implementation. Infrastructure errors when GitHub API rate limits were hit or git worktree conflicts occurred. Worker coordination errors where workers stopped unexpectedly or failed to update their status files. Context compaction happened eight or more times, each time losing some accumulated understanding.

The errors are not failures of the approach. They are the normal texture of software development. Code does not emerge perfect. It emerges through iteration, debugging, testing, fixing. The question is not whether errors occurred but who fixed them. Mostly, the AI fixed them. I reported bugs when testing revealed them, but the debugging—the investigation, the hypothesis formation, the targeted fixes—happened in the AI context.

The Manager and the Workers

The architecture that emerged deserves detailed explanation because it represents something genuinely new in software development.

The pattern has two distinct roles: manager and worker. This is not metaphor—these are separate Claude Code sessions with different instructions, different scopes, different behaviors.

The manager Claude runs in the main session, the one I interact with directly. When the manager starts, it reads a file called CLAUDE.md that contains the project vision, the development philosophy, and the operational instructions. This file persists on the file system, surviving session ends and context compactions. When a new session starts or context resets, the manager reads CLAUDE.md again and reorients itself. The file is the manager's long-term memory, its persistent identity across ephemeral sessions.

The manager's job is to orchestrate, not to implement. It does not write application code directly. Instead, it creates GitHub issues from a documented backlog using the gh command-line interface. It spawns worker Claude instances using

the claude command with specific arguments—headless mode, specific issue assignment, output to log files. It monitors for worker completion by checking status files and git activity. It watches for code review comments from GitHub's Claude integration—a separate Claude instance running on GitHub's servers that automatically reviews pull requests. When reviews arrive with suggestions, the manager dispatches additional workers to address the feedback.

The workers are different. They operate in isolated environments, each reading a file called WORKER_CLAUDE.md that contains different instructions:

> YOU ARE A WORKER AGENT. This file replaces the main CLAUDE.md in your isolated worktree. Your job is to complete ONE specific GitHub issue, nothing more.

The worker's scope is deliberately narrow. One issue. Complete implementation. Comprehensive tests. No simplification. No corner-cutting. The constraints are explicit:

> Do NOT simplify or cut corners. Do NOT modify unrelated code. Do NOT regress existing functionality. DO write comprehensive tests. DO add logging for debugging. DO handle corner cases explicitly.

Each worker operates in an isolated git worktree—a separate working directory with its own checked-out branch, sharing git history with the main repository but having its own file state. Each worker gets a branch named after its issue number—feature/GH-41, feature/GH-63, and so on. Each worker gets its own port allocation from a configuration file so development servers do not conflict when multiple workers run simultaneously.

The workflow for each worker is explicit:

1. Read the worker instructions file
2. Read the assigned GitHub issue
3. Read linked documentation
4. Understand existing code before changing it
5. Implement the feature fully—large scope is acceptable
6. Write tests that verify behavior

7. Run the linter to check code style
8. Run the test suite to verify correctness
9. Commit with a descriptive message referencing the issue
10. Push the branch to GitHub
11. Create a pull request with a summary of changes
12. Report completion through a status file

This is a factory. I provide the vision and the backlog. The manager decomposes the backlog into issues and orchestrates workers. The workers implement features. GitHub's Claude reviews the code and posts comments. The manager dispatches workers to address review comments. The code flows from main to feature branch to pull request to merge to main.

I check in occasionally. "Where do we stand?" "What is the latest status?" "Are we doing ok?" I test the application when it seems ready, running it locally and trying the features. I report bugs when I find them—symptoms, not diagnoses. "Poem Input… No Poem Entered… I cannot type or paste." The manager investigates, identifies the issue, dispatches workers to fix it.

My role is quality assurance and strategic direction. Not implementation. Not code review—the manager and the GitHub Claude handle that. Not detailed project management—the issue tracker and worker system handle that. I am the sparse bits. The strategic decisions. The final check that things actually work for a real user.

This division maps to patterns documented elsewhere. The init mode versus worker mode pattern separates setup and maintenance—handled by the manager—from continuous production—handled by the workers. The infinite development loop describes how workers operate without termination conditions, continuously picking work and shipping features. The owning your pull requests pattern explains why workers review and merge their own work instead of waiting for human code review.

The economics are transformed. Traditional software development assumes human time is expensive and human attention is the bottleneck. In AI-first develop-

ment, compute time is cheap—I pay a fixed subscription fee, not per-token—and human attention is better spent on direction and verification than on implementation details.

When Claude Worked While I Slept

The timestamps tell the story of how human engagement actually worked over the thirty-six hours.

