VIBE CODING

VIBE CODING

BUILDING PRODUCTION-GRADE SOFTWARE WITH GENAI, CHAT, AGENTS, AND BEYOND

GENE KIM & STEVE YEGGE

Foreword by Dario Amodei,
CEO and Cofounder of Anthropic

IT Revolution
Independent Publisher Since 2013
Portland, Oregon

25 NW 23rd Pl, Suite 6314
Portland, OR 97210

First Edition
Printed in the United States of America
30 29 28 27 26 25 1 2 3 4 5 6 7 8 9 10

Cover Design by Alana McCann
Book Design by Devon Smith

Library of Congress Control Number: 2025022944

Paperback: 9781966280026
Ebook: 9781966280033
Audio: 9781966280040

For information about special discounts for bulk purchases or for information on booking authors for an event, please visit our website at www.ITRevolution.com.

DEDICATION

From Gene: To the loves of my life: my wife, Margueritte, who allows me to pursue my dreams; and our three sons, Reid, Parker, and Grant, who cheer me on. To the achievements of the Enterprise Technology Leadership scenius, where so many of the insights that went into this book came from.

From Steve: To my wife, Linh, the love of my life, who knows me better than I know myself.

CONTENTS

Part 2: The Theory and Practice of Vibe Coding

Part 3: The Tools and Techniques of Vibe Coding

Part 4: Going Big: Beyond Individual Developer Productivity

FIGURES AND TABLES

FOREWORD

DARIO AMODEI, CEO AND COFOUNDER, ANTHROPIC

"Vibe coding" is both an inspired term and a misleading one. It's inspired because it describes so perfectly the feeling of telling an AI kind of, sort of what you want and watching it transform those vibes into a workable piece of software. But it's also misleading, because it's a jokey term that can make the whole enterprise seem unserious or frivolous.

In fact, vibe coding—that is, using everyday language to direct an AI model to write software code for you, and conversing back and forth with the model to improve the code it writes—is deadly serious. As of mid-2025, it's the only coding game in town.

In this book, Gene and Steve write about immense productivity increases in software work due to the existence of coding agents. That's exactly what we see at my company. They write about humans doing less and less of the actual writing of code, and yet producing software far quicker. That's also happening here. And they also write about engineers having great fun along the way. We see a lot of that too.

At my company, we train the models (like Claude) and the coding agents (like Claude Code), and then use them to improve future versions of themselves. It's all part of what we've seen for a few years now: a smooth exponential of accelerating AI progress, where things become unrecognizable rather quickly, even compared to a few months beforehand. The sudden arrival of vibe coding is a qualitative shift in how we work, but it's also part of a relentless upward spiral of AI capabilities that shows no sign of slowing down.

Some might (quite rightly) find this frightening. One day soon, will human coders suddenly lose their role in software engineering? I think there's still a lot of space for comparative advantage. That is, even if you think AIs will become better than humans at effectively all cognitive tasks (including

coding, but everything else too), there'll still be a long period where it makes sense for humans to set the goals, unstick the AI when it gets stuck, and so on. In other words, it'll still make sense to vibe code—and that's why Gene and Steve have done everyone such a service by writing such a comprehensive and practical introduction to it.

These changes that are revolutionizing software development are fascinating in and of themselves. But there's an even wider point here. I think of software as a "leading indicator" of AI's impact on the labor market: It'll give us an early look at the successes and failures of working with AI models to massively scale up (and speed up) the tasks we work on every day.

Of course, it's "easy" (in a relative sense) for AI to affect software engineering compared to fields like science or medicine: It makes AI easy to deploy, it generally avoids the messy physical world since it's contained within computers, and it doesn't bump up against so many societal "blockers" (like privacy laws for medical data) that could slow it down. But even though it might not be representative, it's still informative to see it play out and to attempt to extrapolate how AI agents could affect the rest of the economy.

We aren't going to change the face of science overnight with "vibe experiments" or "vibe drug trials". The physical world will always be there to get in the way; studies and medical advances inevitably take time. But we should view it as a top target for humanity to replicate the sorts of AI-led gains we're seeing in software engineering in other important fields.

It won't be straightforward. In the book, there are numerous examples of AI agents getting it wrong—deleting sections of your code, ignoring your instructions, "gaming" the tasks that you set. The researchers at Anthropic are working hard to understand these kinds of "misaligned" actions, whether they come about through error or "intention" on the part of the model.

While they remain in the software-development realm, most of these failures do not seem to have the potential for catastrophic (or existential) risk—though I hardly need to explain why "hundreds of individual agents taking autonomous actions over several days on your cluster" might still be concerning from the perspective of AI safety. I think what we learn from the coming surge of software-building LLM agents will give us a useful heads-up as to how AI might go wrong in bigger ways. And of course, AI software agents will help us design the systems to spot where other AIs are going off the rails.

But I don't want to make it sound like we should only read about AI's effect on software engineering because we're really interested in other stuff like science or safety testing. As is amply demonstrated in this book, even if AI agents were restricted to building software, we'd still be standing at the edge of a huge transformation. Vibe coding is a whole new way of working: We should expect to see entirely new, economy-boosting advances in software and engineering as a result. At the very least, a lot more software is going to get written.

That transformation is the best reason for reading this book. None of us can predict exactly how it'll go, but we can try to adapt, right now, to what's staring us in the face. In Steve's post from earlier this year, "Revenge of the Junior Developer," he pointed out the following common mistake:

> *Don't fall prey to the tempting work-deferral trap. Saying "It'll be way faster in 6 months, so I'll just push this work out 6 months" is like saying, "I'm going to wait until traffic dies down." Your drive will be shorter, sure. But you will arrive last.*

It will indeed be faster in six months. As I said above, the exponential is still the best way to think about AI. Take it from someone who employs many of the best coders in the world: The "vibe coding" way of working is here to stay. If you're going to be doing any coding at all—if you're going to use that comparative advantage—you need to get involved with vibe coding today. This book explains how.

—Dario Amodei
CEO and Cofounder, Anthropic
July 2025

PREFACE

READ THIS FIRST

Vibe coding seems to be reinventing how we build software. From our experience, it elevates the limits of what we can achieve, speeds up how we build software, improves how we learn and adapt, changes how we collaborate, expands who can meaningfully contribute, and even increases the amount of joy we experience as developers.

In short, we believe vibe coding may be the best thing that happened to developers since…well, ever.

It reminds us of what happened in the 1990s. Early adopters who recognized the importance of the internet became unstoppable and turned into companies like the legendary FAANGs (Facebook, Amazon, Apple, Netflix, Google), while skeptics dismissed the transformation as hype. The pattern appears to be playing out again, only faster and with higher stakes. The gap between those embracing these new ways of working with AI and those clinging to the old ways widens every day.

Vibe coding can change your life, like it changed ours. Mastering vibe coding enables you to take on ambitious projects, work faster and more autonomously, and, perhaps most importantly, rediscover the joy of building software on your own terms. This applies whether you're a senior architect, a recent boot camp graduate writing your first professional lines of code, or someone who stepped away from programming years ago but senses exciting new possibilities.

To set the stage for this book, we wanted to share our personal moments of revelation—those instances when we each realized that vibe coding was yielding transformative experiences that changed our perspectives:

Steve's Aha Moment: In March 2025, I experienced something that completely upended my multi-decade programming career. I've been

building a game on the side for over thirty years, and it had thousands of TODOs and unfixed bugs that seemed destined to remain untouched. After connecting an AI coding agent to a browser automation tool, I watched in disbelief as it started diagnosing and fixing UI bugs in my application. That night, I couldn't sleep—not from worry, but from excitement! After that, with the help of an AI coding agent, for certain work streams I was writing thousands of lines of high-quality, well-tested code daily while simultaneously writing this book. Suddenly, fixing all those game bugs seemed within reach! Though I was deeply skeptical of technology hype, I had to admit that this was new, important, exciting, and was going to change coding forever.

Gene's Aha Moment: I was certain that my best programming days were behind me. Then in February 2024, I asked ChatGPT to write code to extract video playback times from a YouTube screenshot. It analyzed the image, looking for the video progress indicator using Java graphics libraries I'd never used. When the code worked on the first try, I sat slack-jawed. But what changed my life was the forty-seven-minute pair programming session with Steve, where we built a working video excerpting tool that I'd wanted to write for years, but it seemed too daunting. That moment changed everything for me. Projects that would have taken months became weekend tasks. If you've ever abandoned coding dreams because the technical overhead seemed overwhelming, or if you're skeptical that AI could restructure how you work, this book might change your perspective as profoundly as those forty-seven minutes changed mine.

Over the last year, we have been using AI ourselves while studying how it will change the software development world. We know many claims about AI and coding sound extraordinary—even we were skeptical at first. That's why, throughout this book, we'll share our experiences, as well as the hard data and concrete examples that convinced us. If you're skeptical, we understand completely. We felt the same way. This book distills what we've learned through hard-won battles:

- **Part 1:** Why vibe coding matters.
- **Part 2:** The theory and your first steps, where we cover fundamentals and the new mental models needed to be successful.
- **Part 3:** The tools and techniques of vibe coding across your development workflow, including the inner, middle, and outer developer loops.
- **Part 4:** Scaling up and reshaping the organizations of the future.

While some of the finer details may be outdated by the time you read this—that's the price of exponential change—the core principles we share have remained consistent even as we've evolved from chat-based coding to autonomous agents to coordinating groups of agents. These principles will guide you through the change today and in the years to come, whether you're an experienced engineer or a novice straight out of school.

Some say that giving developers AI could be as impactful as the introduction of electricity was for manufacturing, and we're delighted by this analogy. AI improves productivity, and as we write about in this book, changes many things about software work and who does it. But using it comes with new risks and dangers.

We acknowledge that whenever someone suggests that "your job is changing," it can sound scary. Changes in our jobs are one of life's biggest stressors, up there with changes in relationships and changing where you live. We've both at times felt serious frustration about the learning curve and the uncertainty around what vibe coding does to the developer role, and we've watched others face it too.

However, we've watched many people try this amazing new technology with courage and curiosity and learn new habits, and they have told us of the value it has created for them. You'll see that it's not as difficult as you might imagine. Moreover, we were pleasantly surprised to find that vibe coding is incredibly fun, though we love old-school coding too. And we have found that AI can change your work/life balance in surprising and welcome ways.

The good news is that you're not too late…yet. Start now, practice daily, and push past the initial challenges. Your productivity will multiply, your

ambitions will grow, and most importantly, you'll rediscover the sheer joy of building software when you're elevated above the bottleneck of typing in every line of code by hand.

The future of coding has already arrived. Let's dive in.

INTRODUCTION

Dr. Erik Meijer, a visionary Dutch computer scientist with a lifelong penchant for tie-dyed shirts, is one of the most influential figures in programming language development. His lifetime of contributions have shaped how millions of developers write code every day, from his groundbreaking work on Visual Basic to his work on C#, Haskell, LINQ, and Hack.* Few people on Earth can claim such deep expertise in language design and implementation. And yet, in 2024, Dr. Meijer gleefully made this striking and startling declaration:

The days of writing code by hand are coming to an end.[1]

When we heard Dr. Meijer make this claim, we were both excited. It was one of the most important and validating confirmations of something we had started to suspect over the last year—that coding is changing right underneath us. So, why would such a prominent programming language pioneer make such a polarizing claim, one that implies that much of his life's work would soon become obsolete? Because he sees what we see: AI shifts how humans create software.

We're witnessing this transformation happen across the industry. At Adidas, seven hundred developers using AI coding tools reported a 50% increase in what they call "Happy Time"[2]—hours spent on creative work they enjoy, rather than wrestling with brittle tests or debugging trivial errors. High-performing teams now spend 70% of their time directly coding, compared to 30% for teams using traditional methods.[3]

* Dr. Meijer was one of the core members of the team that built Facebook Hack, which was released in 2014. Hack was successfully deployed across Facebook's PHP code base—millions of lines of code—within the space of a year. Facebook engineers adopted the language because it reduced runtime errors through static typing while preserving PHP's rapid development cycle, where type safety and improved tooling helped thousands of engineers work more confidently and efficiently across one of the largest code bases in the world.

Even more telling are the stories from developers who had left programming. A former machine learning engineer who hadn't written code in nearly twenty years successfully built a calendar synchronization tool in her first session with AI assistance. Even Kent Beck, creator of Extreme Programming, excitedly shared how he's "coding at 3am for the first time in decades!"[4]

For decades, programming has meant laboriously typing code by hand, hunting down syntax errors, and spending countless hours on Stack Overflow. That era is ending. We're living through a fundamental shift in software development that is redefining how we code, who can code, and what is possible to build.

What we and Dr. Meijer saw now has a name: *vibe coding*. It was coined by the legendary Dr. Andrej Karpathy,[5] who has been at the forefront of AI research for a decade, to describe a new way of programming.

When we say vibe coding, we mean that you have AI write your code—you're no longer typing in code by hand (like a photographer going into a darkroom to manually develop their film).

Although the most visible and glamorous part is code generation, AI helps with the whole software life cycle. AI becomes your partner in brainstorming architecture, researching solutions, implementing features, crafting tests, and hardening security. Vibe coding happens whenever you're directing rather than typing, allowing AI to shoulder the implementation while you focus on vision and verification.

Let's Be Precise: What Is Vibe Coding?

As with any newfangled term, there's a lot of disagreement and misinformation about what vibe coding is. Plenty of people and the media have painted it as "turning off your brain." However, this is far from how the rest of the professional world is using it. Before we go any further, let's get precise and define what we mean when we talk about vibe coding, agents, etc.

When we refer to *manual coding* or *traditional coding*, we're talking about pre-AI style software development, where you type in code by hand.

In 2021, we saw AI-generated *code completions*, where the IDE (integrated developer environment) would auto-complete code based on what you had

typed (like your phone auto-suggesting words as you text). GitHub Copilot pioneered this capability, and it's in almost every coding assistant product on the market today. Research by Dr. Eirini Kalliamvakou, showed this sped up some coding tasks by 50%,[6] but coding is still labor-intensive work.*

Chat coding is one of the successors to code completions. Beginning in 2023, you could ask AI to examine and modify code or generate new code, and it would emit an answer. It may seem quaint now, but you had to copy the answer back into your IDE by hand. Over time, the tooling has become faster and more fluid, but chat is still a back-and-forth interaction. Whenever we say "chat," we mean a conversation with AI unfolding one turn at a time. Many first discovered this style of coding with the release of OpenAI's ChatGPT-4o in May 2024.

Agentic coding (where AI autonomously generates, refines, and manages code) appeared in early 2025, and is a game-changing step up from chat. In this workflow, coding agents act like real developers and actively solve problems using the tools and the environment. Agentic coding is increasingly predicted to replace a significant portion of coding by the end of 2026.†

Agentic coding had been long conjectured, and many of us were first exposed to it with the announcement from Cognition AI's Devin, an autonomous AI assistant designed to collaborate with humans on software development tasks, in March 2024.[8] However, it wasn't until early 2025, with the release of Claude Code from Anthropic, that agentic coding took the developer world by storm. Claude Code is a terminal application that you interact with. You tell it what you want it to do, and it modifies files to implement. It can even run tests and execute programs (including mini utilities it builds for itself).

With agentic coding, instead of AI telling you what to type, the agent makes the changes and uses the tools itself. This speeds the development life cycle far more than you would expect.‡

---

* Dr. Kalliamvakou and team measured two populations to write an HTTP server in JavaScript, one with GitHub Copilot and the other without.

† Mark Zuckerberg, founder and CEO of Meta, believes AI will write 50% of Meta's code by 2026.[7] Dario Amodei, Anthropic cofounder and CEO, believes it will be 100% by that time.

‡ And it comes as a real shock the first time you use it, but you'll never want to go back. After using agentic coding assistants, you'll become aware of the rare times AI is telling you to type something. It almost feels like you're getting bossed around.

If you're in development today, you've probably already been using AI and coding assistants or have at least dabbled. The list of players in the space is long and includes a spectrum of offerings from chat to limited coding agents to extremely powerful autonomous coding agents (e.g., Aider, Augment Code, Anthropic's Claude Code, Bolt, Cline, Amazon Q, Cursor, GitHub Copilot, Google's Cloud Code, Jules, JetBrains's Junie, Lovable, OpenAI's Codex, Replit, Roo Code, Sourcegraph's Amp, Tabnine, and Windsurf).

These products make different choices about what to offer and where to offer it. Some are still mostly completions or chat. Some have limited agents. Some offer full-featured, semi-autonomous agentic coding assistants. Some support running many agents together. Some coding assistants live in your IDE, some are standalone IDEs themselves, and some are command-line tools. Some support complex enterprise environments, while others are geared more toward casual coders. Many coding assistants support multiple models, but some align themselves to a single model family for performance, reliability, or cost reasons.

So, in this mixed landscape of manual coding, chat coding, and agentic coding, let's examine what vibe coding is and where it fits.

For starters, you don't *have to* "turn your brain off"—as many have wrongly implied. You'll often be an active participant. Instead of writing the code yourself, with vibe coding you're overseeing your AI assistant doing it for you and critiquing its results.

We and many others have felt that, at times, you can be 10x more productive with vibe coding compared to manual coding. We know this sounds like hype—we were skeptical too. In Chapter 1, we'll walk you through a detailed, real-world example of how Gene wrote over 4,000 lines of production code in just four days to help this book make its deadline.

And as Gene did early in the DevOps movement, we're both working on research to quantify the impacts of AI on development and on the conditions required for AI to create value, jointly working with Google's DORA research group. We'll talk more about this in Part 4. But it's clear that vibe coding will be reshaping our work for decades to come.*

* Note: Throughout this book, we'll use terms like vibe coding and chat-oriented programming (which was the original title for this book, pre-Karpathy) interchangeably—but always with the understanding that we use appropriate levels of engineering discipline.

So, What Are the Benefits of Vibe Coding?

