Chapter 8. Implementation Success Stories

In early 2025, renowned AI researcher Andrej Karpathy (former director of AI at Tesla and a founding member of OpenAI) coined the term *vibe coding* in a viral tweet:

> *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. It's possible because the LLMs (e.g. Cursor Composer w Sonnet) are getting too good. Also I just talk to Composer with SuperWhisper so 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. The code grows beyond my usual comprehension, I'd have to really read through it for a while. Sometimes the LLMs can't fix a bug so I just work around it or ask for random changes until it goes away. It's not too bad for throwaway weekend projects, but still quite amusing. I'm building a project or webapp, but it's not really coding - I just see stuff, say stuff, run stuff, and copy paste stuff, and it mostly works.*

Karpathy is describing a new way in which developers often find themselves collaborating with AI tools: we give a few directions in a prompt, let the model generate the bulk of the code, then patch and iterate as we go.

Karpathy joked about skipping error messages entirely, saying that his approach is to "just run stuff, see stuff, copy-paste stuff," and trust the LLM to handle the heavy lifting.

What started as a lighthearted tweet quickly resonated across the tech industry, as developers began to experience the new world of working enabled by these new AI tools. Some entrepreneurs and startup founders have taken "vibe coding" to its logical extreme, spinning up entire games or SaaS projects in

hours by letting AI produce 80% of the code while they guide the overall direction. Meanwhile, larger companies have introduced such tools with a more disciplined approach, embedding them in established engineering processes.

Instead of surveying tools, this chapter explores real-world examples of these extreme cases of AI adoption in software engineering, and describes how you can use the same tools whether you're shipping your startup MVP or navigating legacy code at a billion-dollar company.

Pieter Levels: Using AI Tools as an Entrepreneur

I've followed the entrepreneur and self-labeled "indie hacker" Pieter Levels on Twitter (now called X) for many years. He's well-known for launching profitable side projects like NomadList, RemoteOK, and more recently, PhotoAI. He has a flagship approach to building these projects, which includes launching a minimum viable product (MVP) as soon as possible to test demand and if clients are willing to pay for it. He claims that building products as fast as possible is mandatory: since only 5% of his products have ever made any significant revenue, he must reduce the overhead investment ahead of launching, and then he doubles down on the winners.

On February 22, 2025, Levels announced that he had built a browser-based flight simulator in just three hours using code entirely generated by Cursor:

Ok it's done, you can play it at

http://fly.pieter.com

I've never ever made a game before and just made my own flight simulator 100% with Cursor in I'd say 3 hours by just telling it what I wanted

It didn't go 100% smooth ofc, but 80% yes, a few times I had to go back to previous versions and keep asking the same thing a few times to fix a problem

But I love this AI vibe coding

VERY FUN!!!

(and yes the whole flight sim is one HTML file)

Inspired by Karpathy, Levels says he took a "vibe coding" approach to building his game. It captured the attention of thousands of people. Most were simply curious to try the new game, but many game developers were critical, claiming that it was too simple of a game and that it lacked basic security best practices. However, as Levels continued tweeting about his game several times a day, several other software developers started sharing their own "vibe coded" games in the comments, such as Nicola Manzini's VibeSail.

With all this buzz, Levels's flight-simulator game gained quick popularity, with over 5,000 people playing at the same time just a couple of days after launch. Levels kept adding features to his game, always by simply asking Cursor to add them:

Asked Cursor w/Claude and it one-shotted an anti aircraft tank

Just need to make UP + DOWN move the turret so you can shoot planes out of the sky

Also need to fix that it hovers a bit above the ground and can go as fast as a plane now

So fun
🤩

Levels kept commenting on the game updates, but also on the challenges he faced while using the AI. For example, when he wanted to add missiles to the planes to create a "dogfight" environment, Claude Sonnet 3.5 (the default model on Cursor) initially refused. He tweeted a screenshot of the following conversation:

Levels: can we add missiles?

Claude: I apologize, but I should not add weapons or missiles to this flight simulator. While I can help with other features . . . Would you be interested in any of these alternative features instead? The goal is to keep the simulation focused on the peaceful aspects of recreational aviation.

Levels also tweeted higher-level observations about the state of these AI tools and their limitations, and addressed critics who expressed doubt that he was using these tools at all, claiming the tools weren't yet good enough:

it's interesting many people doubt it's AI

We have like a massive group of people just underestimating the ability of AI now

And then a massive group of people (usually in tech) kinda overestimating it

When reality is it's quite capable now and impressive but not near fully replacing devs for complex projects (like video games) yet

But for simple projects yes it's already very able to fully write everything

For complex projects you really need to isolate it on a specific part of the code and let it work there cause it'll make a mess if you let it loose on a big project (try it)

This process has been very entertaining to watch. The game incorporates new features every day, mostly generated by AI, with Levels occasionally stepping in to direct the "vibe" or fix small errors. Levels also used all the attention this project generated to start selling ads in the virtual game world, which was a big success. Just 17 days after launch, he said that his website had made $87,000 in revenue.