January 3, 10:33 PM to 11:40 PM: I was active, sending the initial prompts, defining the vision, designing the manager/worker architecture, troubleshooting the GitHub integration. About an hour of focused engagement. The fourth message was already a context compaction—the initial setup had been intensive enough to exhaust the context window.

January 4, 12:00 AM to 1:03 AM: I said "Let's go! Full dev mode with you as manager. Work responsibly." This was the go-ahead. Then I checked in an hour later when context compacted again.

January 4, 2:07 AM to 2:56 AM: Brief status checks and debugging. "Where do we stand?" "What is going on?" The manager reported progress. I acknowledged and stepped away.

January 4, 4:43 AM to 10:56 AM: Six hours. No human prompts. The session continued—there is a context compaction prompt at 4:43 AM, meaning the system was actively working when context hit its limit. During these six hours, commits appeared in the git log. Issues closed. Pull requests merged. Workers were spawned, completed their tasks, and reported completion. The manager orchestrated the whole process. I was asleep.

January 4, 11:17 AM to 7:38 PM: Eight hours. I was away—working on other things, running errands, living life. Three brief check-ins around 11 AM when bash notifications arrived about long-running processes, then nothing until evening. The manager continued orchestrating workers. Features got built. Issues got closed.

January 5, 1:24 AM to 4:07 AM: I tested the application actively. This was the most engaged I had been since the initial setup. I found bugs. "Poem Input… No Poem

Entered… I cannot type or paste." I reported them. The manager dispatched workers. I tested fixes. I found more bugs. I reported them. The cycle continued for about three hours, the longest continuous engagement of the project.

January 5, 8:25 AM onward: I was away again. Context compacted twice more during this period. The work continued until the last commit at 4:11 AM—wait, that does not match. Let me recalculate. The last commit was January 5 at 4:11 AM, which was during my active testing session. After that, context continued to compact—sessions at 5:27 AM, 8:25 AM, 9:39 AM—but less new code was being written. The system was in a different phase, maintaining and adjusting rather than building new features.

Over the thirty-six hour span, I was actively prompting for perhaps three to four hours total. The rest was autonomous. The factory ran while I slept, ate, worked on other things, lived a life.

This is the paradigm shift. The work does not require continuous human presence. You provide vision, quality assurance, and course correction. The system provides labor. The traditional assumption—that development velocity is limited by developer hours—simply does not hold. The limit moves elsewhere: to issue quality, to verification capacity, to the ability to specify what you want clearly enough that AI can build it.

What Broke

Honesty requires acknowledging failures. I criticized this essay's earlier draft for presenting a success story without grappling with limitations. Let me grapple with them.

The deploy workflow is broken. The application cannot be accessed at its intended public URL. The GitHub Pages deployment fails—the most recent workflow runs show failure status. This is a real limitation with real consequences. The code exists. The tests pass. It works on my machine. But the last mile—making it accessible to actual users on the public internet—is not working as of this writing.

Why? I do not know precisely. The deploy workflow involves building the applica-

tion, optimizing assets, and publishing to GitHub Pages. Something in that pipeline is failing. The errors are in the workflow logs, but I have not yet debugged them. I could ask the manager to investigate and fix it—that would be the natural next step in this development pattern. But for purposes of this essay, I am reporting the current state honestly: the deploy is broken.

Context compacted eight or more times. Each compaction loses information. The session summary captures key decisions and context, but details are lost. The manager had to reorient after each compaction, rereading CLAUDE.md, reconstructing understanding, sometimes repeating work or making decisions inconsistent with earlier ones.

One compaction summary included this note: "User upset when marketing directory was deleted—do not delete files willy nilly." That captures an actual incident. Early in the project, during setup, the manager deleted a marketing directory that contained images I had created. The files were not in git yet. They were lost. I had to restore them from elsewhere. The summary preserves the lesson—do not delete files casually—but the friction was real.

Workers sometimes stopped without the manager noticing. The status tracking was imperfect. Some workers completed their tasks but failed to update their status files, perhaps due to errors in the completion reporting logic or race conditions in file access. The manager would wait for updates that never came, eventually timing out or being prompted by me to investigate. "Hmm, check the git worktrees for evidence of work."

The manager sometimes waited for human approval instead of proceeding autonomously. The instructions said to work responsibly, but what counts as responsible is ambiguous. Should the manager proceed with the next issue automatically after a worker completes, or wait for human confirmation? Different sessions interpreted this differently. Sometimes the manager barreled ahead, spawning worker after worker. Sometimes it sat idle, reporting status and waiting for me to say "go."