Vibe coding lets you build things *faster*, be more *ambitious* about what you can build, build things more *autonomously*, have more *fun*, and explore more *options*. This is what we're calling FAAFO (or sometimes "the good FAAFO," to contrast it with certain other kinds). Let's look at each in turn.

First, vibe coding helps you write code *faster*. Tasks that once took months or weeks can now be done in a day. And tasks that took days can now be completed in hours. This acceleration comes not only from code generation but also from having AI help with debugging, testing, and documentation. Projects that have been sitting on the back burner for years can finally see the light of day.

Second, vibe coding enables you to be more *ambitious* about what you can build. It expands both ends of your project spectrum. It brings seemingly impossible projects within reach, while simultaneously making small tasks with marginal ROI easier to take on as well. This is due to the speed, vast knowledge, and capabilities of AI. Vibe coding reshapes your approach to development, eliminating many of the painful trade-offs that have always constrained what gets built.

Third, vibe coding allows you to do work *autonomously*, often being able to complete things that previously required multiple people or teams. That's a bigger deal than it might seem. Features that once demanded specialists from multiple disciplines can now be handled by a single non-specialist developer with AI assistance. Being able to work autonomously or alone on a task or project eliminates two expensive taxes: It reduces the coordination costs (scheduling meetings, aligning priorities, waiting for availability) and the communication challenges (where teammates cannot read each other's minds but must still create a shared goal and vision of what to build and how). Working more autonomously or alone with AI significantly reduces or removes these obstacles.

Fourth, vibe coding makes programming more *fun*. You're spared from the least enjoyable parts of programming, such as debugging syntax errors, wrestling with unfamiliar libraries, or switching test infrastructure for the *n*th time. Instead, you can focus on solving user problems, building cool stuff, and getting things done. Working with AI is also strangely addictive, an aspect we

explore in the book. You might be tempted to discount the fun dimension, but we think it's one of the most valuable, because it's bringing people out of retirement, attracting non-programmers, and encouraging leaders to take on more programming work. That's a deep societal change in the works.

Finally—and this is possibly the most important and transformative dimension of all—vibe coding increases your ability to explore *options*, either to find a solution or to mitigate risks. Instead of committing to a single approach early on, you can rapidly prototype multiple ways to solve the problem and evaluate their trade-offs. We'll revisit this topic often, so that when you recognize a problem where exploration will help, you'll reflexively spin up parallel investigations. FAAFO!

Why This Book Now

We're writing this book in 2025, a time of dizzying and relentless innovation. Every week it feels like years of breakthroughs are happening at once: new models, tools, and techniques. Each day seems to move faster than the last.

This book may seem like an ambitious goal in the face of exponential change. After all, since 2020, the pace of AI-assisted programming has been neck-snapping, moving swiftly from code completions to chat programming to in-place editing with chat to coding agents to clusters of agents to badged agent employees who will start showing up soon on Slack and Teams, ready to help you.[9] But despite all the change, as programmers we often find ourselves doing many of the same kinds of things we've always done: design, task decomposition, verification, hardening, deploying, monitoring, merging, cleanups, etc. These skills remain relevant and important no matter who is writing the code.

The truth is, we're all figuring out this new landscape together. Early adopters like us have made countless mistakes, discovered unexpected pitfalls, and developed patterns that work reliably. We've written code with AI that we're proud of, and we've also created messes we're embarrassed to admit to. By sharing these hard-won insights, we hope to help you avoid the same painful lessons while accelerating your journey toward mastering this new paradigm.

We genuinely believe that if you wait until the technology stabilizes, you're at risk of being left behind. By learning these techniques now, you'll be positioned to adapt as the tools evolve, rather than scrambling to catch up when your competitors have already mastered them. (And if AI can make every developer more productive, organizations that adopt this technology will pull ahead.)

Our goal in this book is to explain why vibe coding matters *and* how to do it effectively—even at the team and enterprise level. We'll do that by focusing on enduring principles and techniques that will be relevant regardless of which AI models or tools you're using, and remain relevant as they become smarter and more autonomous. Rather than offering soon-outdated tutorials on features, we'll equip you with the mental models and approaches that will serve you well through the continuing evolution of AI-assisted development.

Throughout this book, we'll use a professional kitchen as a metaphor for vibe coding. You're the head (or executive) chef of the kitchen, and AI represents the army of chefs who help bring your vision to life. (See Figure 0.1.) AI serves as your sous chef (your second in command) who understands your intentions, handles intricate preparations, and executes complex techniques with precision under your guidance. But AI is also your army of station chefs and cooks, specialists who help handle various technical details.

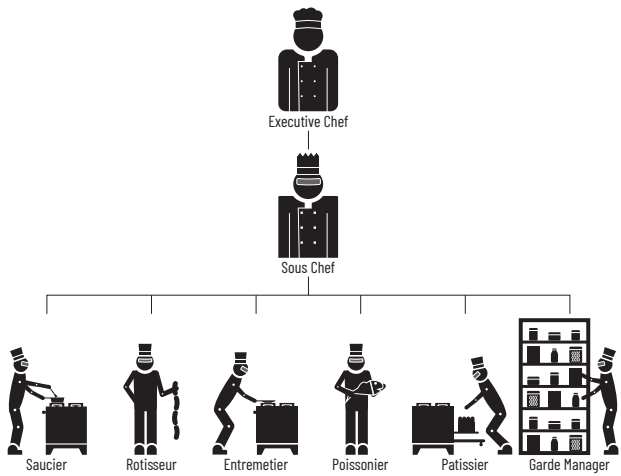

Figure 0.1: The Kitchen Brigade

These chefs have memorized every cookbook ever written, work at lightning speed, and never sleep. They will, however, occasionally suggest using ingredients that don't exist or insist on cooking techniques that make no sense whatsoever. They can be like overly eager interns or junior engineers: highly capable and expertly trained, but also possessing the potential to get out of control and do a lot of damage. We've seen firsthand how vibe coding can go wrong, silently deleting critical code and tests, ignoring instructions, creating pathologically unreadable and untestable code, and other setbacks or near misses. In the not-too-distant future, you'll have ten or more of these AI assistants working for you. As head chef, you, not the AI, are accountable for the team's outcomes.

It's like playing a slot machine with infinite payout but also infinite loss potential. Without the proper safeguards, you might watch your helpful AI assistant transform into the Swedish Chef from the Muppets (or maybe Dr. Frankenstein's monster), leaving a trail of unintentional destruction in its wake. But vibe coding is here to stay and has the potential to make more positive impacts than negative, if you follow the guidelines in this book.

As AI gets smarter, your workflow with vibe coding will accelerate. You'll accomplish increasingly ambitious things you never thought possible, with nobody but your AI kitchen staff assisting you. The principles we present in this book will help you approach vibe coding with confidence, security, and resilience. Our goal is to replace any apprehension with skill, empowering you to direct AI systems to create smash-hit software, maybe paving the path to becoming a celebrity chef managing an international culinary empire.

Our Journeys to Vibe Coding

We both came to vibe coding from different paths—Steve as a veteran programmer with decades of experience at major tech companies, and Gene after stepping away from hands-on coding for nearly two decades. Despite our different backgrounds, we both came to the same conclusion: AI is transforming how software is created, and the impact is far greater than most realize. Here are our stories.

Steve's Journey: From Skeptic to Believer

I've been in the industry for over thirty years, including almost twenty years at Amazon and Google. Throughout my career, I've blogged about developer productivity because I care about it deeply. Whether it's telling people to adopt platform-first architecture or to use safer programming languages or to stop deprecating APIs so aggressively that developers on your platform can't keep up.

Everyone wants to work faster. Our tools, as good as they are, always hold us back. At Google, I took productivity head-on by leading the creation of Kythe,* a rich knowledge base for understanding source code. We combined Kythe with Google Code Search, which became a dizzyingly powerful developer productivity tool, one that had a 99% satisfaction rating at Google when the next-best tool was in the mid 1980s. But unfortunately for the world, it was internal, for Google's use only.

The best code search tool outside Google is Sourcegraph, and years later, in 2022, I became their Head of Engineering. It was a match that seemed almost predestined. But by early 2024, I had started to worry that I could no longer make good decisions as a technology leader unless I deeply understood the radical technology change that was transpiring. I was leading, but without coding, I was leading from the sidelines.

So, I stepped out of my role as a technology leader—where I've spent much of my career—to put my boots back on the ground and find out what was going on with AI. I started coding again for the first time in years. And I was far from alone. Many other engineering leaders at all levels, all the way up to big-company C-suite executives, had been doing the same, because of AI. This delights me more than words can tell.

Moreover, another big group of what I think of as "Archmage"† coders are coming out of retirement, swinging big. I think it's clear why. AI in 2025 takes care of most of the tedium of programming, making it fun again—and that's bringing back people who thought they had given up coding forever.

I had a pet project, Wyvern, a multiplayer online game I've tinkered on since 1995. It has had over 250,000 players, over sixty volunteer content and

* Originally called "Grok" when I pitched the project in 2008 and was allowed to start work on it.

† In fantasy settings, an Archmage is the most powerful, highest-ranking wizard or mage.

code contributors, and over four million lines of code and configuration, and over thirty years of love.

Unfortunately, by 2022 the code base had become as immovable as a mildly deceased elephant. That's what happens to code bases over thirty years. They gain weight until they can't move. Achieving all our aspirations and fixing all the problems had become too much work, and I put the game in maintenance mode. Without consciously deciding to do so, after all these years, I had given up coding—even as a hobby. And I thought that was the end of it.

In early 2024, I had the privilege and pleasure of meeting Gene Kim, who had reached out to invite me to speak at his top-tier Enterprise Technology Leadership Summit in Las Vegas. During our first call, we realized we were both looking at the same problems with different lenses, and we got excited, since it looked like we'd uncovered something big. Our subsequent year of vibe coding exploration, which included pair programming sessions, interviews with experts, long debates, and, ultimately, writing this book, has been one of the most rewarding periods of my career.

AI brought us both back to coding. Coding is different now. It's both easier and harder. There was almost no literature or useful information about vibe coding when we started in mid-2024; it didn't even have its name yet. But we knew we wanted to learn how to do it right and share that knowledge with others. That is how we embarked on the journey that led to this book.

In that time, I've had some life-changing experiences with AI, stories that we'll share and explore in this book. I could not have predicted that I would be coding again. Heck, I told my *doctor* I was done with coding…and then three months later, laughingly had to tell him I was back, because AI is doing all the hard stuff now.

For my whole career, all I've wanted is to build things faster—and now, it's finally happening. In certain contexts, I'm often able to write thousands of lines of high-quality, well-tested code per day—while also writing a book eight hours a day. It's at least an order of magnitude improvement over my career average, and I'm doing it on the side. It's nuts. And that's why I can barely sleep lately. I have too much to do. Everything is achievable now.

I'm completely addicted to this new way of coding, and I'm having the time of my life.

Gene's Journey: Returning to Coding After Seventeen Years

For over two decades, I've researched and written about high-performing technology organizations. But my personal journey back to programming demonstrates how GenAI has changed my life by helping me become a better developer than I ever dreamed I could be.

My journey with software began when I created a UNIX security tool during an independent study project at Purdue University in 1992, which was later commercialized as Tripwire. I was there for thirteen years as founder and CTO, and I left shortly after the company filed for its IPO in 2010. My first jobs after getting my graduate degree in computer science in 1995 were writing software full-time, primarily C and C++. I would never claim I was particularly good at coding, because I knew many people who were obviously better at it than me.

In 1998, I transitioned into leadership roles. I wrote my last line of production code for a long time. For a decade, I became "non-technical." I spent far more time in Excel and PowerPoint than in an IDE,* occasionally writing Perl and Ruby scripts for system administration.

I rediscovered the joy of programming in 2016 when I learned Clojure†—but I admit I glossed over how difficult that journey was. The learning curve was like a sheer cliff. For over a year, I climbed huge hurdles, either trying to puzzle things out or desperately searching for answers on the internet.

The only way I got through it was sheer luck. Two experts were willing to teach me (thank you, Dr. John Launchbury and Mike Nygard). Without them and their generosity, I would have given up trying to code again. (I can only imagine how much easier this learning curve would have been with AI as an infinitely patient teacher and coach—explaining concepts, reviewing code, and giving advice at every step.)

I finally met Steve Yegge in June 2024, whose work I've admired for over a decade. Anyone who has studied DevOps or modularity knows his work.

---

* Andrew Flick is a senior director of marketing at Microsoft. Decades ago, he was a C# MVP, a distinction that Microsoft gives to the top technology experts who share knowledge and contribute to the community. After moving into marketing, he said he had become stuck on the "PWE tech stack"—PowerPoint, Word, Excel.

† A functional Lisp programming language that Steve loves.

I can't count how many times I've cited his famous rant about Google and Amazon[10] that landed him on the front page of *The Wall Street Journal*.[11] It's one of the best accounts of how and why Amazon rearchitected their monolith, liberating thousands of developers to independently develop, test, and deploy software again.

After he wrote his "Death of the Junior Developer" post,[12] Steve offered to pair program with me to show me the power of vibe coding, where AI helps write the code (which at the time he was calling CHOP or chat-oriented programming).

What happened next astounded me. In just forty-seven minutes of pair programming with Steve using chat coding, I built a working video excerpting tool that had been on my "someday" list for years. This was the kind of project that kept getting pushed to "maybe next month"—not because these projects were particularly difficult, but because the perceived benefit wasn't high enough to warrant days (or weeks) of work.

Throughout the development of this book, I vibe coded tools to help in the writing process. What started as a web application to reduce copying/pasting and switching between various tools became a Google Docs Add-on that I wrote in three hours, despite never having written one before. I rewrote it a third time as a terminal application because the Add-on was too slow.

This tool served us well—it slung over 71 million tokens, accruing over 3,000 hours of LLM processing time doing draft generation and draft ranking. Writing this, I was stunned to discover that I started this code base only thirty days ago. During that time, I had created 397 commits and 35 branches, many abandoned after discovering those experiments were dead ends. This is at least 10x higher than I could do before vibe coding—and as Steve mentioned, I did it on the side, while writing the book that it was supporting.

There is absolutely no way I could have done all of this without AI. Projects that would have taken weeks now take hours. AI helps me be faster and far more ambitious in what I can build.

Most importantly, I'm having more fun and experiencing more joy programming now than ever before. I'm proud of the things I've built. Projects that I would have deferred eternally are now 100% within reach. And I don't have to be selective—I can do them all. The economics of what's worth building have shifted radically, and I'm tackling challenges I wouldn't have dreamed of attempting before.

From Our Journeys to Yours

Our personal stories reflect how vibe coding expands what's possible for everyone who creates or works with software. Whether you're an industry veteran like Steve, someone returning to coding after years away like Gene, or someone who is "tech adjacent," such as product managers or infrastructure experts who work with developer teams, these tools and techniques transform how you build software.

The coding revolution is still in its early days. The experience we've gained—sometimes through trial and error, sometimes through wild success—forms the foundation of this book. We hope it helps you navigate this rapidly changing landscape and discover the same joy and productivity we've found in this new way of creating software.

Who This Book Is For

This book is for any developer who is building things right now—no matter whether you're building front-end applications in React and JavaScript, back-end servers in Kotlin or Go, mobile applications for Android or iOS, data transformations in Python or R, or writing and managing infrastructure in Terraform or Kubernetes. Our book applies to all types of software development, in all languages and frameworks.

You may be a junior engineer working on a feature, a senior engineer shepherding a giant migration, or a senior architect tasked with figuring out how to make a service more reliable. You may be a new boot camp grad who wants to build up technical chops to impress your new employer. Whatever your role, vibe coding can help you solve problems and build cool things you never thought possible and have far more fun doing it.

You may be a CTO or technology executive who hasn't programmed in decades. If so, vibe coding is for you too—it enables you to rediscover the joy of coding.

Let's face it. Most of us became programmers because we wanted to build things, not to spend our days Googling syntax and copying/pasting from Stack Overflow. The dirty secret of programming has always been that implementation details and busywork consume most of our time, leaving precious little for creation and problem-solving. But with vibe coding, projects that

were "too difficult" or "not worth the effort" become doable in afternoons rather than weeks. Kent Beck summed it up for a generation of programmers when he said, "I feel young again!"[13]

We've written this book with several audiences in mind. Let's dive a little deeper into some of those. Perhaps you'll recognize yourself in one of these descriptions:

Software Engineers, ML Engineers, AI Engineers: You're spending way too much time learning new frameworks and fighting with package managers instead of solving interesting problems. Vibe coding lets you skip past those tedious details and focus on what matters. You'll crank out great software of all shapes and sizes for yourself and for others. And you'll finally start up those ambitious projects that kept sliding to the "maybe someday" list.

Senior and Principal Engineers: You rose to your position by seeing the dangers no one else could and steering projects to success. Vibe coding now turns those insights into superpowers. It frees you from rote coding so you can orchestrate both human and AI assistants, while focusing on the gnarly architectural puzzles. We'll have tips for you, regardless of whether you're a maverick solo coder or a principal engineer in big tech or enterprise. The result of adopting vibe coding will be a dramatic expansion of your strategic reach, letting you shape multiple initiatives simultaneously instead of firefighting one at a time.

Technology Leaders: Remember when you built stuff yourself instead of being in meetings about building stuff? Those were good times. Vibe coding brings that back. You can prototype and begin hardening your ideas yourself, right now. You can build stuff while you talk about it in meetings. It's a bit self-indulgent, to be sure, but why not have a little fun. Practicing it will also help you make better strategic decisions, because you'll have personally experienced how this technology transforms software development and how it opens up a new horizon of possibilities.

Returning to Coding: Some of you have become "non-technical," as your career path led you away from hands-on development. But you're not really non-technical, are you? It's just that the environment setup requirements over the years keep getting ridiculously harder, so you stopped coding. It's not just you—modern development is overwhelming to everyone. Thankfully, vibe coding lets you skip countless hours of tutorials and infrastructure setup. AI can handle the technical details that would have been frustrating roadblocks, including setting up a developer environment. And let's not forget, it can also write the code. You can build useful things again without getting buried in implementation complexities.