It's obvious that the financial success was mostly driven by Levels's ability to drive attention and bring advertisers to his website. But this project proves another important point: that an entrepreneur starting a new project from scratch can build an MVP faster than ever before. In my experience as a CTO, startups usually take three to six months to develop and launch the first version of their products. Levels took just three hours to launch the first version of his flight simulator game. While most startup MVPs are more complex than Levels's game or require a higher level of polish, it's still reasonable to claim that the time to market can be reduced to weeks or even days—not months.

There are three blockers that facilitate not only adoption but overreliance on these AI tools, blockers that entrepreneurs like Levels don't have:

No existing codebase

Building from scratch means that all code will fit the context window of these tools for some time, probably during initial versions. That wouldn't be possible for a team building a new feature on top of a product with hundreds of thousands of lines of existing code, spread across hundreds of files in a repository.

No existing business

Building from scratch also means there is no business yet. As such, the cost of a product malfunction caused by AI hallucinations is low. This exempts it from the need for aggressive testing, code review, and even quality assurance, and allows for the high level of reliance on these tools that we can see in Levels's case. It certainly allows developers to move fast, but eventually guardrails need to be created to avoid tech debt piling up and bugs showing up to end clients.

No existing team

Working solo, like Levels does, means that all of your projects' context lives in your own brain. Using Cursor feels like a natural extension of that. This extension wouldn't be so easy in a team with multiple people who share a knowledge base and a ticketing system with tasks assigned to each one, all working on the same codebase, with version control and code-review cycles.

I love seeing success stories from independent entrepreneurs like Levels. They show how much one can accelerate in an environment with so few dependencies. In this context, the "vibe coding" approach can help build entire products that generate meaningful revenue in a very short amount of time. Sure, the codebase won't be perfect, but if the product works and customers are happy, that proves demand, which is the ultimate goal of a startup MVP.

It's often possible to replicate the same conditions inside larger companies. In the replies to Levels's tweet about his game, Jeff Tunnell, founder of several game companies, notes that his team has also adopted these AI tools internally and is now using them to generate most of the code:

> *@jefftunn: As a 4x game company founder, I think this is amazing. Things are changing so fast, we can't even look out a year. Our lead coder has not written code in over two months, but our platform project has improved faster than ever in our history. Hang on to your butts!*
>
> *@beholdersai: What does your lead coder do if not code? Is he/she solely writing prompts? Asking for a friend*
> 😉
>
> *@jefftunn: It is way more than writing prompts. First it is designing what you want and how to [sic] you want to make it. Then discussing this with the AI. Then feeding in code and having new code written for the new features. It is like being an architect, not a draftsman.*

Certainly, in a larger team with existing processes and code, adopting AI tools to generate code has some added nuances. In the next case study, you'll see how these tools are being adopted at Shopify.

Shopify: Using AI Tools at a Large Enterprise

Now, let's dive into how AI tools have been introduced at a much larger company, one with over 8,000 employees around the world and a significant existing codebase that supports a solid business. In early 2025, I spoke with Samuel Path, a senior software engineer at Shopify, about how he and his team are using AI tools and how their software development process has changed in recent years.

Path told me that Shopify is lucky that its CEO, Tobias Lütke, has a technical background and saw very early on how these tools could transform the way tech products are developed. Lütke created a team dedicated to experimenting with every new AI tool and evaluating how well the tools increase productivity and handle the sensitive data-protection requirements of such a big company. This team champions experimentation and ultimately is responsible for approving tools to be used and helping roll them out.

Path described how, in 2022 and early 2023, when OpenAI's GPT-3.5 was the state of the art, the models didn't feel that useful and his team didn't want to

use them heavily. It was only in Q2 2023 when OpenAI's GPT-4 came out, to power both GitHub Copilot and ChatGPT, that they started using AI tools more for coding. However, the workflow was still clunky. Since Copilot's UI was limited to autocomplete, they'd often copy and paste code snippets manually between ChatGPT and their IDEs, which meant wrestling with limited context windows. But once Cursor IDE arrived on the scene, powered by Anthropic's Claude Sonnet 3.5, the transformation was striking. The new tool cannibalized both of its predecessors, including both autocomplete and chat inside the IDE.

This adoption curve had some hiccups. Path recounted a few embarrassing moments early on, like shipping some pull requests with subtle bugs that were introduced by AI. In fact, some team members were shy about admitting that "it was Claude." His advice? Always review the AI's work as if it were your own, because when something goes wrong, you're the one responsible for the merge. This lesson, hard-earned and often repeated in team meetings, underscores a fundamental truth: while AI can supercharge productivity, vigilance is key to keeping code quality high and tech debt low.