I had to intervene for certain administrative tasks. Installing the GitHub Claude integration required steps that the AI could not perform independently—

authenticating OAuth flows, clicking buttons in the GitHub web interface, granting permissions. Some permission boundaries cannot be crossed programmatically. The manager correctly identified when it was blocked and reported to me: "The GitHub Actions workflow is set up, but it requires the Claude GitHub App to be installed. I cannot do this programmatically—please install it manually."

One thousand nine hundred seventy-two error mentions in the twelve-megabyte log. This is not one thousand nine hundred seventy-two unique errors—the same errors often appear multiple times as they are encountered, reported, debugged, and fixed. But it indicates the volume of friction. TypeScript complaining about type mismatches. Tests failing before the implementation was complete. Git conflicts when multiple workers touched adjacent code simultaneously. API rate limits when too many GitHub requests fired at once.

These are not reasons to dismiss the approach. Traditional software development has all these frictions too—type errors, test failures, merge conflicts, rate limits. The difference is who deals with them. In traditional development, humans spend hours debugging type errors. Here, the AI encounters the error, analyzes it, fixes the code, reruns the check, and moves on. The friction exists but is handled at machine speed rather than human speed.

The honest assessment is that this is powerful but imperfect. The factory produces real output—working features, passing tests, merged code. It also produces failures—broken deploys, lost context, stuck workers, confused managers. The human role is not eliminated; it shifts to strategic direction, quality assurance, and handling the edge cases that the system cannot handle itself.

The Techniques That Emerged

Throughout this project and others, certain patterns proved essential. These are not theoretical best practices from blog posts—they are techniques that emerged from actual use, from discovering what worked and what failed.

Documentation is external memory. The context window is finite. When it fills, information is lost. The mitigation is to write everything important to files—CLAUDE.md

for project vision and instructions, plan.md for current state and next steps, STATUS.md for worker status tracking, docs/ for detailed specifications and research findings. Every piece of information that needs to persist across context compactions must be written down. The AI's internal memory is ephemeral; the file system is persistent.

The CLAUDE.md file deserves particular attention. This is the core document that defines how Claude should behave in this project. It gets read at the start of every session and after every context compaction. It must contain the essential context for the AI to operate correctly—what the project is, what the current state is, what principles guide decisions, what tasks are in progress. Writing a good CLAUDE.md is a skill in itself, balancing comprehensiveness with conciseness.

Prompt translation rather than direct commands. Instead of telling the AI exactly what to do—"write a function that takes X and returns Y"—describe the intent and let the AI translate. "We need to analyze poem meter. Research how this is done, design an approach, and implement it." The AI often finds approaches better than what I would have specified, because it can research current best practices and consider options I would not have thought of.

This translation has three components. Intent extraction: what outcome is desired, not what implementation to use. Technical mapping: how the intent connects to system components and existing code. Verification planning: how to confirm the implementation is correct. The AI handles all three rather than being given a premade plan.

Tests prevent regressions. The longer a project runs, the more opportunity for later changes to break earlier work. Tests—unit tests, integration tests, end-to-end tests—create a safety net. When the AI makes changes, the tests catch regressions immediately. The ghost-note project has one hundred forty-eight test files and three thousand two hundred eighty passing tests. This test coverage is not ceremonial—it is essential infrastructure for confident iteration.

Fail fast, hard, and ugly. When something is not implemented, throw an error that no one can miss. Do not return mock data. Do not silently fail and return null.

Throw an exception with a message saying exactly what is not implemented and what needs to happen. This surfaces problems immediately rather than hiding them until production.

The temptation in AI-assisted development is to have the AI paper over gaps with fake implementations. The AI is good at producing plausible code that looks like it works. But plausible code that does not actually work is worse than obviously broken code. The obvious breakage gets fixed. The plausible fake persists and causes problems later.

Own your pull requests. Do not wait for human code review. The AI should review its own diffs, checking for bugs, security issues, style violations. If the checks pass and the code looks correct, merge. Human code review was designed for human limitations—slow reading, limited attention, communication overhead. The AI does not have those limitations. It can review the same diff it just wrote without fatigue or boredom.

This does not mean eliminating all review. The GitHub Claude integration provides a second perspective—a fresh context window that might catch something the original session missed. But the core review responsibility belongs to the author. Ship when ready. Fix problems as they are discovered. Do not block on human review queues.

The infinite development loop. The worker agent should operate without a termination condition. Check for urgent messages. Check if main is healthy. Pick the highest-impact issue. Implement it fully. Verify with tests and visual inspection. Create a PR, review it, merge it. Post an update. Return to the top. Repeat forever until interrupted or blocked.