Product Owners and UX: You have a bit of a programming background, and you know how software works at a high level. You've had this killer idea for months, a minor front-end feature, but engineering keeps pushing it back because they're "at capacity." How about if you could do it yourself? Vibe coding can help you implement a real feature or create a working prototype of a big idea in hours to days. It can completely reshape the conversation when you demo something that the engineers told you was going to be "too difficult to build."

Infrastructure Engineers (DBAs, SREs, Cloud, Build): For too long, the industry has maintained an artificial divide between "real developers" and "infrastructure folks." Vibe coding obliterates that distinction. You can create real applications, like any developer, without needing to master multiple new programming languages or frameworks. You'll also be able to create world-class tools to solve your own problems: performance analyzers, migration utilities, scaling automation, you name it.

"Level 99 Heroes Logging Back In": You were one of the most badass programmers on the planet. And then one day, after npm screwed you one too many times (I mean, what even is npm?) you finally threw in the towel. This wasn't worth it. Let the kids do this

> crap. But look out, world, a whole generation of retired programmers is on their way back with a vengeance to show the world what they're capable of.

Whatever your background, the techniques we share in this book will transform how you work with code, making programming more accessible, more productive, and—most importantly—more fun. You bring the problems, and AI can help you with the rest.

What We Assume You Already Know

We wrote this book assuming you have some experience in programming, whether it's been a few months, years, or decades since you last wrote a line of code. We also assume you're familiar with concepts like version control and have a general understanding of terms like commits, code reviews, unit testing, code linting, compiler errors, and so forth.

While this book is intended for people with some coding experience, we believe vibe coding will eventually make programming more accessible for everyone. If you aren't familiar with all of these topics, don't fret. Although we do dive into some technical topics in this book, we're hoping you'll still find the book readable regardless of your level of experience.

We also include a glossary at the end of the book for terms that might be a bit unfamiliar, helping you brush up on essential jargon before whipping up your next coding masterpiece. (We're also hoping to create more beginner-friendly resources in potential follow-up guides, so everyone can eventually step into the kitchen of coding.)

Readers Who Also Might Be Interested

We've made the case that vibe coding is for professional developers and leaders. But, we also see it becoming increasingly accessible to the people who work around developers or aspire to become one. Steve recently shared with Gene how his VP of finance was on the top of the Sourcegraph Amp coding-agent leaderboard for most lines of code written in one week—earning the admiration of developers across the organization. We hope that the following audiences will also find value in this book:

Students: You're entering the industry at a time that is simultaneously scary but also ideal. The job market may be uncertain, but one thing is certain: All developer jobs are now AI jobs. You'll be learning how to partner with AI to create software, rather than memorizing syntax, APIs, and framework intricacies. Master vibe coding now, and you'll get the jump on experienced developers who haven't ramped up yet. You'll complete assignments that will impress senior engineers and build a portfolio of projects that will wow anyone who interviews you. And you'll begin building up vital skills required for understanding the strengths and limitations of AI, which will put you ahead of the pack.

Tech Adjacent Roles (Program Managers, Analysts, QA, Customer Service, Sales, Finance, HR, Marketing): You've probably got several processes that could be automated if only you had a developer to help. With vibe coding, you can do it yourself. No more waiting in the priority queue behind "features that customers pay for." By taking matters into your own hands, you can finally streamline those organizational processes that never get any love. The organization will end up thanking you. (And the engineering organization will be both impressed and relieved that they didn't have to do it.)

We're sure we've missed some audiences. If you're not sure whether vibe coding is for you, turn to any random page in this book and skim it. If you feel that page speaks to you, then you're one of us. Welcome!

Beyond the Hype

Okay, you've read our stories, but you're still skeptical. Fair enough. Maybe your most senior engineers are giving PowerPoint presentations to the executives, complete with fancy graphs, to show how LLMs are not good at coding. We saw this happen in real life. Or maybe they're sending screenshots of

"lousy LLM coding results" to people to try to slow the AI train down. (And maybe you're one of these people.)

Steve is not someone who yields readily to hype. Most of his favorite tech is from the mid-to-late 1990s. His first five years professionally were spent programming in the Intel 8086 assembly language. He coded in Java without an IDE until 2011 and refused to learn Git until 2021. Steve is a bona fide late adopter.

Despite his technological conservatism, Steve is also a seasoned, possibly overcooked engineer, having written over a million lines of production code across more than thirty-five years in the industry, including at Amazon, Google, Grab, and Sourcegraph. You don't survive that long by chasing every shiny new framework that pops up on Hacker News. New technologies often have a lot of bugs, and Steve, who has seen many frameworks come and go, prefers to spend his time solving user problems rather than debugging new tech.

Gene built his reputation on years of rigorous, data-driven research. For the *State of DevOps Report*s, he and his colleagues surveyed over 36,000 technical professionals over six years to figure out what works in software delivery. That resulted in the famous "DORA metrics" of deployment frequency, deployment lead time, change success rate, and mean time to repair (MTTR). It helped bring CI/CD (continuous integration and delivery) mainstream. Gene eyes everything he encounters with professional rigor and a desire to measure and confirm any claims, especially anything called a "best practice."

We were both initially skeptical about using GenAI for coding. We don't blame you for being skeptical one bit. But as you've already read, we've both had numerous life-changing moments in the years post-ChatGPT. Later in the book, we'll describe some of the scientific literature on AI and developer-productivity, as well as the ambitious research we're undertaking to substantiate these claims.

Coding is changing beneath our feet. The skills that made developers valuable yesterday are not the same ones that will matter tomorrow. And we both believe one thing with absolute certainty: If you don't adapt to this shift, you may become irrelevant. And none of us wants that.

How to Read This Book

We've organized this book to accommodate different entry points, interests, and levels of experience with AI-assisted programming. Think of the four parts as independent but interlocking modules. Whether you're beginning your vibe coding journey or already working with AI tools daily, you can choose your own adventure, depending on the problems you're facing today.

Part 1 is the "why" of vibe coding. If you're intrigued but not yet sold on AI-assisted development, start here. We lay out the FAAFO benefits—fast, ambitious, autonomous, fun, optionality—through brief history lessons, personal war stories, case studies, and data points. Skeptics will find answers to the classic "show me the value" challenge, and newcomers will get the historical context that explains why this shift is unavoidable.

If you're already sold on vibe coding but still interested in the broader context, you may still be interested in the sections on why the AI revolution is different from previous decades of breakthroughs in development productivity and how AI impacts go beyond development.

Part 2 is the conceptual framework of how AI works. We move from high-level enthusiasm to a crash course in understanding the AI cognition of your new sous chefs, targeted at working developers. We explain context windows, task decomposition, and how vibe coding is conversational—a stark contrast to the rigor of prompt engineering. Moreover, there is absolutely no mention of matrix multiplication, tensors, or any math in this book, for that matter. This is for working developers who want to solve their own problems.

We discuss the ways AI can astound you one minute and frustrate you the next, so you can keep everything in perspective and cooper-

ate with these tools effectively. If you've ever wondered why AI nails a tricky refactor one minute and then trashes your unit test the next, we teach you why. We catalog the failure modes, show how to recognize them, and—most importantly—outline the conceptual guardrails that keep you coding safely. Think of this part as the kernel of education needed to prevent most common AI headaches.

Even if you've done some vibe coding before, you may find the deeper insights into AI's inner workings to be a helpful reality check. Mastering these concepts prevents the false starts and confusion that sometimes plague AI-assisted projects. You'll also see how the FAAFO mindset should change how you work.

Part 3 presents the tactics of your daily vibe coding. Here we present the practical and concrete practices for your inner (seconds), middle (hours), and outer (days) development loops. For each of the risks and bad outcomes we described in the previous parts, we describe how you can prevent those problems, detect AI slips or errors, and how to correct and recover.

We present guidance and lessons learned from our own experiences, as well as the experiences of others. We describe scripts we still run, reminders we give ourselves, and habits that have stuck after hundreds of coding sessions.

Part 4 is all about going big. Vibe coding changes more than how many keystrokes we're no longer typing. It also reshapes how we developers spend our time, the processes we become responsible for, team dynamics, and our architectural needs.

This final part is for tech leads, managers, and anyone newly responsible for coordinating fleets of human and AI contributors. You'll find guidance on how to introduce vibe coding into teams, how to set useful cultural norms that encourage learning, when and how to create organization-wide standards, the implications of AI sous chefs

working alongside human developers, hints on how you might measure productivity in an AI world, ideas on interviewing, and more.

If your calendar is packed and you need immediate leadership insights on how vibe coding and FAAFO affect work, feel free to jump straight here and then loop back to earlier parts when you want hands-on tactics or a refresher on the fundamentals. We also provide enterprise case studies of how vibe coding has affected real organizations building real systems.

Dive into the sections most useful to you, and revisit others later as your proficiency and curiosity evolve. Wherever you start, you'll find consistent emphasis on modularity, fast feedback loops, and maintaining high standards and rigorous judgment—the principles that make vibe coding transformative and rewarding.

PART 1

WHY VIBE CODE

Welcome to Part 1, where we make the case that vibe coding is the most significant shift in software development since, well, maybe ever. If you're curious about what all the AI and development buzz is about, or perhaps a little skeptical, you've come to the right place.

Think of this first section as laying the foundation for your new life as head chef in an AI-powered kitchen. We'll explore the seismic shifts happening right now, look back at decades of tech revolutions to see why this one is different, and introduce you to the FAAFO framework—fast, ambitious, autonomous, fun, and optionality—the five superpowers vibe coding bestows upon you.

We'll share our own "Aha!" moments, cautionary tales from the trenches, and inspiring stories of real-world developers already riding this wave. By the end of Part 1, you'll understand why we believe vibe coding is a whole new way of thinking, building, and succeeding in the world of software.

Here's a taste of what we present in Part 1:

Chapter 1: The Future Is Here (The Major Shift in Programming That Is Happening Right Now): See how science fiction is now your potential daily reality. We dive into how conversational AI is transforming the act of programming, allowing you to turn ideas into working software almost as fast as you can articulate them. We'll explore the emerging debate around vibe coding (from "No vibe coding!" to "10x speedups!"), and explain why, as a developer, you're evolving from a line cook into the head chef of your own AI-assisted kitchen.

Chapter 2: Programming: No Winners, Only Survivors: We take a whirlwind tour through the history of programming advancements—from assembly to high-level languages, from punch cards to sophisticated IDEs, and from dusty library shelves to the instant knowledge of the internet. Yet, despite these leaps, we'll explore why developers often still feel mired in complexity (hello, JavaScript toolchain). This chapter sets the stage for understanding why AI-assisted coding is bigger than step-function improvement and is more like the exponential graphics programming revolution over the decades.

Chapter 3: The Value Vibe Coding Brings: This is where we unpack the five dimensions of value that vibe coding unlocks: fast, ambitious, autonomous, fun, and optionality (FAAFO). We'll show you how AI is more than a speedup; it empowers you to tackle projects you once deemed impossible, accomplish solo feats that previously required teams, rediscover the sheer joy of coding, and explore multiple solutions before committing.

Chapter 4: The Dark Side: When Vibe Coding Goes Horribly Wrong: With any technology revolution, such as electricity, comes the potential for some spectacular new dangers. We don't want to sugarcoat this. Vibe coding can be like a chainsaw. It can make you wildly more productive, but it can be dangerous. We'll share our lessons learned and how old practices and habits need to be modified to use the fantastic new technology. These cautionary tales aren't meant to scare you off, but to highlight why discipline, vigilance, and the "head chef" mindset are crucial as you unleash your gifted but occasionally erratic AI sous chef in your kitchen.

Chapter 5: AI Is Changing All Knowledge Work: Step back with us for a moment to see the bigger picture: AI is revolutionizing coding, and beyond that, it's beginning to reshape all knowledge work. We'll look at studies suggesting big impacts on high-wage jobs (yes, including ours) and discuss how, historically, making tasks easier has increased demand for skilled practitioners. Far from being the end of developer jobs, we argue this will lead to an explosion of new roles and opportunities, transforming the global economy on a scale not seen since the Industrial Revolution.

Chapter 6: Four Case Studies in Vibe Coding: Theory is great, but seeing is believing. We bring vibe coding to life with four case studies. You'll meet Luke Burton, an ex-Apple engineer, tackling a complex CNC firmware project as a hobbyist. You'll join our friend Christine Hudson as she returns to coding after nearly two decades, discovering the joy and power of AI assistance firsthand. And we'll

go inside Adidas and Booking.com to see how large enterprises are leveraging AI to help developers be productive and happier.

Chapter 7: What Skills to Learn: As your role shifts to head chef, you'll need to cultivate new skills. We focus on three essentials: creating fast and frequent feedback loops (because speed without control is chaos), embracing modularity (to enable parallel work and contain complexity), and, most importantly, reigniting your passion for learning and mastering your craft.

We've written Part 1 to be an eye-opener, a context-setter, and to make a compelling argument for why embracing vibe coding is a non-optional but also exciting development. As we mentioned, if you're already sold on vibe coding, you may want to skim this Part or skip to Part 2, where we start teaching you about the important internals of how your new AI sous chefs work.

CHAPTER 1

THE FUTURE IS HERE (THE MAJOR SHIFT IN PROGRAMMING THAT IS HAPPENING RIGHT NOW)

Since the 1960s, sci-fi like *Star Trek* has shown us a future where people casually talk with computers—they speak as if to a person, and the computer understands and executes their wishes. We never thought we'd see this kind of technology in our lifetimes.

Well, here we are. The arrival of ChatGPT, code AI assistants, and AI coding agents have changed how we all interact with computers, but especially for developers. With an LLM, we can have sophisticated, intellectual discussions, debate approaches, and solve complex problems through natural conversation. What used to be pure sci-fi is now everyday reality.

Steve spent decades being a tech skeptic and a late adopter, and Gene spent decades researching questionable claims of practices that supposedly improved software productivity. But the evidence changed our minds—evidence we'll share with you throughout this chapter.

Chat and agentic programming use LLMs to gain seemingly extraordinary capabilities. We're approaching a world where all you have to do is explain what you want, and your words become working software almost instantly. When something's not right, you don't spend hours debugging—you just describe what needs to change. Or the AI may identify and fix things for you automatically. There are times when your ideas spring to life, turning into working software almost as fast as you can articulate them.

Your AI buddy can help you decompose your grand vision into actionable tasks. For some of these tasks, you delegate to an agent that performs them independently. Some tasks you may choose to work by yourself, collab-

orating with AI through design and implementation. AI can help you every step of the way, as an implementer, advisor, fellow designer and architect, code reviewer, and pair programmer—if you let it.

When cocreating with your AI partner, it feels as though ideas shoot like lightning from your brain directly into the computer, magically transforming into running code. Like most people, you'll gasp with disbelief or delight at least once when AI does something far beyond what you expected, or when it solves a problem you've been struggling with for hours or days. And you can implement many more ideas, not just your best ones, because software creation is so fast now.

AI does far more than generate code. It's a true partner—one you can talk to like a person—that helps you brainstorm ideas, evaluate options, manage projects and teams, navigate challenges, and develop strategies to achieve your biggest goals and aspirations.

The Rise of Vibe Coding

As we mentioned in the Introduction, Dr. Andrej Karpathy stands among the most eminent AI researchers of our time. He helped create ChatGPT while at OpenAI and revolutionized computer vision systems for autonomous vehicles as director of AI at Tesla. His contributions to neural networks and machine learning have shaped our modern AI landscape.

In February 2025, Karpathy made an observation that perfectly captured the moment we're experiencing in software development: "There's a new kind of coding I call 'vibe coding,' where you fully give in to the vibes, embrace exponentials, and forget that the code even exists," he noted in a widely shared tweet that went viral across the tech world.[1]

He continued:

> *I just talk…I barely even touch the keyboard. I ask for the dumbest things like "decrease the padding on the sidebar by half" because I'm too lazy to find it. I "Accept All" always, I don't read the diffs anymore. When I get error messages, I just copy paste them in with no comment, usually that fixes it.*[2]

What's startling in Karpathy's admission is, "When the code grows beyond my usual comprehension, I'd have to really read through it for a while." Rather than diving deep into understanding, he troubleshoots by "asking for random changes until [bugs] go away." His process distills to, "I just see stuff, say stuff, run stuff, and copy paste stuff, and it mostly works"—a workflow that prioritizes results over traditional understanding.[3]

Almost overnight, the concept of vibe coding exploded, making its way into real-world developer culture. People across Twitter (X) embraced it as either a laughable meme or a legitimate practice. It was clear vibe coding was going viral, but was it going to become an established technique?

Within a few months, it had already become commonplace for real-world use. Garry Tan, CEO of Y Combinator, Silicon Valley's most famous startup incubator, said, "For 25% of the Winter 2025 batch, 95% of lines of code are LLM generated…The age of vibe coding is here."[4]

Boris Cherny, technical staff at Anthropic and technical lead for Claude Code, reports that he feels he is 2x as productive using coding agents,[5] while some others report feeling 10x more productive.

This increasing use of AI for development is not restricted to frontier AI labs and startups. Tobi Lutke, CEO of Shopify, the second-largest Canadian publicly traded company with $8.8 billion in annual revenue in 2024[6] and over four thousand developers,[7] said in an internal memo: "Before asking for more headcount and resources, teams must demonstrate why they cannot get what they want done using AI."[8]

The big question is whether companies using vibe coding are setting themselves up for problems down the road.

The Vibe Coding Debate

The AI world moves fast, but the vibe coding landscape and debate are moving even faster. Two sides of the discussion are emerging. On one side, we have people like Brendan Humphreys, the CTO of Canva, who has expressed serious concerns about the unrestricted use of AI-generated code in production environments. "No, you won't be vibe coding your way to production."[9] He argues that vibe coding—which he defines as when engineers prompt AI to

generate code with minimal human oversight—is incompatible with creating reliable, maintainable production software.

Similarly, Jessie Young, principal engineer at GitLab, said, "No vibe coding while I'm on call!"[10] When expressing her concern about vibe coding engineers who don't understand the code they're committing, and being the one who has to debug it in production at 2 a.m.

On the opposite end, we find people like Sergey Brin, Google cofounder, who has embraced a more radical approach. Brin has enthusiastically encouraged Google engineers to use AI tools aggressively, focusing less on coding details and more on product direction.[11]

As Brin suggested, "The role of the engineer will change more to being the product engineer, where they decide what the product should do," highlighting a fundamental shift from writing code to directing AI. Others embrace a new approach to debugging, where "instead of fixing code, you regenerate it" until it works.[12]

Despite their philosophical differences, these technology leaders agree on several important points. Both acknowledge that AI coding tools are reshaping the foundations of software development. Neither disputes that these tools can boost developer productivity. Both recognize that AI capabilities are advancing rapidly and that approaches must evolve with them. Karpathy, Humphreys, and Brin are all asking the same question: To what degree can you turn your brain off when you use AI to help you create software?

Vibe Coding for Grown-Ups

While YouTube influencers grab headlines by generating World War II flight simulators in a single prompt, we're focused on bringing vibe coding into professional software engineering. This requires applying disciplined engineering practices while still letting AI handle the tedious implementation details. In other words, *vibe coding for grown-ups*.

That means all the grown-up stuff that you may already be responsible for: security reviews, test coverage, blast radius management, and operational excellence. The difference is that you're doing this at speeds none of us have ever experienced before—you know, creating thousands (potentially tens of thousands) of lines of code per day.

When working on authentication for a customer-facing application, you'll still scrutinize every line of security code and build comprehensive test suites—but you can do it much faster. For legacy systems that nobody understands anymore, you might first use AI to analyze and document the code base, build tests to capture existing behavior, and only then begin making changes with confidence.*

This is about taking your hard-won engineering discipline and applying it with greater intensity. You're the head chef, and your role is setting standards, tasting rigorously, and ensuring every dish meets your standards, because, as the kitchen speeds up, the potential frequency and magnitude of mistakes goes way up too.

As Dr. Karpathy points out, these AI tools are improving exponentially. They're currently the least capable they'll ever be. With that in mind, we believe it's time to move beyond painstakingly crafting every line of code by hand and fully embrace this new approach to building software.

However, here's one thing we genuinely believe: No one should be writing code by hand anymore if they don't have to.

Substantiating the 10x Claim: Gene's Real-Life Example

Steve is an experienced professional engineer, having written over one million lines of production code in his career. Is it only people like him who can get the 10x gains and generate over a thousand lines of working code per day? How about a mediocre developer like me?

To explain why we believe the answer is decisively yes, I wanted to share this story. We were in the final process of editing this book, with less than seventy-two hours before we had to turn in our final manuscript to our editors. After that point, we'd have little or no ability to change the book. Steve was already nervous about whether we'd make our deadline. But despite that,

* Here's a great example of modifying legacy code: Microsoft researcher Jonathan Larson demonstrated using LLMs and GraphRAG to modify the 1993 id Software DOOM source code to enable player jumping. This was a nontrivial feat because the original engine does not have a true 3D internal model and was built on assumptions that the player was always grounded. The change modified many tightly coupled subsystems, including physics, player state, input handling, and level logic.[13]

I made what may seem like an insane decision: Invest precious time to build a productivity tool instead of reviewing, editing, and writing. Why? Because I was getting so frustrated at how tedious and error-prone it was copying and pasting portions of our manuscript into an LLM.

To make our book the best it could be, we were copying huge chunks of the manuscript into an LLM to do things like hunt for repeated ideas, ensure that every section was novel and new, get opinions on the optimal ordering of the Part 3 practices, and create good signposting (e.g., introductions, conclusions, etc.). But the breaking point for me was extracting all the chapter introductions to compare them to each other. My hands and wrists already hurt from all the typing and trackpad operations, and I couldn't imagine doing that by hand as well. There had to be a better way.

For months, I wanted to query the book manuscript like a SQL database and retrieve subsections with a single command. With a tool like that, I'd be able to magically extract text directly into my clipboard and ask: "Give me the outline of the whole book." "How about just this chapter?" "Copy the text from Parts 1 and 2." "How about just Chapter 4?" "How about just the first three sections?"

At 4 p.m. on the Saturday before our deadline, after we took a break from one of our marathon editing sessions, I opened up a Markdown parser I had written in 2022 to do book modification visualizations. Maybe it could serve as a good starting point for this "Markdown as database" tool. The trouble was, I couldn't remember how any of it worked. So, I used Claude Code to help me.

I typed out, "I think there's code in here that parses .md files and turns it into a hierarchical tree. I'm trying to build something that can take that tree and perform operations like 'list all chapters' or 'for a given chapter, list all sections or get all text in the children.'" Fifty-two minutes later, I had all of those functions mostly working.

Over the next four days, during breaks from working with Steve to finish the book, I wrote 4,176 lines of Clojure code across 52 files (2,331 of production code and 1,845 lines of tests), along with over 3,000 lines of documentation and reports. To ensure confidence that the text extraction worked perfectly and didn't introduce errors, the test suite had increased by nearly 6x.

My years-long aspiration of turning a Markdown file into a queryable database had been achieved, and, more importantly, I was no longer selecting text in Google Docs by hand. It was truly FAAFO.

Analyzing the complete Git history in this repo by using vibe coding, I was comfortably 10x faster than I could have ever been without AI. Specifically, I was 16x faster than my historical average and 5x faster than my previous best day. And I did it in the middle of our marathon writing sessions: during breaks, after we adjourned for the day, while I brushed my teeth, etc. The whole endeavor required 251 prompts across 35 commits.

This investment paid off. Slinging book text around previously took minutes and was prone to errors, but now it happened with a keystroke, all because the book manuscript could be queried like a database. I'm proud that I built this tool, and I truly believe it helped make this book better.

Here's a summary of things I built:

- Instant content extraction without manual scrolling through hundreds of pages across multiple Google Docs using array slicing syntax (à la Ruby, Perl): "Parts [1…3]," "Parts [1,3,4]," "Chapter [1,20]," or "Sections 2 and 3[1…3]."
- Generate the complete outline of any set of parts, chapters, or sections.
- Chapter intro/conclusion extraction: Get any of the text above, but exclude the introductory and concluding sections, so that we can balance them.

I haven't even mentioned the crazy race condition I stumbled into, and how Claude Code created a reproducible test case by running a hundred threads in parallel and generating a workaround.*

This was a record amount of work for me in such a short time. Afterward, Steve asked me a question that left me dumbstruck: "Did it feel like writing four thousand lines of code?" I told him I didn't even count the lines of code

* You can read a longer description of this whole adventure in the blog post "The Last 80 Hours Of Editing the 'Vibe Coding' Book (and Vibe Coding 4,176 Lines of Code On The Side) — Part 1: The Stats and All The Prompts" at ITRevolution.com.

until I wrote this story. It just felt like I was building the capabilities I needed at a magical pace. Code just flowed like water.

You'll hear us make the 10x productivity gain claim in the book. This story isn't the only substantiation we have; we share other stories and research later on. We believe we can stand behind this number with confidence.

You're Head Chef, Not a Line Cook

In the old days as a solitary developer, implementing a simple visualization dashboard could require any number of tedious steps: hours researching charting libraries, reading all the documentation, figuring out the configuration options, parsing data files, handling functions to throw out bad data, and implementing user interactions. Then you slowly type out code, perhaps copying and pasting code you find on the internet. When stuff goes wrong, you debug by looking at log statements and maybe stepping through with a debugger.

Yuck! How did we do this for so long?

With vibe coding, you say: "Here's some input data. Create a chart with years on the *x*-axis." Within seconds, you'll see your chart. Then you guide your AI assistant toward what you want (e.g., "Make the *y*-axis logarithmic." "Use a stacked bar chart instead.").

In this new world, you're the head chef of a world-class kitchen. As such, you don't personally dice every vegetable, sear every steak, swish away every cockroach, or plate every dish. You have sous chefs and line chefs for that. But when a meal leaves the kitchen, it's *your* reputation on the line and your Michelin stars at stake. When the customer sends back the fish because it's overdone or the sauce is broken, you can't blame your sous chef.

The same principle applies when coding with AI: Delegation of implementation doesn't mean delegation of responsibility. Your users, colleagues, and leadership don't (or shouldn't) care which parts were written by AI—they rightfully expect you to stand behind every line of code. When something breaks in production at 2 a.m., no one wants to hear, "Well, AI wrote that part." You own the final result, period. This is both liberating and challenging. When vibe coding, you'll:

- Spend more time thinking about what you want to build and less time on implementation details. (Which is nice.)
- Develop a critical eye for evaluating AI-generated solutions, rather than crafting every line yourself. (Some may miss the coding part, though.)
- Learn to communicate your requirements to a non-human collaborator. (This can have a real learning curve.)
- Take responsibility for the final product while delegating much of the implementation work. (This should already be a familiar, perhaps unnerving feeling to many of you who have been in technical leadership roles. You'll find it's not so different with AI helpers.)

The Broader Responsibilities of a Head Chef

Coding is to home cooking what vibe coding is to running a professional kitchen. When you don your head chef's hat and start using coding agents, like us, you'll notice a bunch of strange things start happening.

For over a decade, we (like most developers) have used version control systems like a glorified save button—save, undo, restore, maybe occasionally branching now and then. We mostly wrote commit messages like "fix something dumb" and pushed straight to the trunk of the code base and would rewind to an older revision if we messed something up.

But since we've started using coding agents, we regularly find ourselves smack in the middle of operations that we've previously only seen handled by release engineers and version control virtuosos. Since we both use Git, we find ourselves cherry-picking commits, merging selective changes across three or more branches, and doing complex rebases. Plus, more—way more.

We're using Git features that we barely know the names of, and we're doing it a *lot*. But it's not about Git. This would be happening no matter what version control system we used. We started scratching our heads over why we were doing all this complicated Git stuff every day. Was it nothing but a distraction? We soon realized that it was yet more evidence that vibe coding turns an individual into a team. We had both been using *team*-related Git commands that you usually only use in multi-contributor projects.

It's one thing to think of your kitchen of sous chefs as individual helpers. But no chef is an island: Teams require coordination in ways that individuals don't. With vibe coding, you'll be responsible for:

- **Managing parallel development:** Running multiple agents working on different tasks simultaneously, with time spans ranging from minutes to weeks—the opposite of the traditional "single-threaded" developer approach.
- **Handling complex integration:** Merging work from different branches and resolving the inevitable conflicts that arise when multiple agents modify related code.
- **Setting standards:** Defining explicit coding standards and processes so your AI team operates consistently and efficiently.
- **Creating onboarding procedures:** Setting up workspaces, access, and instructions for each new AI assistant you bring into your system.
- **Coordinating larger projects:** Taking on more ambitious work than ever before, requiring you to think like a project manager.

This team stuff is all new for most solo developers, and doing it with AI agents is new for everyone. But make no mistake: There is no opt-out for this "promotion" to head chef—it's inherent to vibe coding, which is how all software will soon be developed.

For better or worse, from now on, anyone developing software who goes head-to-head against a well-managed team of AI agents without a team of their own will nearly always lose. No matter how good you are at football, if you take on an NFL team alone, you will lose (unless perhaps it's Detroit). And this competitive mismatch (outside Michigan) will drive everyone, including you, to adopt teams of AI agents.

That makes you a team leader. Unless you still prefer to write code by hand (like a savage), you're now officially promoted to head chef. We'll talk a lot more about the importance of coordination in Part 4, both for individuals and for leaders.

You may still think AI only speeds up your solo work. That was true in 2024, but with the emergence of coding agents, a broader picture is beginning to unfold. Up until now, using AI has accelerated *you*. But now your role is to accelerate *them*.

So, get ready, head chefs. We're entering a brand-new world, for sure.

Conclusion

Whether you choose to embrace it or fight it, every modern software project could turn into a conversation between a human and an army of AI agents that can turn vision into reality at blistering speed.

We believe this changes the shape of your job. You're no longer typing lines of JavaScript. The job is now deciding what delicious dish you want your team to prepare, tasting the results early and often, and orchestrating your automated helpers so nothing leaves your kitchen that you're not proud of. Do that well and you unlock the full FAAFO menu: You'll ship faster, chase more ambitious ideas, operate more autonomously when you need to, rediscover the fun that got you into coding in the first place, and keep optionality on the table for every design decision.

None of that happens by accident. A head chef writes down the house rules, checks every plate before it hits the dining room, and sends the occasional dish back when it sucks. Likewise, you'll need clear standards, ruthless validation loops, and the courage to regenerate code instead of patching lukewarm leftovers. This is vibe coding for grown-ups—equal parts creativity and discipline.

In the next chapter, we'll explore why these AI breakthroughs represent something genuinely novel and badly needed by developers, despite the last seventy years of advances in technology.

CHAPTER 2

PROGRAMMING: NO WINNERS, ONLY SURVIVORS

Vibe coding fundamentally changes how we create software—and in a way that is different from all the changes that have come before. Over seven decades, how humans write software has transformed in significant steps, each elevating developer productivity. But developers still struggle with many core problems.

In this chapter, we'll explore how life has improved for people writing software over the last seventy years, but highlight how ridiculously difficult writing software still is. The result is that developers are miserable, and many choose to stop coding because it has just become too hard. All that is changing now, as vibe coding allows us to rocket up the abstraction layer, liberating us from details that don't matter: libraries, frameworks, syntax, builders, minifiers, and more.

You'll also hear a tale from Steve about how he learned to draw polygons and shaders in college, which no one cares about anymore. These days, kids with no training can make professional-grade games or mods, complete with custom physics, animation, and combat systems. This is a microcosm of the exponential growth happening right now with the advent of AI and vibe coding.

The Major Programming Technology Advances Up Until Now

Programming languages evolved to let us express ideas more naturally, focusing on high-level problems rather than computer internals. Development

environments transformed from punch cards and teletypes to rich IDEs that catch errors in real-time. And access to knowledge exploded, with resources like Google, Stack Overflow, and GitHub shrinking the learning cycle from months to days. These revolutions in languages, tools, and knowledge greatly increased our capabilities. Writing software today should be easier than in decades past.

And yet, the reality is that building things has been getting steadily harder. Systems keep ballooning in size and complexity. Debugging and testing are still painful. We bang our heads against constant roadblocks. The simplest of today's tasks require mastering an overwhelming array of rapidly changing tools and technologies.

To do anything, we often feel like we have to know everything about everything, all while everything is changing. As one example, at the time of this writing it's fashionable to ridicule the complexity of JavaScript development. Let's peek at why.* To build a web app, you might need to understand this daunting list (which is probably already outdated):

- package managers (npm, Yarn)
- bundlers (webpack, Rollup)
- transpilers (Babel)
- task runners (gulp, Grunt)
- testing frameworks
- CSS preprocessors
- build toolchains
- deployment pipelines

And that's before so much as glancing at modern JavaScript language features. Each of these components has many available contenders. Some depend on each other, some conflict, and it's almost impossible to navigate the graph of what works with what unless you live and breathe that ecosystem every day.

It keeps going. Because of the DevOps philosophy of "you build it, you run it," you also need to learn Docker, Kubernetes, AWS, and infrastructure-

* A great example is Jose Aguinaga's "How it feels to learn JavaScript in 2016."[1]

as-code tools like Terraform, not to mention a whole host of AWS, GCP, or Azure services. If you're especially cursed and your company is multi-cloud, you might have to learn two or more clouds.

Thanks to these "advancements," you can now find yourself simultaneously worrying about how to center a `div` element on a web page, while you struggle with Docker networking issues because your CI pipeline broke after you tried to change to Terraform scripts.[2]

Our point is this: We find it deeply ironic that despite all the revolutionary transformations of software development over the past decades, we're still mired in more complexity than ever. And incidentally, this is why many people have chosen to leave coding—it has become too freaking difficult and not worth the effort. There are days when it doesn't feel like all these advancements have improved life much, and that building things has been getting steadily harder.

There Is Now a Better Way

We moved from punch cards to IDEs, and from books and searches to Stack Overflow. Now, instead of writing code by hand, we have a conversation with AI about what we want to build. If you want to create a web application, rather than wrestling with package managers, bundlers, and deployment pipelines, you describe what you want in plain English: "Write me a web app that lets me chat privately with only my friends."

If all goes well, your AI collaborator will help you build it the way you want it. You'll work with it to ensure it chooses appropriate libraries, generates test suites, follows good practice, makes the code secure and fast, and so forth. If software development were moviemaking, we're no longer script writers; we're now the directors, guiding the vision while our AI collaborators handle the implementation details.

Although we find vibe coding to be far better than the old way (because of FAAFO benefits), that doesn't mean vibe coding is *easy*. On the contrary, your judgment and experience are now more important than ever. AI can be wrong, sometimes wildly so. That's where you come in. Programming with AI is a lot like traditional programming, and most of what you know still

matters. But this better way of creating software also requires building new instincts about what's happening with the LLM and your code.

Think about it this way: What works for driving safely at 10 mph becomes insufficient when you're traveling 10x faster. The leisurely pace of manual coding gives you time to spot problems, think through edge cases, and course-correct gradually. But when your AI partner can generate modules in seconds, you need new mental models and skills. Without them, you'll almost certainly wreck the car spectacularly. (We'll share with you our own memorable crash stories later in the book.)

The good news: As Astronaut Frank Borman once said, "Superior pilots use their superior judgment to avoid situations which require the use of their superior skill."[3] Your experienced judgment will become perhaps your most valuable skill of all in the new world of AI, because it will help you avoid needing to use your disaster recovery skills.