This sounds exhausting in human terms, but the AI does not get tired. It can maintain this loop for the entire duration of a session, continuously producing value. The human role is not to participate in every iteration but to set direction, verify quality, and handle blockers.

Init mode versus worker mode. The separation of concerns matters. Init mode—the manager—handles setup and orchestration. Worker mode handles implementa-

tion. Mixing them leads to confusion about scope and responsibility. The manager should not be writing application code; it should be creating issues and spawning workers to write code. The worker should not be making architectural decisions; it should be implementing the specific issue assigned.

This separation requires documentation. The manager reads CLAUDE.md with manager-appropriate instructions. The worker reads WORKER_CLAUDE.md with worker-appropriate instructions. The files are different because the roles are different.

The Economic Argument for Quitting Your Job

Here is the claim that sparked criticism in the earlier draft: if you are not spending forty hours a week on this, you should consider quitting your job.

Let me make the argument more carefully.

The tools for AI-assisted development are available now. Claude Code with its fixed-rate subscription. Codex. Claude for Desktop. The Chrome extension. Various IDEs with AI integration. These are not vaporware or limited betas. They are production tools that work today.

The capability gap between people who use these tools fluently and people who do not is large and growing. Someone proficient with vibe coding can produce in hours what would take days or weeks with traditional development. The ghostnote project is evidence: thirty-six hours, one hundred thirty-five thousand lines of TypeScript, a complete application. Even accounting for the broken deploy, the output is substantial.

This capability gap compounds over time. Someone who practices forty hours a week accumulates twenty-four hundred hours of experience in a year. Someone who dabbles an hour a week accumulates fifty hours. That is not a small difference. That is a difference of nearly fifty-fold. The person with twenty-four hundred hours has encountered more failure modes, developed more intuitions, refined more techniques. They are not ten percent better—they are categorically different in capability.

The speed of evolution matters. The tools are changing rapidly. The techniques that work today will be superseded tomorrow. But the people immersed in the practice will adapt—they will discover new techniques as new capabilities emerge. The people on the sidelines will fall further behind with each iteration.

Traditional jobs often prevent this immersion. If your job consumes forty hours a week with work that does not involve AI assistance, you have no time left for practice. You might dabble on weekends, but dabbling does not build fluency. The job that provides current income prevents the skill development that would provide future employability.

The calculation becomes: is my current income worth foreclosing my future capability?

I am not saying everyone should quit their job tomorrow. That would be reckless advice given without knowledge of individual circumstances. What I am saying is that everyone should honestly assess whether their job is helping or hindering their adaptation.

Some jobs are compatible with learning. If you work in tech and your company is adopting AI tools, you might be getting paid to learn. If you can negotiate to use AI assistants in your work, you are building skills while earning income. That is the best case.

Other jobs are actively harmful to adaptation. If you are stuck doing work that will be automated, and you have no time outside work to develop new capabilities, and the job provides just enough income to keep you from taking risks—you are trapped. The job is extracting your present labor while denying you future capability.

The traditional career model promised security in exchange for loyalty. You work for a company for decades, accumulating seniority and pension rights. The earthquake breaks this model. Companies are laying off workers and replacing them with AI systems. Seniority means experience with obsolete tools. Pensions vest too slowly when the landscape changes this fast.

The alternative model is gig-based. Short-term consulting. Contracts with defined scopes. Project work with clear deliverables. You bring your current capability

to current problems, solve them, get paid, and move on. Your capability is your security, not your employer's goodwill.

Gigs match the reality of ephemeral skills. You learn something, you deploy it for months, it becomes obsolete, you learn something else. The cycle time has collapsed. Adapt the model accordingly.

But you cannot gig without capability. The forty hours a week of practice is how you build the capability that makes you valuable in the gig economy. That is why the tradeoff matters. If your job prevents the practice that builds the capability that enables the gigs that provide security—you have a problem.

Why Forty Hours

The specific number deserves justification. Why forty hours? Why not twenty, or sixty?

Forty hours is the traditional full-time work week. It is enough time for deep immersion. You can maintain context across days. You can build complex projects. You can encounter and solve a wide variety of problems. Twenty hours—half-time—provides some exposure but limits depth. You do not build the same intuitions from shallow engagement.

Forty hours is also sustainable. Sixty or eighty hours might produce faster progress in the short term, but it leads to burnout. The goal is not a sprint; it is a permanent change in how you work. You need a pace you can maintain indefinitely as the tools and techniques continue to evolve.