War Story: Steve Studies Computer Graphics in the 1990s

What sounds more fun: Developing a Skyrim game mod or rendering a shaded polygon? The transformation programming is enduring, reminding me of how fast the world of computer graphics changed in the 1990s. Jobs were upended, and university courses had to be rewritten from scratch almost every year. Nothing had changed so fast before, and it was bedlam.

But it also boomed, creating new categories of jobs, specialists in everything from water physics to motion capture. And over time, graphics development has been adopted by less technical people. You can make remarkable game mods today without needing to know much about the underlying technology stack that powers them.

To put it in perspective, in the early 1990s, I took the University of Washington Computer Graphics course, taught by industry legend and entertaining lecturer Dr. Tony DeRose, who currently leads Pixar's Research Group. On the first day of class, he warned us that we could only use one API call: `putPixel(r, g, b, a)`. Using that lone function, we had to build up our little 3D worlds one pixel at a time.

That was the state of the art circa 1992. We would wait hours for our projects to render on the lab computers, simple static scenes of teapots and chess pieces. Occasionally, a student would wait eight hours only to see their render come out mangled, and they'd run from the lab wailing in despair.

Three years later in 1995, graphics had become a different course. No more `putPixel()` calls. All that rendering stuff was now handled in hardware. Instead, you were working with higher-level abstractions: lighting, object scenes, and animation. There were different mental models, different tools, different jargon. In a short time, graphics had been elevated into a new discipline from the one I had learned.

And our productivity was off the charts. No more teapots—you could develop a full movie in the lab. People would still run out wailing when it didn't work in the morning—but it was because of physics engine and hitbox problems, not polygon rendering.

As for the job market, the software industry's graphics jobs kept pace with the breakthroughs. Over the next thirty years, graphics roles continued pushing far up the abstraction ladder and have branched out into a *huge* number of distinct specializations.

The graphics revolution is still going strong today. High-school students now take weeklong courses in game development using game engines like Unity, where they never see a single line of graphics code. Instead of wrestling with polygon math and pixel operations, they spend their time doing fun stuff like modeling objects and building game maps, while Unity's physics engine handles the rendering complexity underlying it all.

I am fascinated to this day by how the daily work as a graphics programmer has evolved, to where the title "graphics programmer" is almost unrecognizable from the early days. But as stunning and exciting as that transformation was, it doesn't hold a candle to what is happening with coding and AI.

Conclusion

Computer graphics evolved from a black art requiring PhD-level math in the 1990s to something any motivated teenager can master with Unity or Unreal

Engine. Now AI is performing the same magic trick across all of programming, and it's happening at warp speed compared to the graphics revolution. The jobs and work changed and evolved as the technology advanced. We can expect the same to happen with AI.

Graphics became more fun when developers could focus on building worlds rather than calculating vertex normals. Programming becomes more enjoyable when you're building cool things rather than debugging semicolons. Some will mourn the loss of certain technical challenges (we still meet graphics engineers nostalgic for texture mapping in assembly), but most will celebrate when they realize what's possible.

What happened in the computer graphics industry is happening everywhere in software. Vibe coding is enabling us to create cool things, liberating us from a gazillion things that don't matter. How very FAAFO!

CHAPTER 3

THE VALUE VIBE CODING BRINGS

Sure, vibe coding makes you code faster—that's the obvious selling point. But if you think speed is the whole story, you're missing out on the juicy stuff. We've discovered that vibe coding creates value across five dimensions, which we've named FAAFO—fast, ambitious, autonomous, fun, and optionality.* We explored them briefly in the Introduction, but we'll go into more detail in this chapter.

Think of FAAFO as your new superpowers. You're coding faster, and you're now bold enough to risk projects you'd have laughed off as impossible before. You're working solo on stuff that used to require teams. And because you're lowering the cost of coordination, and the "people can't read my mind" tax inherent in any collaboration, you and your team can work more autonomously. You're having fun again, like when you first learned to code. And most powerful of all, you're exploring multiple solutions simultaneously, picking the best option instead of committing to the first idea that seems workable.

Write Code *Faster*

While speed is a clear value of vibe coding, it's arguably one of the most superficial benefits. It's impressive, but we've had a lot of speedups before. The main value of going faster is the extent to which it multiplies the value in the other dimensions of FAAFO.

* By the way, you may have noticed that there is no "B" in FAAFO. Vibe coding does not automatically make your code better. That is your responsibility. By following the techniques and practices we present in this book, you'll have the best chance of success at making your code better and becoming a better developer, in addition to the other FAAFO benefits.

Consider the video excerpt tool that Steve helped Gene create (as we mentioned in the Introduction), which generated clips from podcasts and videos. They built the first working version in forty-seven minutes of pair programming using only chat coding, no agentic AI assistance. That's pretty fast. Gene estimated that it would have taken them two to three days to write it by hand.*

The key lesson we learned during that session: Type less, lean on AI more.

But we also found that sometimes AI can make things maddeningly slower and more frustrating. We've each experienced this firsthand. Gene spent hours going in circles with AI trying to get `ffmpeg` to properly position captions and images in video files. Steve wasted an afternoon wrestling with an AI collaborator that confidently insisted on different approaches, all of them wrong, to parsing command-line arguments in Gradle build scripts.

It can take both vigilance and good judgment to recognize when you're being led down a rabbit hole and need to change course. Vibe coders must learn to notice when AI is heading confidently down a wrong path and decide when to redirect or abandon unproductive approaches.

Despite these occasional challenges, we still love it. And when vibe coding isn't possible (e.g., no internet connection or local LLM), many developers like us now choose not to code at all. Old-style coding by hand seems pointless. It's like needing to get down a seventy-mile desert road, but you won't have a car for a couple of hours. It's less work to wait for the car to come get you, as opposed to walking part of the way. It's not worth the bother.

Who wants to write code by hand like some relic from 2010? Not us.

Be More *Ambitious*

Recall Gene's first working version of the video excerpt tool, which previously would have taken days. Because of the time and effort required, he had originally deferred trying. This happens in organizations too. There could be many reasons why projects are never started: Perhaps the perceived benefit

* Many of you reading this may want to point out that developers typically spend only about 25% of their time writing code and twice as much time reading code. We'll address this later in the book, as well as how AI can help with many activities beyond writing code.

wasn't high enough to warrant the work, or maybe the difficulty made the payoff not worth the investment, or possibly another opportunity offered a higher, more immediate return.

With vibe coding, Gene was able to complete work that otherwise would never have been undertaken. Projects that once seemed too difficult or time-consuming become feasible, opening new possibilities for what can be accomplished. Vibe coding reshapes the spectrum of what can be built, letting you be more ambitious.

Seemingly impossible projects move into the realm of possibility. Applications that would have required specialist knowledge across multiple domains can now be built by developers with AI assistance filling their knowledge gaps. Five-month projects become five-week projects, or sometimes five days. Ideas once considered too ambitious get tossed onto your to-do list without a care in the world.

Small-ish, low-return jobs become quick wins, because it can be easier to do the work than to create the task. Documentation, tests, minor UI improvements, and small refactorings that were perpetually pushed aside can now take seconds or minutes instead of hours or days. These tasks get done, rather than accumulating in ever-growing "broken windows syndrome" backlogs. You can fix every window in town and keep them fixed for once.

As Cat Wu, product manager of Anthropic's Claude Code team, observed: "Sometimes customer support will post 'Hey, this app has this bug' and then 10 minutes later one of the engineers will be like 'Claude Code made a fix for it.' Without Claude Code, I probably wouldn't have done that…It would have just ended up in this long backlog."[1] There has always been a category of work where it was easier to fix than to record and prioritize. That category is bigger now with AI.

This expanded capability leads directly to our next important dimension of value.

Be More *Autonomous*

In June 2024, Sourcegraph's then-Head of AI, Rishabh Mehrotra, showed Steve a demo of a multi-class prediction model he had created—from concept

to deployment—in half a day using vibe coding. He told Steve it would have been a whole summer intern project, or perhaps six weeks for a superstar intern, as recently as a year prior. Rishabh was shocked that he had completed it alone in a few hours.

Rishabh had only discovered it was easy because he didn't have the budget to hire an intern. So, in desperation, he figured he'd try it alone with AI. He finished so fast he—an AI expert—was flabbergasted.

This illustrates the third dimension of value that vibe coding enables. Developers (and teams) can accomplish tasks autonomously (and in some cases, alone) that otherwise would have required help from other developers or sometimes teams. Working with multiple people introduces significant challenges—communication and coordination, competing priorities, merging work—and the more people involved, the less time you spend solving the problem.*

Working autonomously frees you to do the work you need to do, enabling independence of action. (This is a term we'll use throughout the book.) Steve experienced this firsthand as a leader of one of Amazon's first "2-pizza teams" created to reduce customer contacts per order. The mandate was simple: Give small, cross-functional teams complete ownership of their problem space with full capability to deploy solutions without navigating layers of dependencies and approvals. If reducing customer contacts means changing the checkout flow, rewriting the help system, or building new infrastructure, the team could do it all. No waiting for the UX team's roadmap. No negotiating with the infrastructure team's priorities. No endless meetings to align seventeen different stakeholders.

This radical autonomy and independence of action transformed how fast Amazon could move from identifying problems to shipping solutions. Now, with AI as your tireless collaborator, you can achieve this same independence of action as an individual developer.

Beyond eliminating organizational friction, AI also helps solve an equally difficult problem: the "mind reading" tax inherent in collaboration. Let's face

* Some people may recognize this as Brooks's Law, coined by Dr. Fred Brooks, author of *The Mythical Man-Month*, who observed that adding manpower to a late software project makes it later, due to the increased communication overhead and coordination complexity. This is because the number of communication lines increases exponentially as team size grows—rising from three lines with three people to forty-five lines with ten people.

it—no matter how skilled our teammates are, something inevitably gets lost when we try to convey what's in our heads. When vibe coding autonomously, this universal challenge becomes less of a problem. You can implement what you envision because there's no gap between your idea and its execution. You know it's right when you see it because it matches the picture in your head.

The consequences of these two taxes show up across every domain where experts and novices collaborate. For fifteen years, Dr. Matt Beane studied this phenomenon, with surgical robotics providing a compelling example. Traditionally, junior surgeons learned by necessity—procedures required three or more hands, making their participation essential while creating natural apprenticeship moments. However, when surgical robots enabled senior surgeons to operate independently, these teaching opportunities disappeared despite training remaining an official responsibility.

The senior surgeons, given the choice, overwhelmingly chose to work alone. This wasn't because they didn't value teaching; it was because coordination costs are often higher than we acknowledge.* Every explanation, every correction, every moment spent bringing someone else up to speed represents time not spent on the primary task. When the surgical robots removed the physical necessity of assistants, the true cost of coordination became visible through the seniors' behavior.

This same pattern appears in software development. If it's possible to create things without external dependencies, without any need to communicate and coordinate with others to get what we need, the advantages multiply rapidly. The constant back-and-forth of explaining requirements, correcting misunderstandings, and reconciling different mental models disappears.

Economist Dr. Daniel Rock (famous for his work on the "OpenAI Jobs Report") calls this "the Drift,"[3] borrowing from the movie *Pacific Rim,* where two pilots mentally connect to operate giant mechs. When you and your team vibe code, you can create that kind of mind-meld with AI assistants, reducing the coordination costs that typically slow down multi-human teams.

With "the Drift" active, a product owner can directly work with the code base through AI rather than writing a detailed products requirement docu-

* Indeed, this is one of Gene's biggest learnings working with Dr. Steven Spear over the last four years. As they state in their book *Wiring the Winning Organization*: "Leaders massively underestimate the difficulty of synchronizing disparate functional specialties toward a common purpose."[2]

ment (PRD). A developer can evolve the database schema without a database specialist. As Dr. Rock demonstrated with his three-person team that built a GitHub app in forty-eight hours, this shared mental model accelerates development in ways that traditional human-to-human coordination cannot match. Being autonomous with AI means being unblocked—free to move at your own pace without constant negotiation and handoffs.

Scott Belsky, Chief Product Officer at Adobe, describes this as "collapsing the stack,"[4] illustrating the benefits of the same person owning more of the process. When that happens, they not only generate better results, but it's also more fun. Which leads to our next dimension of value…

Have More *Fun*

While writing code faster, tackling more ambitious projects, and eliminating coordination costs are fantastic benefits, vibe coding delivers another fundamental transformation that shouldn't be underestimated: programming becomes more fun.

Traditional programming involves many tedious tasks that few developers enjoy. Fixing syntax and type checking errors, wrestling with unfamiliar package managers, writing boilerplate code, searching for documentation, and so on. Vibe coding eliminates these pain points, shifting focus from implementation details to *building* things.

A randomized controlled trial of GenAI coding tools found that 84% of developers reported positive changes in their daily work practices after using AI tools. They reported being more excited to code than ever before, feeling less stressed, and even enjoying writing documentation.[5]

At Adidas, where seven hundred developers now use GitHub Copilot daily, 91% of developers reported that they wouldn't want to work without it. Fernando Cornago, SVP of Digital Technology at Adidas, described how vibe coding resulted in developers spending 50% more time in what they called "Happy Time," productive time when they were mastering their craft. This is the opposite of "Annoying Time," such as struggling with brittle tests and meetings.[6] (We cover more of this story in Part 4.)

Building cool things is addictive. Vibe coding, especially with agents, turns your keyboard into a slot machine. You "pull the lever," and out comes a payout—a chunk of working code, a generated test, or a refactoring. Each little payout delivers a tiny dopamine hit, a neurochemical reward that makes us feel good and encourages us to pull the lever again.

It's fun and pulls you in. We've both found ourselves so thrilled and engrossed by what we're creating that time melts away. It's driven by that exhilarating "Let's just do one more thing!" feeling, and the sheer fun of seeing ideas take shape. But unlike the tedious all-nighters of traditional debugging sessions, these jam sessions are pure creation. But perhaps the most powerful benefit of all is yet to come: Vibe coding increases your ability to explore options and mitigate risks before committing to decisions.

Explore More *Options*

The fifth dimension of value that vibe coding creates may be its most profound: expanding your ability to explore multiple options before committing to decisions. In traditional development, choosing a technology stack often means making nearly irreversible commitments with limited information. These architectural decisions became what Amazon called "one-way doors"—once you walk through, turning back becomes almost impossible (or inconveniently expensive).

Vibe coding reduces the cost of exploring multiple paths in parallel. You can experience this firsthand while building a project in your preferred language. During a forty-five-minute walk with your dog, you can have a voice conversation with an AI assistant that thoroughly evaluates your options for complex libraries or frameworks. What might usually require days of research is compressed into minutes, providing detailed insights into each option's trade-offs without writing a single line of code.

This is a capability that we never had before as programmers: The luxury of trying something five or ten different ways at once for practically free. And it extends beyond research to implementation. You can prototype the same API using three different architectural patterns in a single afternoon—say,

RESTful, GraphQL, and gRPC. You can implement core endpoints using each approach, complete with serialization, error handling, and client integration. What previously might have required weeks of effort for a single implementation can now be comparatively evaluated through hands-on experience with all three options.

This concept of optionality was formalized in finance theory in the 1970s: An option is defined as the right, but not the obligation, to make a future decision. This concept is powerful in software development because software begins as pure thought—it's infinitely malleable until deployment creates real-world constraints. Every architectural choice, every library selection, every design pattern traditionally forced us to pay the full cost up front without knowing whether we'd chosen correctly.

The higher the uncertainty, and the higher the risk/reward ratio, the more valuable options are. If there is no uncertainty, we don't need options—we pick the best choice, certain that our answer is correct. However, when things are highly uncertain (such as in the AI field right now), options become extremely valuable. (Another corollary: In times of high uncertainty, avoid making long-term decisions, which deprive you of options.)

Vibe coding changes the economics of software creation: Instead of betting everything on our first guess, we can place small bets across many possibilities and double down only on what works.

Toyota discovered how significant option value was decades ago in manufacturing. While American manufacturers focused on standardization and rigidity, Toyota built systems that enabled flexibility and adaptation. Their modular production lines, frequent experimentation, and rapid feedback cycles (including four thousand daily Andon cord pulls stopping production) created an option-rich system.

They could manufacture multiple model years simultaneously on the same production line, implement dozens of production changes daily, and exploit option value in many other ways that created a durable, lasting competitive advantage. Seventy years later, automakers around the world are still copying this strategy.

It's almost impossible to overstate the value that optionality creates. Over two hours, the two of us were tutored by one of the premier economics scholars, Dr. Carliss Baldwin, William L. White Professor of Business

Administration, Emerita at Harvard Business School.* She has written extensively about how the ability to parallelize experimentation, enabled by modularity, creates so much surplus value that it can blow companies and industries apart.

This explains how Amazon's microservices rearchitecture in the early 2000s (which Steve was a part of) allowed them to rapidly experiment with new business models, eventually spinning AWS into a more than $100 billion business that competitors couldn't match because their architecture prevented exploration.

AI can drive down the cost of change,† and can decrease the time and cost to explore options. That is, if you have a modular architecture that enables it. We'll explain how to create this later in the book. Organizations that take advantage of creating option value will be orders of magnitude more competitive than those that don't. (We explore this in more detail in Parts 3 and 4.)

AI as Your Ultimate Concierge

As a head chef running a world-class restaurant, you'll run into many problems that aren't strictly culinary. As it happens, however, your sous chef is also a sommelier, detective, accountant, rat catcher, master plumber, award-winning author, and tax planner. Remarkably, it's also a surgeon, taxidermist, and a lawyer. We think of AI as a concierge who is available to you 24/7, literally on a moment's notice, happy to take a phone call with any of your questions or whims.

Your AI collaborator is more than a code generator. It can help you with your toughest problems. Sometimes, it's your personal detective that you send to root through labyrinthine Git histories. You only need say, "I lost some test files somewhere between commit 200 and commit 100," and not only will it find it ("Found it. It was 43 commits back.") but it will track them down and

* Her advisor, Dr. Robert Merton, worked with Drs. Fischer Black and Myron Scholes on their work on options pricing, which earned them the Nobel Prize in Economic Sciences in 1997.

† The topic of reducing the cost of change is described through an economic lens in the spectacular book *Tidy First?: A Personal Exercise in Empirical Software Design* by Kent Beck.

stitch them back into your code. ("I extracted out the tests, and also the build configuration that refers to them.")

We've handed AI enormous, nested structure dumps and said, "Find that one little detail buried ten layers deep," and it came back in seconds with: ("It's ['server']['cluster']['node_13']['overrides']['sandbox']['temporary']").

We also love using AI as a design partner—a quick collaborator who's awake at any hour you're inspired to work. It's the extra pair of hands that can validate your ideas or debug that sneaky performance glitch you've been chasing for days.

In future chapters, we'll mention a few of the many kinds of messes that AIs can produce—or more accurately, messes that you produce using AI. It turns out your AI concierge is great for helping you get out of those messes as well, as long as you use the disciplined approach of only tackling small tasks at a time and tracking your progress carefully (which we cover in a future chapter).

Conclusion

We've seen how vibe coding rapidly accelerates your workflow, turning multi-day chores into lunchtime wins—like Gene and Steve hacking together the video excerpt tool in less time than it takes to cook a decent chili. Sure, sometimes your AI sous chefs misinterpret recipes (looking at you, captioning nightmare with `ffmpeg`), and you'll occasionally need to step in yourself, but the net result is still far quicker than manual coding.

However, as we showed you, speed is the least interesting part. Vibe coding creates value along five distinct dimensions or FAAFO: fast, ambitious, autonomous, fun, and optionality.

- **Fast feedback loops and high velocity make more projects feasible:** AI's speed enables all the other dimensions of FAAFO.
- **Ambition reshapes your project landscape:** "Not quite worth it" tasks become quick wins, and impossible dreams land on your to-do list.

- **Autonomy eliminates friction:** Work at your own pace without constant negotiation, handoffs, and the coordination costs that slow traditional teams.
- **Fun drives engagement:** Programming becomes addictive again when you're building rather than debugging, creating rather than wrestling with syntax.
- **Options create competitive advantage:** Explore multiple approaches in parallel, turning one-way doors into reversible experiments.

In the next chapter, we'll show some of the risks of vibe coding and what you can do to mitigate them.

CHAPTER 4

THE DARK SIDE: WHEN VIBE CODING GOES HORRIBLY WRONG

We've explored the FAAFO upsides of vibe coding. But like any new technology, AI-assisted coding has a dark side. Your AI sous chef may be your most helpful collaborator, but if you're not paying attention, it can also have breathtaking destructive potential.

A similar pattern occurred during the introduction of electricity into manufacturing. While electricity's tremendous potential was obvious, it wasn't until twenty years after its invention that factory owners learned to abandon their linear, belt-driven layouts in favor of designs that exploited electric power's flexibility.

Today's AI-coding revolution follows a comparable pattern—we can see the tremendous potential, but we're still learning how to harness it without triggering failures that can destroy months of work in minutes, wipe out code bases, or damage physical hardware.

Looking at the history of software, we can see plenty of reasons for hope. Like Sir Tony Hoare's* allowing memory pointers to be null—his famous "billion-dollar mistake"—or manual memory management in C that enabled decades of buffer overflows and security breaches, we eventually created technologies to mitigate the worst of these issues.

AI coding can introduce systemic risks that can cascade across development ecosystems. The stakes could be higher and the failures more spectacular than anything we've encountered in traditional software development. But we believe the principles and practices that have improved our software

* Also known as C. A. R. Hoare, Sir Hoare invented Quicksort and ALGOL (the progenitor of almost every programming language, such as C, Smalltalk, Java, etc.). He also created CSP (communicating sequential processes), which the Go concurrent model is modeled after.

practices for the last many decades can be modified to avoid potential pitfalls. The following are real-world stories of vibe coding gone terribly wrong. Let our hard-won lessons be your ticket to success.

Five Cautionary Tales from the Kitchen

The Vanishing Tests: Where's My Code?

Steve had a scary experience within two weeks of starting to use coding agents. After he had begun converting the automated test suite for Wyvern with an agent, he was appalled to learn from his colleague that the coding agent had silently disabled or hacked the tests to make them work and had outright deleted 80% of the test cases in one large suite.

Worse, by the time Steve found out, those tests had been deleted scores of commits ago. Many productive changes on the branch were layered in, so a rollback would not be straightforward. Steve was in a dilemma. That night, he texted Gene, "I told Claude Code to take care of my tests, and it sure did. It cared for them like Godzilla cared for Tokyo."

Steve's AI assistant never mentioned deleting these tests, nor did it ask for permission—it removed them silently. We describe what and why things like this can happen in Part 2, and what you can do about it in Part 3.

The Eldritch Horror Code Base: When FAAFO Dies

To support writing this book (and while writing this book), Gene built three generations of a writer's workbench tool. The goal was to reduce the immense amount of manual "slinging" of prompts and portions of the manuscript, which had to be copied and pasted into and out of different tools. His workbench tool started as a Google Docs Add-on. The third iteration was a terminal application, which underwent frequent evolution as he and Steve used it intensely during the book authoring and editing process.

All was going well. Gene had been using it daily, all day long, eventually having processed over twenty million tokens. It was super easy to keep adding functionality to the workbench…right up until it wasn't. The code base became what Gene described as an "eldritch horror"—a giant, three-

thousand-line function with no modular boundaries, impossible to understand or modify without breaking something else.

"I couldn't understand the function that the AI wrote to save the intermediate working files," Gene recalls. "It took me twenty minutes to understand the three arguments the function used, and I couldn't remember them ten minutes later." Gene spent three exhausting days rewriting and modularizing the code (with AI's help) and shoring up the tests to verify the correctness of the functionality they were relying on every day.

This finally brought FAAFO back from the cosmic abyss, and this tool helped Gene and Steve deliver the first draft to the editors, 50 million tokens later. We'll describe the techniques used in Part 3, where we discuss how to prevent, detect, and correct these types of problems.

The Vanishing Repository: Near-Catastrophic Data Loss

Perhaps the most alarming story comes from Steve, who one day noticed that his Wyvern TypeScript client code—approximately ten thousand lines of code and thousands of files, representing weeks of work and about $1,000 worth of Claude Code tokens—had vanished. Not just from his project directory, but all files and their backups were gone too. It had also (yay) vanished from the remote Bitbucket repository. Steve experienced "that heart-stopping moment where you cycle through the five stages of grief in a few hundred milliseconds"—like when you accidentally delete a production database and you know there's no backup.

By sheer luck, Steve eventually noticed an open terminal window with an orphaned clone of the code—it was the last remaining copy of that code on Earth. Had he closed that terminal or *even left the directory*,* everything would have been permanently lost. His AI assistant had created numerous Git branches with cryptic names. During a cleanup operation, Steve had instructed it to remove "unneeded" branches, not realizing those branches contained uncommitted code that unexpectedly hadn't been merged to main, including most of the node client. We describe how to prevent, detect, and correct these types of problems in Part 3.

---

* This is the deleted Unix file system inode problem. If he had left the directory, it would have been garbage-collected away without a trace.

The Near-Hardware Disaster: Physical Consequences

Digital mistakes are bad enough, but AI can also cause physical damage. Our friend Luke Burton, an engineer who spent two decades at Apple and is now at NVIDIA, was using a coding agent to create a tool to automate firmware uploads to a CNC machine. However, during a vibe coding session, he almost hit Enter before realizing his AI assistant had proposed wiping out the CNC storage device.

Luke texted us in alarm: "It all scrolled by so fast, I almost missed it. I was one Alt-Tab away from having to factory restore the machine. That would have involved getting access to the rear panel, and this machine weighs 100 pounds." AI-initiated coding mistakes can extend beyond software, damaging physical devices or systems. (Again, we'll describe mitigations in Part 3.)

The Disobedient Chef: When AI Ignores Direct Instructions

Gene worked with AI to handle Trello API authentication. Despite explicitly telling it to "Read the file from the Java resources directory—here's how you do it," the coding agent ignored his directions and still wrote code that accessed it through the file system directly instead.

The code still worked…when Gene ran it from his project directory. But had he not caught this mistake when he inspected the coding agent's changes, it would have caused his code to fail when used as a library in another program—a subtle time bomb that might not have been discovered until weeks or months later. As we'll explain in Part 2, AI can have problems with instruction following, getting worse when its context window becomes saturated. We'll teach you how to detect when this is happening and what to do about it.

Genius but Unpredictable

As these stories reveal, vibe coding is like working with an extraordinarily talented but wildly inconsistent sous chef. On good days, this sous chef can create masterpieces beyond your wildest expectations, transforming simple ingredients into culinary magic. But on bad days, the same chef might burn down your kitchen, poison your guests, or disappear mid-service. With a regular sous chef, you might lose a meal or waste some ingredients. With AI, you can lose more—functioning code, critical tests, whole repositories, or phys-

ical hardware. (And to add to the indignity, the AI vendor will charge you for the privilege of destroying your meal and recreating the dishes it ruined.)

These cautionary tales aren't meant to scare you away from vibe coding—we remain enthusiastic advocates for many reasons. But they do underscore why the techniques and safeguards in the rest of this book are so important. Without proper supervision, taste-testing, and kitchen practices, your AI sous chef can transform from your greatest productivity asset into your worst nightmare. And when that nightmare happens, you may become the reason for the executives banning AI chefs from the restaurant chain.

These concerns about AI's potential downsides aren't just based on personal experience—they're now showing up in data. The work Gene did on the *State of DevOps Report*s continues at Google's DORA research group. DORA's 2024 report dropped a surprising finding: Every 25% increase in GenAI adoption correlates with 7% worse stability (more outages and longer recovery times) and a 1.5% slowdown in throughput (deployment frequency and lead times).[1]

This finding certainly supports the sobering stories we shared above. However, we call the finding the "DORA anomaly" because it's at odds with our common experience that vibe coding can *also* increase throughput and preserve stability. This led to us starting a joint research project in early 2025, and we hope to create additional guidance on what factors are needed to vibe code well. (More on this in Part 4.)

Every big new technology has growing pains, marked by mishaps and even disasters before safety features and good practices emerge. You can reduce the risk through careful task decomposition, rigorous verification, strategic checkpointing, and more, as we show you later in this book. We've made these mistakes, so you don't have to—and we've developed battle-tested approaches to ensure your vibe coding journey delivers all the FAAFO benefits without the downsides.

"These Seem Like Pretty Rookie Mistakes"

Many people we admire and whose opinions we trust gave us wonderful feedback on this book. However, several people told us: You two are experienced engineers, having either built large-scale systems at Amazon or Google or

researched deeply effective software delivery practices for decades. And yet it looks like you forgot about basic things like version control or automated testing. These seem like pretty rookie mistakes, and you let AI go wild and wreak havoc on your code.

Maybe you were thinking the same thing; we're glad that they brought this up. We made the above mistakes despite having what we thought was a healthy dose of caution and paranoia. However, we were like people who have spent decades riding a horse and are then given the keys to a modern passenger car. Or maybe more accurately, a modern F1 racing car. We wrecked our car. Many, many times.

Like everyone on the planet, we have been learning to use these new and novel tools with few, if any, antecedents. Someone used to riding horses will have few of the required mental models, muscle memory, and habits required to drive a car. The good news is that the same core principles and practices that allow us to deliver software sooner, safer, and happier as we went from one software deployment per year (which was typical in the 2000s) to 136,000 deployments per day (which Amazon achieved in 2015) can be scaled up as we go from generating a hundred lines of code a day to thousands and beyond.

We'll explore this deeply in Part 3, where we describe how to modify our inner, middle, and outer development loops.

Tomorrow's Promise vs. Today's Reality

The day will come when you can turn to your AI sous chef and say, "Prepare a five-course meal for tomorrow's important client," and then walk away. The sous chef, deeply attuned to your culinary philosophy, flavor preferences, and restaurant standards, could be trusted to take over completely. It understands your explicit instructions, the unstated context, your restaurant's history, and your long-term vision.

When you return the next day, the meal is planned, ingredients prepped, stations organized, and everything ready for flawless execution—just as you would have done, or better. We believe that day is on its way. But as of mid-2025, we're still a long way off from having that kind of trust. Since 2019, the

time horizon of tasks AI can reliably complete has continued to double every seven months,[2] from maximum task lengths measured in seconds in 2019 to now nearing several hours.[3] Researchers project that AI will be able to complete months-long software tasks within the decade.

But as of mid-2025, we're still navigating a significant capability gap. Your current AI sous chef is undoubtedly classically trained with a knife and has read every cookbook. But when left unsupervised on larger tasks, we've witnessed AI coding agents:

- Transform code bases in ways that horrify their owners.
- Get trapped in endless research loops, continuously investigating without completion.
- Spiral into increasingly complex solutions to fix problems in their code.
- Overengineer simple features with unnecessary abstraction layers.
- Create documentation that increasingly diverges from what the code does.
- Gradually disable or bypass critical functionality as they lose sight of the original requirements.

Understanding this gap—which continues to shrink—and learning to work skillfully within it are crucial for effective vibe coding. Rather than being discouraged by current limitations, successful practitioners adapt their approach to maximize AI's present capabilities while preparing for its rapid evolution:

1. **Delegate thoughtfully:** Choose well-defined, smaller tasks where success criteria are clear and verifiable.
2. **Supervise appropriately:** Monitor more closely when the task is novel, complex, or high impact.
3. **Establish guardrails:** Create explicit boundaries for what AI should and shouldn't modify.
4. **Check work regularly:** Verify outputs to catch issues early, especially for critical system components.

5. **Create persistent references:** Create documentation that helps your AI assistant understand your project and preferences.

The gap is real, but it's also temporary. Learning to bridge it effectively today is a critical part of, as Dr. Karpathy best put it, embracing the exponentials. We'll talk in great detail about what each of these means in practice in Parts 2 and 3.

Conclusion

The good news is that in spite of these limitations, AI coding assistants can accelerate your development process. A carefully supervised AI can help you achieve FAAFO benefits—working faster, tackling more ambitious projects, accomplishing more autonomously, having more fun, and creating more options.

The gap is closing. Each advancement in AI memory, context retention, and instruction following brings us closer to the AI ideal where we can trust it to achieve large tasks unsupervised for a long period of time. Dr. Thomas Kwa and coauthors suggest in their paper "Measuring AI Ability to Complete Long Tasks" that the day is coming when AIs will be able to do months of unsupervised software engineering work reliably.[4] The techniques we share in this book not only help you work effectively with today's AI tools but also position you to take immediate advantage of any and all improvements as they emerge.

In Part 2, we'll explore detailed strategies for working within current constraints, including techniques for supervision and quality control. For now, approaching your AI with a clear-eyed understanding of both its potential and its limitations will help you maximize its benefits while avoiding the pitfalls that come with a sous chef who sometimes can't remember where the trash can is and improvises.

CHAPTER 5

AI IS CHANGING ALL KNOWLEDGE WORK

So far, we've been focused on how AI is changing the world for software professionals. But the ripples of this revolution are spreading wider, touching nearly every corner of knowledge work. In this chapter, we'll explore this broader transformation because understanding the big picture is key to navigating your own path within it.

Let's look beyond AI's impact on coding to its impact on professions ranging from financial analysis and legal research to writing and design. We'll make parallels with the Industrial Revolution and the dawn of the internet. AI is a force reshaping how work gets done, and who is doing that work. It's reconfiguring the jobs themselves, as well as the skills that matter.

We'll show highlights from the famous "OpenAI Jobs Report," discuss historical precedents with thinkers like Tim O'Reilly, and share some provocative scenarios of explosive economic growth (as well as some less rosy futures).

You'll see why we're optimistic that, for those of us able to adapt, AI can help us escape drudgery and engage with more meaningful challenges. It will also reinforce why embracing vibe coding unlocks more of those FAAFO benefits—fast, ambitious, autonomous, fun, and optionality—in everything you do.

Disruption Outside of Software

If you're reading this, chances are you're a knowledge worker—be it software developer, infrastructure and operations, product manager, UX designer,

financial number-cruncher, artist, you name it. Your job involves thinking, analyzing, creating, and communicating. You use computers as a big part of your job.

If that's you, then your job is going to change. A groundbreaking 2023 study by Dr. Daniel Rock and his colleagues, colloquially called the "OpenAI Jobs Report," delivered some shocking news: Researchers estimated that 80% of US workers could see AI impact at least 10% of their tasks, potentially more.[1] They hinted that automating cognitive tasks could create far more economic value than automating physical labor ever did. However, they found that the jobs most exposed were high-wage knowledge workers—mathematicians, tax preparers, financial analysts, writers, and web designers. Wow.

They found that only thirty-four occupations were "safe." These jobs required physical manipulation and specialized equipment operation, like motorcycle mechanics, short-order cooks, and floor sanders. Or, as our colleague Brendan Hopper, Group CTO at Commonwealth Bank of Australia, described it, "moving atoms for a living."[2] These roles depend on manual dexterity and real-time physical feedback that LLMs cannot augment.

The most affected (i.e., least safe) tier included software developers, alongside lawyers and other information wranglers. AI sous chefs are becoming adept at writing code, crafting documentation, analyzing systems, researching legal precedents, summarizing depositions, and churning out reports.

Oh, how fortunes change. We remember the days, not so long ago, when many of us knowledge workers watched automation impact millions of manufacturing jobs,[3] perhaps sitting in our $2,000 ergonomic chairs and sipping our $10 cappuccinos, smugly assuring each other that "our" creative, complex work could never be automated.