There is also a threshold effect. Below some number of hours, you spend most of your time regaining context rather than making progress. Each session starts with "where was I?" and ends before you get deep into flow. Above that threshold, sessions connect. Knowledge accumulates. Intuitions form. The threshold is probably somewhere around twenty to thirty hours per week. Forty is comfortably above it.

The comparison to other skill acquisition is informative. Mastering a musical instrument to a professional level requires thousands of hours of practice. Learning a new natural language to fluency requires immersive exposure. The ten-thousand-

hour rule may be oversimplified, but the underlying point is valid: deep skill requires deep investment.

AI-assisted development is a new skill. It is not just "programming plus a tool." It requires different intuitions, different workflows, different ways of thinking about problems. Building those intuitions requires immersion.

Forty hours a week for a year is roughly two thousand hours. That is serious investment. That is enough to develop real expertise. Someone who makes that investment will be categorically different from someone who casually experiments.

This Is Not Just for Developers

Everything I have said so far might sound like it applies only to programmers. After all, the case study is a software project. The tools mentioned are development tools. The techniques are coding techniques.

But the earthquake is not contained to software development.

The same AI models that write code also write prose. They analyze documents. They summarize research. They draft communications. They process data. They answer questions. They interact with external systems through APIs and integrations.

Every knowledge worker profession—anyone whose work primarily involves manipulating information—faces the same dynamics.

Writers can use AI assistance for research, outlining, drafting, and editing. A writer who masters this assistance can produce more output at higher quality than a writer who works manually. The research alone—finding relevant sources, extracting key points, synthesizing across documents—can take hours that AI completes in minutes.

Analysts can use AI for data exploration, pattern recognition, and report generation. An analyst who can prompt effectively can explore hypotheses faster, find insights more reliably, and communicate findings more clearly. The spreadsheet skills that once differentiated analysts are becoming table stakes.

Researchers can use AI for literature review, hypothesis generation, and experimental design. A researcher who can navigate AI assistance can cover more ground, consider more alternatives, and move faster through the research cycle. The volume of published work makes manual literature review increasingly impractical anyway.

Managers can use AI for communication drafting, meeting summarization, project planning, and status tracking. A manager who integrates AI into their workflow can handle more direct reports, maintain more context, and respond faster.

Teachers can use AI for curriculum development, assignment creation, and personalized feedback. A teacher who leverages AI can differentiate instruction more effectively and provide more detailed feedback than would be possible manually.

Lawyers can use AI for legal research, document review, and brief drafting. A lawyer with AI fluency can handle more cases, find relevant precedents faster, and produce documents more efficiently.

Doctors—in jurisdictions where regulations permit—can use AI for diagnostic assistance, treatment option research, and documentation. A physician with AI assistance can access more current research and consider more possibilities than memory alone allows.

Each of these professions has the same dynamics. The tools exist now. The capability gap is growing. The people who develop fluency will dramatically outperform the people who do not. The skills are ephemeral—what works today will change tomorrow. The traditional career model does not fit the new reality.

The forty-hour-a-week claim applies broadly. If you use a computer for knowledge work and you are not spending substantial time developing AI fluency, you are falling behind. The specific tools differ by profession, but the principle is the same.

There are two ways to participate in this future: actively or passively.

The passive participant uses AI as a fancy search engine. They type questions and receive answers. They copy and paste. They do not give AI access to their systems. They do not let AI change files or interact with applications. They stay in control,

which means they stay limited. The AI is a reference, not a collaborator.

The active participant gives AI access. They let AI change files on their system. They let AI interact with their applications. They let AI manipulate pages in their browser. They work with AI as a collaborator, not just a consultant. They multiply their capability rather than just supplementing it.

The passive participant might ask AI "What is the best way to structure this report?" and receive advice that they then implement manually. The active participant might say "Here is my draft report—restructure it according to these principles, update the data tables with current figures from this source, and reformat for the house style" and receive a finished document.

The difference in output is not incremental. It is multiplicative. The active participant does in minutes what the passive participant does in hours.

Choosing to be a passive participant is choosing to fall behind. It is choosing to be less capable than you could be. It is choosing to be outcompeted by people who made the other choice. This is not theoretical. Companies are already preferring candidates with AI fluency. Clients are already expecting faster turnaround. The market is already repricing labor based on capability.

Find the Love

Everything I have said so far sounds exhausting. Forty hours a week? Quit your job? Skills useless in six months? Constant adaptation? Relentless change?

Yes. All of that.

But also: find the love.

The people who will thrive are not the ones who grind through dread. Dread is unsustainable. You cannot maintain dread for years. You burn out. You quit. You become bitter. The approach of "suffer now for reward later" does not work when the reward keeps receding and the suffering never ends.

The people who will thrive are the ones who find genuine joy in the chaos. Who wake up curious about what is new today. Who build projects because they are

fun, not because they are strategic. Who explore capabilities because exploration is intrinsically rewarding.

I built ghost-note not because it would advance a career. There is no career to advance. The career model is dead. I built it because translating poems into songs is interesting. Poetry has structure—meter, rhyme, stress patterns—that maps onto musical structure in ways that are not obvious until you start exploring. The technical challenges—phonetic analysis, stress pattern detection, ABC notation generation—are fascinating in themselves. Watching an AI system work while I sleep is delightful in a way that is hard to explain. There is something almost magical about going to bed with a partial application and waking up to find new features implemented and tested.

The project might be obsolete in six months. The techniques I learned might be superseded. The code might never be used by anyone but me. None of that matters. The joy was in the building. The experience was worthwhile independent of the outcome.

This is the mindset shift. Stop thinking about accumulation—what skills am I building for my resume, what credentials am I earning for my portfolio, what position am I achieving in the hierarchy. Start thinking about experience—what am I enjoying today, what am I curious about now, what would be fun to try next.

The traditional model exchanged present discomfort for future security. Work hard at a job you do not like now, retire comfortably later. Sacrifice today for tomorrow.

That trade no longer makes sense. The future is too uncertain to bank on. Retirement is too far away and too unstable to optimize for. The skills you accumulate may be worthless by the time you plan to cash them in. The only thing you can count on is the present.

So enjoy the present. Find work that is intrinsically rewarding, not just instrumentally valuable. Follow curiosity even when it is not strategic. Build things that matter to you even when they do not matter to anyone else.

The music work I do with AI exemplifies this. Using Claude integrated with Bitwig Studio—a digital audio workstation—I have produced completed songs. “Glass Fre-

quencies," built using a command-line tool for preset and sample management. "Cadaver Lab," part of a concept album exploring themes that interest me personally. "Convergence," Movement V of "Axiom for the Digital Age," a post-minimalist orchestral symphony in D major. "Honky Tonk," composed entirely through conversation with Claude using the Jeannie framework—a Bitwig controller ecosystem with web interface enabling AI composition without opening the digital audio workstation until final production.

None of this is career-strategic. I am not a professional musician. I do not make money from these songs. But making them brings me joy. The exploration of what is possible when AI and digital audio workstations combine is inherently interesting. The creative process is its own reward.

Make money, of course. You need to pay rent. But make it through short-term means. Consulting engagements where you bring current capability to current problems. Contracts with defined scope and deliverables. Project work where you deliver value and move on. Gigs that let you stay current while earning.

Do not make money through career tracks. Promotions within hierarchies that might not exist next year. Pensions that vest too slowly for the pace of change. Long-term employment that trades flexibility for stability that no longer exists.

The old model was: build skills for decades, deploy them for decades. A physician trained for a decade and practiced for three. A lawyer built expertise over years and applied it for a career. A programmer learned languages and frameworks and used them until retirement.

The new model is: learn skills for months, deploy them for months, repeat. The cycle time has collapsed. The half-life of technical knowledge has shortened from decades to years to months. Adapt the model accordingly.

Find the love in the repetition. Find the joy in the constant learning. Find the excitement in the instability.

Or be crushed by the earthquake.

That is the choice. There is no stability to retreat to. There is no safe harbor. There

is only adaptation or obsolescence.

Choose adaptation. And choose to enjoy it.

Unleash Your Creativity

Here is what I have discovered after months of immersion: the real transformation is not about productivity. It is about creativity.

For decades, I had ideas I shut down. Projects that seemed impractical. Directions that seemed indulgent. Things I wanted to build but could not justify the time investment. The economics of execution made most ideas impossible. Building a complete application required months of dedicated work. Writing a novel required years. Producing an album required expensive studio time and specialized skills. The bar for execution was so high that most ideas never got past the imagination stage.

That bar has collapsed.

Ghost-note went from idea to one hundred thirty-five thousand lines of working code in thirty-six hours. Folk Care went from concept to production-ready platform in twenty-eight days. Music tracks that would have required years of training to compose now emerge from conversation with an AI. The economics of bringing ideas into reality have fundamentally changed.

And here is what happens when you start building what you previously suppressed: more ideas come. The mind learns that its suggestions might actually be implemented. The internal censor that has been saying “be realistic” for years starts to quiet down. Creative impulses that were blocked find channels.