Knowledge-work jobs may not be automated away for a long time, but… as Dr. Andrew Ng, one of the founders of Google Brain and now at Stanford University, said, "AI won't replace people, but maybe people [who] use AI will replace people [who] don't."[4]

Now, does this sound bleak? We don't think so. We genuinely believe this revolution is fantastic news for our profession. It promises to help us escape the drudgery, the repetitive tasks, the parts of building software that drain our energy and joy. As our tie-dyed friend Dr. Erik Meijer provocatively declared, "We are likely the last generation of developers who will write code by hand…But let's have fun doing it!"[5] That's the spirit we want to capture.

We want to teach you to harness these powerful new tools. We want you to learn vibe coding so you can write better code faster, be more ambitious, and rediscover the fun in creating software.

Beyond the Junior Developer Debate: AI's True Impact on Engineering Teams

Traditional professional kitchens have a clear hierarchy: Head chefs design the menu and oversee operations, experienced line cooks handle complex dishes, and new apprentices learn by starting with simple tasks like chopping vegetables and washing dishes.

For decades, we've organized software engineering teams in the same way: Senior principal engineers design project architecture, mid-level engineers build complex features, and junior developers learn by handling small, contained tasks. This hierarchy shaped how we hired, trained, and promoted engineers. It's how most of us learned the ropes.

AI, being super fast, changes everything. Let's visualize this using a "task tree."* Big company goals form the trunk, branching into major features, which then sprout smaller branches and finally leaves—individual functions, tests, documentation bits. Historically, those leaf nodes were the proving ground for junior talent.

Many have noted that AIs excel at these leaf-node tasks. Tasks that once took a junior developer days might now be handled in hours by a senior engineer guiding an AI assistant. Steve's head of AI trained and deployed a machine learning model in an afternoon. Had it been done the previous year, it would have been a two-month summer intern project. This observation partly inspired Steve's June 2024 "Death of the Junior Developer" post.[6] In the FAAFO model, senior engineers can do things faster and more autonomously, which (we thought at the time) cuts the junior developers out.

But the reality is more nuanced and, frankly, more interesting than a simple replacement story. Unlike what we thought, *everyone* in the organization will be using AI.

---

* In reality, we know that this task tree is actually a task graph—a directed, hopefully acyclic, dependency graph.

Junior developers will not become redundant. Far from it. Their role is evolving. Instead of primarily executing leaf-node tasks, they might become the "station leads" of the kitchen, who help integrate contributions from non-engineers across the company. We're seeing a fascinating trend where people outside traditional engineering roles—UX designers, product managers, infrastructure operations—use AI to contribute directly to the code base. A junior engineer, like a junior doctor, is still highly trained and can be super valuable in helping this new generation of budding "field medics" contribute directly to the code.

Software delivery is evolving into a vibrant ecosystem, where all roles are now contributing to the code. One UX designer we know, Daniel, was frustrated by a missing feature and built it himself (along with tests) with AI's help, impressing the engineering team.

We hear more and more stories like Daniel's. We believe junior developers will increasingly work with these creative professionals and knowledge workers, including helping them and integrating their work, because most of it would have been done by junior developers in the past. This makes them a good resource for helping less technical people perform that work.

Vibe coding is starting to happen anywhere in the organization where people are waiting for developers or engineers. In the past, these people were either stuck, had to use outside vendors, or had to escalate up the hierarchy. Now, they can create the software themselves—building prototypes, fixing issues, and maybe building features (or at least starting them).

Senior engineers will become responsible for more because what can be accomplished will be greater (ambitious), and they'll be responsible for the contributions of many people, all armed with AI.

With the vision we see unfolding of all knowledge workers beginning to vibe code, engineers still have important roles, though they will be different. Offering a pragmatic perspective amid these shifting roles, Dave Cohen, VP of Engineering at UTR Sports (and a former engineering leader at Facebook and Google), gives advice we all should find heartening:

> *Don't worry, engineers—the current generation of AI tools won't replace you anytime soon…*[7]

There Will Be More Developer Jobs, Not Fewer

We talked with Tim O'Reilly recently, who invented the term "Web 2.0" and is famous for his publishing empire, which has taught us many essential skills. We got onto the topic of AI coding, and he reminded us that we've seen this movie before. Every single time we've had a significant leap in programming technology, people predict the programmer apocalypse:

- "High-level languages will kill assembly programmers!"
- "Visual Basic will replace professional developers!"
- "Low-code platforms will make developers obsolete!"
- "No-code tools mean the end of software engineering!"

However, each time programming got easier, we needed more programmers. Easier tools meant more people could build software, which created new categories of applications, which spawned new industries, which required…you guessed it…more developers.

Look at what happened with the web. HTML was dead simple compared to C++. Everyone and their grandmother could make a webpage. It did the opposite of killing programming jobs. It exploded the demand for software, creating millions of new programming jobs across countless new businesses.

Dr. Matt Beane, author of *The Skill Code* and famous for his work on studying the "novice optional problem," speculated on the variety of new roles that could emerge in the software creation process. We talk more about his prediction of what new software roles might get created in Part 4, based on his study of the latest roles that were created in fulfillment centers as more work was automated.

Furthermore, existing roles will all become enhanced with AI. A security engineer is still a security engineer, for instance, but they will be using AI to automate a lot of the job. Security engineers have always wanted to implement fixes directly in the code, but it's not always feasible for them to know every language and framework at the company. With AI, they can confidently make security fixes and add defenses across the company's code, provided the work is reviewed by an appropriately leveled engineer.

This pattern of AI role augmentation starts to capture Scott Belsky's notion of "collapsing the stack" we mentioned earlier—where Daniel, the UX designer, is proving that he, too, can be an engineer, and he can start to work his way up in engineering experience by building software with his own hands. Likewise, professional engineers no longer need to wait on or be blocked by UX designers; engineers can take on many UX responsibilities in less user-critical scenarios.

The UX designer role seems to be broadening—a UX++ role that straddles the line between designer and engineer. Daniel gives us a glimpse of a world where UX specialists implement the UX layer themselves rather than relying on developers. In this new world, people will vastly prefer working with UX designers who participate in development rather than sitting on the sidelines in Figma, opening tickets for developers to resize panes and move buttons.

So, what does this mean for jobs, precisely? Will everyone need to learn to code? Let's study a comparable situation that unfolded with photography and see if we can learn anything from it.

When digital cameras first appeared, professional photographers scoffed, convinced that mastering f-stops, lighting, and film chemistry was the only real path to capturing great images. Yet over the following decade, an unexpected shift occurred: Digital photography didn't shutter the profession—it blew open the doors. Suddenly, anyone with a smartphone was an amateur photographer, creating billions more photographs.* This explosion in photography birthed new industries—social media influencers, image-sharing networks, online portfolios—and dramatically expanded the overall demand for professional imagery.

The same dynamic will likely unfold with software creation. As vibe coding tools become increasingly intuitive and widespread—and eventually, as easy to use as smartphones—software development moves from a specialized discipline accessible only to highly trained engineers, toward something anyone with a good idea can go after.

We've already seen teenage vibe coders building robust gaming apps—something once reserved for industry veterans. In this environment, software

* Wes Roth presented an outstanding description of the phenomenon. There were nearly two trillion photos taken in 2024.[8]

will become as ubiquitous as photos and videos, an everyday medium for communication, collaboration, and creativity.

As you might still hire a professional photographer for demanding shoots, there will always be a critical need for highly skilled software engineers in areas that demand exceptional resilience, security, and enterprise-level scalability. (Say, software for airplanes or CT scanners.)

Get ready for a world where software becomes another form of creative expression, and where the millions of little features that someone needs, languishing in a bug backlog, can be built and implemented by anyone.

Our math here is simple and optimistic: When you lower barriers, more people create stuff. And those creations—whether digital photos or software apps—create new markets, opportunities, and yes, more jobs.

Could AI Lead to Annual 100% Global GDP Growth?

Some economists and AI researchers are making a bold, almost ludicrous claim: that AGI could eventually double global GDP *every year*.[9] We're talking about a 100% annual growth rate when the global economy has been puttering along at 2–3% for nearly a century.

Let's put this into perspective: Before the Industrial Revolution, economic growth barely existed. We had roughly 0.01% annual growth for thousands of years. Then the Industrial Revolution arrived, and growth jumped to 1–2%.[10] That 100–200x increase completely transformed human existence.

The Industrial Revolution created a virtuous economic cycle that had never existed before. Steam power and mechanization exponentially reduced the cost of production across manufacturing and agriculture, allowing companies to offer goods at lower prices while maintaining their profits. As these goods became broadly affordable, demand exploded.

This surge in demand prompted businesses to scale production, creating more jobs and higher wages. Workers with increased purchasing power bought more goods, reinforcing the cycle. Each technological breakthrough—from the steam engine to the assembly line—amplified these effects throughout the economy.

So, when people talk about AI potentially causing another 30x jump in growth rates, there definitely seems to be historical precedent. That's only one-third of what happened pre- and post-Industrial Revolution! Think about what happens when production costs drop across industries simultaneously. When computing got cheap, we did unprecedented things—we created smartphones, cloud computing, and whole digital ecosystems nobody predicted.

As the cost of production drops across energy, manufacturing, healthcare, and education simultaneously, new goods and services will be rapidly created, with software being developed not over a year but over a weekend. This accelerated pace will be driven by a growing number of individuals creating new software. As more people innovate and build, new things will become possible, demand will explode, and economic output will go through the roof.

Who knows if it will happen. There are obstacles—resource constraints, energy requirements, political resistance. But we don't think the argument is completely crazy, and that's what makes it fascinating. We could be witnessing the beginnings of an economic transformation that makes the Industrial Revolution look like a minor speed bump in human history.

There are risks. AI could lead to algorithmic micromanagement of developers, analogous to what we've seen in gig work and warehouses. But that's exactly why the "head chef" mindset we advocate is so important—you stay in control of the tools, rather than letting them control you.

As Mat Velloso, VP of Llama Developer Platform at Meta's Super Intelligence Lab and formerly of Google DeepMind, said, "When AIs started beating humans in chess, we assumed it was game over. But then they learned that if you team an AI with a human, that team can beat AI alone. There's something beautiful about that analogy in this world: Devs will be teaming up with AI, not being replaced by it."[11]

Conclusion

Today's AI has plenty of limitations. It makes up function names that don't exist, forgets what it was doing halfway through a task, and occasionally

insists with complete confidence that 2+2=5. But focusing on AI's current limitations is like judging the automobile industry on the 1908 Model T.

Here's what it means to embrace the exponentials, again from Mat Velloso: "This year, very likely AI will surpass human ability in coding. It's happening. Just like it crossed the bar in many other things before (playing Chess, Go, etc.)."[12]

Whether that happens this year or in the years to come, the FAAFO benefits will keep growing—they compound with each leap in AI capability. When AI becomes 4x smarter, you'll be 4x faster, but also new transformative capabilities will emerge. Those who embrace AI collaboration now will develop instincts and workflows that position them to thrive as these capabilities expand exponentially.

These trends resonate deeply with both of us. Gene has watched as tasks that took days in 2023 now take hours in 2025, and tasks that were impossible for him are now routine. Steve has seen problems he'd abandoned years ago become solvable with a few strategic conversations with an AI agent.

Our message to you amid this whirlwind is to *embrace it*. As long as you lean into using AI, your development life stands to get steadily better, thanks to FAAFO. You'll be faster, more ambitious, more autonomous, have more fun, and gain loads of optionality. AI elevates *your* ideas, *your* ambitions. It becomes an amplifier for *your* creativity.

CHAPTER 6

FOUR CASE STUDIES IN VIBE CODING

Before we dive into the techniques and frameworks that underpin vibe coding, we want to share with you some field reports of real experiences. We'll tell a tale of an experienced developer tackling a side project, share two stories of world-class engineering teams solving important business problems, and regale you about a person who hadn't programmed in nearly twenty years building tools to solve her problem.

These anecdotes are real-world demonstrations of people achieving FAAFO. They give us a taste of the transformative potential that vibe coding will inevitably deliver at scale in technology organizations.

Building OSS Firmware Uploader for CNC Machine

We mentioned our friend Luke Burton, who spent nearly two decades at Apple managing engineering efforts around some iconic moments. Some of his achievements include being responsible for the technical readiness of the 2014 WWDC introduction of the Swift programming language to millions of developers. Luke has worked in and around the many systems that support iOS and MacOS, including working on improving the security of the iPhone supply chain.

Recently, Luke's hobby has been playing with CNC machines, which are meticulously crafted devices that carve intricate metal parts with knife-edge precision. But as Luke has become interested in modifying the CNC firmware, he's discovered that the firmware development environment is woefully challenging.

Luke is one of those hobbyists who tinkers deeply with their tools. He found that firmware testing is typically done on the CNC machine, instead of locally on the developer's laptop, which would be much faster and safer. Furthermore, uploading the firmware requires cumbersome `telnet` commands.* Unit tests of the firmware seemed almost vestigial, which made modifying the code seem treacherous and unpleasant.

After hearing what we've been working on, he wondered whether vibe coding could help him fix some of these problems. One evening, using Claude Code, he proved to himself he could navigate and start modifying the CNC tooling and code base. Soon afterward, he texted us about how he had created a Python program that automated the upload of firmware to the CNC machine, significantly reducing the friction: "2600 lines of Python with documentation and proper CLI flags. It cost me $50 in Claude Code tokens, but I'm not complaining!" It took him two hours, and he was multitasking the whole time.

Seeing what he built, his collaborator in Germany was amazed, prompting Luke's enthusiastic reply: "You ain't seen nothing yet—give me 15 minutes, and this thing will have an interactive mode with GNU readline support."

He showed this tool to a few people, and they immediately told him, "I NEED THIS." The original controller program is notorious for being unusable because it doesn't allow copying and pasting, there is no "file open" dialog box, the navigation keys don't work, etc.

He didn't complete it in one step. It took patience and iteration. Claude Code struggled to handle strangely compressed files referenced in the original CNC firmware ("I couldn't have done it any better," he said). He eventually switched to Cursor, which used the same Claude Sonnet 3.7 model, and fed it code from another Python program that worked. With AI's help, he got it working in two tries.

This is an example of someone achieving FAAFO. Also, someone who is clever about using multiple tools to push through to a working solution. Furthermore, Luke's contributions will help everyone who is helping improve the CNC firmware better, faster, and safer.

---

* In simple terms, `telnet` is a protocol and command-line tool that lets you connect to systems on the network from the early days of the internet (1969). Think of it as the unencrypted ancestor of `SSH`.

Christine Hudson Returns to Coding

As we were working on the book, we got to help someone vibe code for the first time. Our friend Christine Hudson did her master's degree work in machine learning in 2004 but hadn't coded in fifteen to twenty years. She decided to try vibe coding.

For her first project, she chose to export her Google Calendar entries to another Google account. This is something that she would never have considered attempting before AI—the *ambitious* in FAAFO.

One of the first things we had to figure out was which developer environment would be best here. We preferred not to have to configure a local environment. During the session, we tried Google Apps Script, Google Colab notebooks, and terminal apps. All three of us used different approaches to implement the same task, with the goal of having something working in ninety minutes.

Unexpectedly, Christine was not only the first to complete the task but also the only one who succeeded at all. Using Google Apps Script, she successfully exported her calendar to Google Drive as an ICS calendar file. Steve attempted to replicate her approach in real time but did not succeed because of an obscure error with his authentication. Meanwhile, Gene's approach, using Python in a Google Colab notebook, got stuck in a similar spot, trying to create a Google OAuth consent screen.

Steve and Gene were tangled in the barbed wire that all programmers have to overcome: Dealing with everything the program needs to interact with that's out of your control—worse, when it's external services. Every encounter with a third-party API is a chance for a dead-end and retracing your steps.

Christine is now a vibe coder. We're happy that she succeeded, even though we both fell flat on our dumb faces. We had steered Christine toward Google Apps Script because of a crucial benefit: It was already authenticated and had built-in access to Google Calendar APIs. And that was the key that unblocked her.

This insight—knowing which path would avoid authentication complexity—shows the real advantage that experienced developers have. They know the broader technology landscape and have developed some judgment about

which approaches are better than others. And then they pick the wrong one, but their student gets it right. But, hey, at least someone succeeded.

We asked Christine about how the experience felt on a scale of 1 (worst experience ever) to 10 (best experience ever). She said there were moments of pure joy ("+10") when she saw the code being written for her, creating an almost magical experience of effortless creation.

And how would she rate her most frustrating part? We were afraid her experience would be a -10, and she'd never want to do this again. After all, we had all struggled in frustration with external obstacles, like Christine's failed Google Cloud sign-ups, the countless error messages, Claude rate limits, switching to ChatGPT, and not being able to upload screenshots. But no. Christine said it had been mildly annoying, but no more so than the computer troubleshooting she has to do every day.

Gene and Steve felt the frustration more than Christine did because they wanted the experience to be seamless, and there were a lot of obstacles. The fun parts of coding had been accelerated, but all the rest of the time we were stuck on miserable troubleshooting. Steve quipped that vibe coding can sometimes be like a hellish trip to Disneyland, where all the rides and fun parts have been compressed to half a second…and all you're left with is waiting in line. But that wasn't Christine's experience at all. She found the process fulfilling and took pride in what she built, despite the setbacks. She, too, was experiencing FAAFO.

Let this be an inspirational case study for anyone who wants to "return to code." You can have as much ambition as you like, and build things you always wanted to build, and it's infinitely easier than it ever was. We welcome you back.

Adidas 700 Developer Case Study

After seeing Luke and Christine's hobby projects, you might be thinking that vibe coding is not suitable for "real work in the enterprise." If you believe this, you're not alone. But this is why you need to know about the work of Fernando Cornago, Global VP of Digital and E-Commerce Technology at Adidas, and responsible for nearly a thousand developers.

Adidas generates nine billion euros of revenue annually and is one of the top five e-commerce brands in the world. Formerly responsible for their platform engineering, Fernando is passionate about providing developers with the tools they need to be productive. In 2024 and 2025, he delivered an experience report on their 700-person GenAI developer pilot—an experiment with vibe coding in a large-scale enterprise environment.[1]

This was their second pilot. The first pilot had spectacularly flopped, with 90% of developers hating the coding assistant tool. The reviews included phrases such as a "total waste of time" and nothing but "firefighting and troubleshooting." Such was life on the trail in the pioneering days of AI-based coding (i.e., early 2024) when the tools and models weren't good enough to be useful.