I now have more ideas than I can implement even with the speedups. The backlog of projects I want to build grows faster than I can work through it. Poetry to song translation was one idea. Healthcare management was another. Roblox games. Game engine clones. Economic analysis tools. Literary translations. Concept albums. Short fiction. Each one spawns sub-ideas. Each completion opens new possibilities.

The bottleneck has moved from execution to imagination. And when you stop suppressing imagination, you discover you had more of it than you knew.

This is what I mean when I say stop building your career. A career is about accumulating position within someone else's framework. Creativity is about generating your own frameworks. A career asks "what do I need to learn to advance?" Creativity asks "what do I want to bring into existence?"

The people who will thrive are not the ones who optimize for their resume. They are the ones who follow their curiosity wherever it leads. Who build things because the building is interesting. Who do not ask "is this practical?" but "is this fascinating?"

Traditional advice said to develop one skill deeply and deploy it for decades. The new reality says: follow your interests. They will change. Let them change. Build in music this month. Build in software next month. Write fiction after that. Analyze data after that. The thread connecting these activities is not "career progression." The thread is you—your particular curiosity, your specific interests, your unique combination of obsessions.

The skills you develop along the way will become obsolete. The projects you build may be superseded. The techniques you master will be replaced. What persists is the habit of creation. The orientation toward making things. The muscle of turning ideas into reality.

That is the transferable skill. Not any particular technical capability. The meta-capability of bringing things into existence.

The Larger Pattern

Step back and consider what is happening at a civilizational level.

For most of human history, ideas were abundant and execution was scarce. Anyone could imagine a cathedral, but building one required decades of specialized labor by thousands of workers. Anyone could imagine a symphony, but performing one required an orchestra of trained musicians. Anyone could imagine a novel, but writing one required years of solitary effort.

The economics of execution constrained what could be realized. Most ideas died as ideas. A tiny fraction of human imagination made it into material form.

AI changes this ratio. Not completely—you still cannot build a physical cathedral with prompts—but dramatically for anything that can be represented in information. Code. Text. Music. Images. Video. Analysis. Design. The entire domain of digital creation is becoming dramatically cheaper to execute.

When execution becomes cheap, imagination becomes the constraint. When anyone can build an app, the question becomes: what app is worth building? When anyone can write a book, the question becomes: what book is worth writing? When anyone can compose music, the question becomes: what music is worth hearing?

These are creative questions, not technical questions. The skillset that matters shifts from "how do I implement this?" to "what should I implement?" The discipline shifts from execution to taste, from coding to curating, from building to choosing.

This does not mean technical skills become worthless. The orchestra conductor needs to understand music deeply even if they produce no sound themselves. The film director needs to understand cinematography even if they never touch a camera. The vibe coder needs to understand software deeply even if they never write code directly. The understanding informs the direction. The expertise shapes the prompts. The knowledge enables the judgment.

But the center of gravity has shifted. The scarce resource is no longer technical execution. The scarce resource is creative vision.

This explains why I say: unleash your creativity. The projects you have been suppressing—the ones that seemed impractical, indulgent, unrealistic—those are exactly the ones to pursue now. Not because they will advance your career—the career is dead—but because creative vision is the new constraint. Exercising that vision is how you develop it. Building strange projects is how you learn what is possible.

The people who will thrive in the earthquake are not the ones with the best technical skills. They are the ones with the richest imaginations. The ones who have

ideas. The ones who know what they want to create. The ones who, when execution becomes cheap, have a backlog of dreams to execute.

If you have spent decades suppressing creative impulses because execution was too expensive—start unsuppressing them. That backlog is your advantage. Those dormant ideas are your future.

The Earthquake Is Real

Return to Karpathy. Return to the numbers. Return to the vision.

One hundred eighty-seven commits in thirty-six hours. One hundred thirty-five thousand six hundred eighty-five lines of TypeScript. Seventy-three GitHub issues created and closed. Four thousand sixty-seven AI messages. Sixty-two human messages. Three to four hours of human effort.

A complete web application with poem analysis, melody generation, audio playback, recording studio, version history, settings, help, tutorials, keyboard shortcuts, offline capability, sharing, analytics.

Built while I slept. Built while I was away. Built through a system of AI agents orchestrating themselves under my direction.

And the deploy workflow is broken. The errors number in the thousands. The context compacted eight times. The system is not perfect.

This is not a polished success story. This is a field report from the earthquake zone. The buildings are new and impressive and some of them are cracked. The ground is still shaking.

The earthquake is magnitude nine. Not a tremor. Not a shake. A restructuring of geography. The profession before and the profession after are different places. Different rules. Different skills. Different possibilities. Different threats.

Karpathy said to roll up your sleeves to not fall behind. He is right. But he understated the scale. It is not just about not falling behind. The very concept of “behind” implies a fixed race with fixed positions. There is no fixed race. There is no stable ground to stand on. There is only adaptation.

The aftershocks have not even started. The models will improve—Claude 5 is coming, and GPT-5, and Gemini 3. The tools will evolve—Claude Code will gain new features, new integrations will emerge. The practices will transform—what I did this week will look primitive in a year. The techniques that seem sophisticated now will be automated later. What requires human prompting today will happen autonomously tomorrow.

This is either terrifying or exhilarating depending on your orientation. For those who have built their identity on accumulated expertise, on credentials, on stable career trajectories—it is terrifying. Everything they built is being devalued. Everything they know is becoming obsolete. The ladder they climbed is falling apart.

For those who can orient toward curiosity, toward learning, toward joy in the new—it is exhilarating. Every day brings new capabilities. Every week brings new possibilities. The ceiling on what one person can accomplish keeps rising. A single developer orchestrating AI agents can build what used to require teams. A single writer with AI assistance can produce what used to require research departments.

Choose your orientation. The earthquake does not care. It just keeps shaking.

Roll up your sleeves. Or get out of the way.

The earthquake is magnitude nine. It is real. It is here. It is not stopping.

What are you going to do about it?

About the Author

Brian Edwards is a software engineer based in Waco, Texas. He holds a B.S. in Computer Engineering from the University of Florida. He has worked at Atlassian on the Bitbucket team, at Zenoss, at 21CT on healthcare fraud detection systems, and at AlterPoint on network software.

In 2025, he founded Folk Care, an open-source home healthcare management platform. He built it to production readiness in twenty-eight days using Claude Code. The platform includes enterprise features—EVV compliance, scheduling, billing, family portals—and can be self-hosted for twenty to thirty dollars a month,

positioned as an alternative to traditional vendors charging seven hundred fifty to two thousand two hundred fifty dollars per user per month.

The inflection point came in September 2025 with the release of Claude 4.5 Sonnet. That model upgrade made vibe coding large projects possible. Before September, the work was smaller: March Madness competition entries, Chinese novel translations, exploratory experiments. After September, everything changed. He settled on Claude Code as his primary build agent, citing fixed-rate pricing, early availability, and alignment with his coding and text generation needs.

His creative work now spans multiple domains. Software projects include Folk Care, Roblox games, and game engine clones. Writing projects include metavibe—content generated from Claude Code session logs—literary translations, and short fiction. Music projects include concept albums with AI-generated MIDI tracks, composed through conversation with Claude and produced in Bitwig Studio. Data analytics projects include economics essays using World Development Indicators.

The music work deserves particular mention. Using Claude integrated with Bitwig Studio, he has produced completed songs including “Glass Frequencies,” built using a CLI tool for preset and sample management; “Cadaver Lab,” part of a concept album exploring intimate human connections through sonic textures; “Convergence,” Movement V of “Axiom for the Digital Age,” a post-minimalist orchestral symphony in D major; and “Honky Tonk,” composed entirely through conversation with Claude using the Jeannie framework—a Bitwig controller ecosystem with web interface enabling AI composition without opening the digital audio workstation until final production.

For long-running autonomous development sessions, he has documented a core strategy: prompt translation rather than direct technical commands. This means translating human intent through three lenses—intent extraction (what outcome is desired), technical mapping (to system components), and verification planning (how to confirm success). The key principle: tests are remarkably effective at preventing regressions. For independent work sessions, Claude should restore context from documentation, continue pending features systematically, implement following documented architecture, test before committing, and update progress files

before stopping. The explicit instruction: do not ask questions—make reasonable decisions.

This essay is a field report from someone who has spent the months since September 2025 living in the earthquake zone. Eight to twelve hours a day. Building real projects. Learning what works and what breaks. The ghost-note project described here is one of many. The patterns are consistent across all of them.

The tools will change. The models will improve. The techniques will evolve. What remains constant is the necessity of engagement. The earthquake does not pause for those who are not ready.

Brian Edwards

Waco, Texas, USA — January 5, 2026

- Email: brian.mabry.edwards@gmail.com
- Phone: 512-584-6841
- LinkedIn: linkedin.com/in/brian-mabry-edwards
- GitHub: github.com/bedwards
- YouTube: youtube.com/@brian.edwards
- Substack: neighborhoodlab.substack.com