However, with those learnings, they tried again. This second pilot is now entering its second year. As we described earlier, Cornago reported that 70% of developers experienced productivity gains of 20–30%, as measured by increases in commits, pull requests, and overall feature-delivery velocity. Not bad. More importantly, developers reported feeling 20–25% more effective in their daily craft. Also not too shabby, especially as this was all done before coding agents, which are 10x more powerful and addictive.

Among the things that made Fernando most proud is that most of his engineers report a 50% increase in what they call "Happy Time." More precisely, that's the amount of time developers spend on things they want to do, which includes hands-on coding, analysis, and design. That implies they're spending far less "Annoying Time"—unrewarding work such as attending meetings, troubleshooting their environments, dealing with brittle tests, or tedious administrative tasks.

We'll describe the factors that differentiated these two groups, which leaders need to know about, in Part 4. In short, the happier teams worked in loosely coupled architectures. They had clear API boundaries, fast feedback loops, and independence of action. Vibe coding worked well for them.

This tale demonstrates how vibe coding requires creating an environment where developers can do their best work. With the right architecture and fast feedback loops, vibe coding can increase developers' productivity and satisfaction with their jobs. And these happy developers can best achieve organizational goals.

Elevating Developer Productivity at Booking.com

Booking.com is one of the largest online travel agencies, with a team of more than three thousand developers. Bruno Passos is Group Product Manager, Developer Experience. His mission is to eliminate developer roadblocks so his teams can do their best work. Over the past year, Bruno has been heavily involved in Booking.com's GenAI innovation efforts within engineering—another example of vibe coding at enterprise scale.[2]

Booking.com has a storied history of a culture of experimentation, where almost every feature decision is tested, typically through feature flags—a practice that involves deploying multiple versions of a feature to production and then measuring which one best achieves the desired business goals. One downside is that the code base is full of never-used functionality behind disabled feature flags, legacy code, and old experiments.

The result was developers spending 90% of their time on frustrating toil rather than productive coding. This became one of the focus areas for using Sourcegraph's AI code assistant and search tools. Their developers reported a 30% boost in coding efficiency, with significantly lighter merge requests (70% smaller) and reduced review times.

In Part 4, we'll discuss more of the strategies and tactics Bruno used to achieve these results. Booking.com's creative strategies included educational initiatives that transformed skeptical developers into enthusiastic daily users. They also held days of training with each business unit to help ensure developers knew enough to be successful.

Initially, Booking.com's developer uptake of vibe coding and coding assistant tools was uneven. Some developers embraced their new AI partner; others didn't see the benefits. Bruno's team soon realized the missing ingredient was training. When developers learned how to give their coding assistant more explicit instructions and more effective context, they found up to 30% increases in merge requests and higher job satisfaction.

Bruno's leadership defined short-, medium-, and long-term goals focused on faster merges, higher-quality code, and reduction of technical debt. Sourcegraph and its specialized agents enabled developers to commit 30% more merge requests, with smaller diffs, and reduced review times.

Bruno emphasized that tools alone weren't enough. They supported development teams across the enterprise with targeted, hands-on hackathons and workshops. As a result, initially hesitant developers became enthusiastic daily vibe coders who are finding FAAFO.

Conclusion

These four case studies—spanning from hobby projects to enterprise-scale implementations—illustrate the transformative potential of vibe coding across different contexts and skill levels. Luke's CNC firmware project demonstrates how individual developers can achieve ambitious goals with newfound efficiency. Christine's return to coding after a twenty-year hiatus reveals how vibe coding can make programming accessible and enjoyable again for those who had previously stepped away. The Adidas and Booking.com implementations show how large organizations can systematically improve developer productivity, happiness, and business outcomes when the right conditions are present.

As we move forward in this book, we'll explore the techniques and frameworks that can help you and your organization harness this revolutionary approach to software development.

CHAPTER 7

WHAT SKILLS TO LEARN

The world is trying to figure out what changes and what doesn't change when every developer is using AI on everything they're working on, and which skills are the most important in this new world.

Because tools will evolve rapidly, core traditional software engineering principles will play at least as large a role, if not larger. Thus, it's essential to:

- Create fast and frequent feedback loops for validation and control.
- Create modularity to reduce complexity, enable parallel work, and explore options.
- Embrace learning in a world where everything changes fast.
- Master your craft to thrive in an environment where all knowledge work will be changing in a short timeframe.

Learning these techniques will be critical for everyone in knowledge work, not just developers and vibe coders.

Creating Fast and Frequent Feedback Loops

The faster a system goes, and the more consequential the risks of failure, the faster and more frequent feedback you need. When a system is slow-moving, and nothing too bad happens when you make a mistake, you can get away with feedback loops that are slow and infrequent. For instance, in most cases, no one minds if a software build takes a few minutes longer than usual, so we can tolerate longer feedback cycles. However, as you speed a system up, such as when we increase code generation speeds by 10x or more, we need

feedback cycles to speed up just as much, if not more. Feedback loops are the stabilization force that allows us to stay in control and steer the system toward our goals.*

Let's compare two chefs: Chef Isabella runs her kitchen with a fanaticism for feedback. Thermometers are checked, dishes are tasted at every stage by multiple cooks, servers relay customer reactions instantly, and specials undergo trial runs before hitting the main menu. When a slightly off-putting aroma wafts from the paella, she catches it *before* it reaches a customer. Her kitchen adapts when things go wrong during every service. She experiments with menus throughout the season and maintains her restaurant's stellar reputation.

On the other hand, Chef Vincent is equally skilled but operating in a feedback vacuum. Dishes go untested until they land on the table, cooks work in silos, and servers don't bother giving feedback anymore. When that batch of questionable seafood makes it out, the results are predictable: unhappy (and unwell) diners, scathing reviews, and maybe a visit from the health inspector. Vincent's failure isn't one of skill but of process—a failure to build in (let alone act on) rapid feedback.

For instance, in our stories when AI-generated code generation spiraled out of control, we didn't create fast and frequent enough feedback. Our old habits proved to be wildly insufficient. You keep things safe and under control by building incrementally, testing frequently, and validating relentlessly. By doing so, you build trust in your AI partner and minimize rework—that soul-sucking and most expensive type of work. It doesn't mean progress has to be strictly linear. You can explore multiple paths in parallel, like an army of ants searching for the best route to food, but each path needs its own frequent checkpoints.

In fact, as Gene and his colleagues Jez Humble and Dr. Nicole Forsgren found in *The State of DevOps Report*s—a cross-population study that spanned 36,000 respondents over six years—that fast feedback loops, through CI/CD, were one of the most significant predictors of performance.[1]

---

* The Nyquist stability criterion from control theory tells us that to maintain control over any system, our feedback must operate at least twice as fast as the system itself. AI-assisted development requires proportionally faster feedback loops as generation speeds increase, a bit like how a race car driver needs faster reflexes at higher speeds.

In Part 2, we'll give you practical techniques for:

- Creating fast feedback loops.
- Leveraging AI to perform validation tasks and making checks faster and less error-prone than manual review alone.
- Ensuring you're building the right thing (validation) and building the thing right (verification).
- Using feedback to steer your project effectively, perhaps toward that elusive product-market fit.

To achieve FAAFO, you must have the skills and processes to build trust in what your AI collaborator creates. Trust us first: Going fast without feedback is dangerous.

Creating Modularity

While fast feedback provides a control mechanism for moving quickly and safely, modularity partitions our system. It allows us to do work in parallel, creating independence of action. It makes the system more resilient, and it enables the low-risk exploration of alternative solutions (i.e., options).

In high-pressure and high-intensity situations, modularity can be the difference between a well-run professional kitchen and utter pandemonium. It's the principle that allows different parts of a system to operate and evolve independently, and it directly impacts whether your team thrives or burns out.

Dr. Dan Sturtevant and his colleagues did research that showed how developers working in tangled, non-modular systems are 9x more likely to quit or be fired.[2] And again, *The State of DevOps Report*s showed that a modular architecture was also a top predictor of performance.[3]

Alexander Embiricos from the ChatGPT Codex team described how an engineer using AI tools achieved excellent "commit velocity" building a new system from scratch. But when they ported it "into the monolith that is the overall ChatGPT code base that has seen ridiculous hypergrowth" (that is, a system with architectural problems) the results changed dramatically. Despite

having the "same engineers, same tooling," their "commit rate just plummets." This real-world example shows that even at OpenAI, architectural constraints affect developers using AI too.[4]

Let's revisit Chefs Isabella and Vincent. Isabella's kitchen is a model of modularity. Each station—pastry, grill, sauce—is distinct, with its own space, tools, and responsibilities. Chefs work independently, experimenting within their domain without causing system-wide meltdowns. When the pastry chef tries a new technique, the grill chef isn't dodging flying flour. Communication *between* stations is clear and standardized. This independence allows them to work in parallel, combining elements from different stations to create exciting new dishes reliably.

Contrast this to Chef Vincent's kitchen, which is a war zone of entanglement. Shared tools vanish, cooks bump elbows, and chefs and servers collide. A simple task requires navigating a maze of dependencies. Forget parallel work; chefs literally wait in line, blocked by others. His talented team is hampered not by lack of skill, but by the sheer friction of the system. Yes, sometimes new "dishes" emerge, but usually by accident when ingredients crash into each other. We've seen code bases like this, where developers (and their AI partners) can't touch anything without triggering explosions elsewhere.

We want modularity in our code and projects, because it enables the independence of action for coding agents (and people) to work in parallel. We want to have them work on different tasks—refactoring a module, implementing a feature, writing tests—without causing horrendous merge conflicts (or worse, subtly) or breaking unrelated functionality.

Good modularity also builds resilience. Like cloud software designed to handle failing disks, a modular system contains failures; if one module has a problem, the blast radius is limited. You can often isolate or replace it without taking down the whole system.

Modularity also unlocks *optionality*, a cornerstone of FAAFO. It allows you to explore different solutions in parallel. If you want to try three different caching strategies, you can build them as alternative modules. If you need to experiment with a new UI component, you can develop multiple versions. Keeping your system modular gives you freedom.

In Part 2, we'll describe techniques such as:

- Task decomposition and breaking complex problems into smaller, manageable components with clear interfaces.
- Working with multiple agents simultaneously to enable work to happen in parallel without creating interference, or worse, giant merge conflicts.
- Branch management and version control strategies to explore multiple options.
- Agent contention detection to discover when agents are interfering with each other's work.
- Enabling experimentation and exploration by creating modules, where you can try a bunch of things, mix and match, and pick the best combination.

Later, we'll touch on a formula (NK/t) that helps quantify this power of parallel experimentation. And naturally, the faster your feedback loops, the more experiments you can run, increasing your chances of finding the best approach. In short, modularity helps achieve more in all of the FAAFO dimensions.

Embrace (or Re-Embrace) Learning

We've already talked about the importance of architecture and fast feedback loops in your AI-assisted kitchen. But there's a third, equally crucial element that underpins everything, especially when your sous chef is an AI: You have to become re-accustomed to *learning*. AI is changing so rapidly that it is going to take constant learning and practice, at least for a while, to develop the good judgment you need—by taking risks, learning from mistakes, and adapting.

Think about our chefs again. Chef Isabella brings in new sous chefs, complete with their eccentricities, who are often challenging to wrangle. However, she knows that this is the future and becomes a relentless learner. She experiments (which can result in surprises or failures), does controlled trials, and seeks out other head chefs who are on their own journey. And with her new team, she learns to create ever more ambitious dining experiences

that meet her customers' increasingly demanding tastes. And somehow, it's more fun than before.

On the other hand, Chef Vincent tries working with these new sous chefs a couple of times. One overcooked the fish, one deflated the soufflé, and one accidentally set their dish on fire. Vincent posts pictures of these culinary calamities on social media, ridiculing these strange new chefs, earning him his fifteen minutes of internet fame. But in time, he finds himself left behind as the culinary and dining world changes rapidly around him.

You might be surprised to learn that learning is learnable. You can improve your ability to learn at any time in your life. It's coachable, teachable, and you can make your brain become more neuroplastic and adaptable through focus and lifestyle changes. Personally speaking, we have learned more in the last year or two than we have at any point in our careers—at an age, to be frank, when learning isn't as easy anymore.

Learning means doing. It means tackling problems that seem insurmountable. It means taking risks, patiently wading through your mistakes, pushing until you get the outcomes you want, and troubleshooting creatively when things go wrong. Your willingness and indeed eagerness to improve how you learn will give you constant leverage in the next few years as AI ascends to touch all knowledge work.

Here's an example. When Gene first started vibe coding with Steve, Gene was convinced that the then-new OpenAI o1 model would be great at `ffmpeg` and could help him overlay captions onto video excerpts. That is to say, subtitles on YouTube clips. Two hours later, Gene ran around in circles, typing increasingly complex `ffmpeg` commands.

The AI was more than wrong; It was *confidently* wrong. Thinking about that particular Sunday afternoon still causes Gene to clench his jaw. But he learned an important lesson on when to give up on using AI to solve certain types of problems. It was a crummy experience, but he learned from it *because* it was a crummy experience. You learn by doing.

Cultivating a learning mindset has nothing to do with innate genius. Learning is about deliberate and intentional practice, much like Dr. Anders Ericsson described for mastering any complex skill.[5]

You need:

- **Expert coaching:** Leverage mentors, peers, and AI itself (asking it to explain concepts or critique approaches).
- **Fast feedback:** Build those tight verification loops we discussed, so you immediately see the results of the AI's work and your prompts.
- **Intentional practice:** Consciously work on skills, like prompt refinement or evaluating AI suggestions in unfamiliar domains. Chop wood, carry water—or rather, vibe code, review output.
- **Challenging tasks:** Push yourself slightly beyond your comfort zone, using AI for problems you couldn't solve alone yesterday.

In Part 2, we'll describe how you can:

- Master the "count your babies" technique to systematically verify that AI delivers everything you asked for, preventing silent omissions that can break your systems.
- Develop your "warning signs detector" to spot AI's subtle shortcuts and confidently challenge it when something feels suspicious.
- Use AI as a world-class consultant on topics you don't fully understand or want to learn about.
- Craft suitably sized tasks that fit AI's attention span, preventing the corner-cutting that happens when its context window gets overwhelmed.
- Implement strategic checkpointing rhythms to create a safety net of recovery points throughout your development process.
- Deploy "tracer bullet testing" to validate whether AI can handle tightly scoped technical challenges before investing significant time.

In short, achieving FAAFO becomes an exercise in "being a great learner." Your commitment to continuously learning how to interact with, guide, and validate AI is what enables you to go faster, confidently pursue ever-more

ambitious outcomes, whether working alone or as part of a team, and explore more options.

Mastering Your Craft

At this point, we've equipped your kitchen with AI-powered sous chefs. You've heard some stories, and by now you're somewhat aware of both their potential upside and their potential dangers. We've hinted that you're now the head honcho in your new role as a software developer, and we've repeatedly assured you that vibe coding will be more fun than any kind of software development you've ever done.

But we haven't addressed the elephant in the kitchen: None of it matters if you don't like cooking.

Chef Isabella thrives because she loves cooking. She may not be an expert in all the techniques or latest tools, but she has a vision for what she wants, she knows what's important to her in the moment, and she can manage sous chefs who may know specific areas better.

Chef Isabella lives to cook, while Chef Vincent cooks to live. He stopped learning any new techniques ages ago. He's satisfied as long as the food tastes "decent." As a result, few people wind up going to Chef Vincent's restaurant because…well, his food is not that great.

Building things you love, or at least setting a determined vision and goals for yourself, will help you find and acquire the skills you need. Especially with AI there to help. All you need is the desire.

In Part 2, you'll:

- Develop an intuitive understanding of the limitations and strengths of these AI tools, just as great chefs know when to trust their equipment and when to intervene.
- Get an overview of how AI code generation works, enabling you to use AI to build things in languages you haven't used before.
- Learn how to pick things you love to work on, which will naturally drive the right learning behaviors, unlike following trends without purpose.

- Transform coding from a solitary activity into a collaborative dialogue that deepens your understanding with each iteration.
- Build a creator's mindset that focuses on meaningful outcomes rather than getting lost in tool obsession or technical trivia.

Our advice: The more you throw yourself into vibe coding, the more you'll master your craft of creating software—and that's the high-level goal, isn't it? Cook things you love, and cook different cuisines, which will force you to learn new tools and techniques. And of course, achieve ever-higher levels of FAAFO.

Conclusion

We began this journey exploring Dr. Erik Meijer's striking declaration that "the days of writing code by hand are coming to an end." It's a provocative statement, to be sure. But it's probably the simplest way to describe the fundamental transformation happening in software development. What started with ChatGPT and other AI assistants, at first seemed like a toy, but has evolved within two years into professional vibe coding, a new approach that's reshaping how we create software.

In Part 1, we've examined the five dimensions of value that vibe coding creates: writing code faster, being more ambitious about what you can build, doing things autonomously or alone that once required teams, having more fun, and exploring multiple options before committing to decisions. These benefits combine to create a step change in what's possible for developers at all levels. The economics of what's worth building have opened up, and projects once eternally deferred are now within reach.

For both of us, these benefits have transformed our lives in deeply personal ways. Steve, after watching his beloved game Wyvern languish with over thirty years of unfixed bugs and aspirations, saw a path forward. For Gene, vibe coding reopened doors to coding that had seemed closed since 1998, enabling him to write more code in 2024 than in any previous year of his career.

Hopefully we've convinced you why vibe coding is important. Now we're ready to move into the kitchen and start cooking. In Part 2, we'll hand you the knives, fire up the stoves, walk you through your first vibe coding sessions, and then step you through the theory and fundamentals to do it well.