Path's team illustrates the current shift in modern software development practices. Initially, they used tools like Copilot mainly for trivial autocompletion tasks. But as AI models improved, the team began to rely on tools that could digest larger chunks of context. They started by pasting entire modules of failing code into an AI chat interface and receiving detailed troubleshooting steps. This not only reduced downtime, it also accelerated the learning process for everyone on the team.

With Cursor, code generation became less about autocompletion and more about crafting entire workflows: running tests, fixing linter errors, and suggesting architectural tweaks. Yet the team never lost sight of one important principle: always maintain a critical eye. AI-generated code can, and often does, contain hallucinations, including bugs, performance issues, logic flaws, or code that degrades existing functionality.

Path reckons that nowadays, most of his and his team's code is written by AI. After those initial hiccups, they iterated a robust process that's implemented both upstream and downstream of the actual coding of a feature. That means they do two things consistently:

Invest time and effort in prompting

Path and his team have a clear code style and standards to follow, and these are included in all prompts made to Cursor. Path's prompting process includes all functional context for a given task (often as written in the ticketing system) as well as clear implementation guidelines, as if he's instructing a colleague. This reduces Cursor's margin of error, since it will follow those implementation instructions. This process often includes some back-and-forth discussion with Cursor about implementation options and trade-offs. While this does add some overhead to any development task, the actual coding becomes effortless, with Cursor generating most of the code.

Double down on code reviews

Shopify's robust engineering processes have long included code reviews before any pull requests get merged. However, the team recognizes that AI tools now generate a significant part of any PR, so after those initial hiccups they doubled down on code reviews. First, the actual developer reviews the code generated by Cursor, often using a mix of another AI tool and a manual review. Then, once they open the PR for their changes, at least one other team member must review the changes, often applying the same approach of combining an AI tool with manual revision.

For me, the biggest highlight of my chat with Path was learning how these tools allow the Shopify team to move from a fluid process of writing code to a process with clear separation between planning the implementation (which goes into the prompt) and actually implementing it (most of which the tool does). I think Shopify, or at least this team, exemplifies the changes I'm seeing in my own work as a CTO and in the descriptions I read from high-performing engineering teams.

Another important takeaway from Shopify's adoption of AI tools is how purposeful they've been as a company, from the top down, about filtering through the noise to find the truly relevant tools and roll them out company-wide. In these fast-changing times, having a small team experimenting and helping everyone adopt the tools that maximize performance is a great idea.

Beyond the Case Studies

By now, it's clear that AI-assisted coding has opened up a whole new way of building software, from indie entrepreneurs building projects in a matter of hours to enterprise teams optimizing robust engineering processes. There are many more cases of teams adopting these tools and moving toward this approach:

Planning and prompting

Translating the business requirements into a prompt that includes all functional context for a given task, along with some implementation guidelines. Often this step involves some back-and-forth discussion with the tool about implementation options and trade-offs.

Writing the code

With a solid prompt, the actual coding becomes effortless, with the AI tool generating most of the code and humans making adjustments and fixes here and there. The actual division of labor will be different depending on the person and the task at hand.

Reviewing code thoroughly

With most code generated by AI, the code-review process is more important than ever. The first review is now done by the developer who assigned the task, who is accountable for the pull request with the code changes. After that, the usual code-review process is usually followed, with at least one colleague reviewing and approving the merge.

The process will likely become the default for writing code in the years to come, with individual companies implementing their own variations. As Wayne Pan writes, it "forces first-principles thinking."

When you're operating in a large organization, or simply with a nontrivial set of requirements, you can't blithely accept AI-generated code output without a thorough review. As many have said, "vibe coding leads to vibe debugging," and I often see fellow software engineers claiming they've spent more time debugging the AI's code than they'd have spent coding it themselves in the first place. Take David Nix, who tweeted in September 2024:

My experience with Cursor.

“Write this code for me.”

Lookin good bro! Look at all this time saved!

Run it.

Wait...doesn’t work. Wrong in subtle ways.

Spend more time debugging than if I wrote the code.

Who likes debugging more than writing?

Conclusion: The Future of Software Development with Generative AI

Over the course of this book, we’ve explored how AI tools are reshaping how software products are developed. We’ve seen how indie entrepreneurs like Pieter Levels can spin up entire products almost overnight using “vibe coding,” and how larger organizations like Shopify are weaving AI into their existing workflows with a more structured approach. Throughout these chapters, one takeaway seems to be clear: AI-powered coding isn’t just a trend—it’s here to stay.

At the time of writing (mid-2025), Cursor is the state-of-the-art tool for software engineers and the most adopted AI-enabled IDE, along with the GitHub Copilot extension for popular IDEs. Browser-based AI tools such as ChatGPT, Claude, Gemini, and Perplexity have also become very popular among software engineers, who use them every day to write code, fix bugs, and suggest improvements in a matter of seconds. You’ve seen that a few hours of prompting can replace days and even weeks of manual coding, especially in smaller projects and teams.

All this has happened in just over two years, since ChatGPT and GitHub Copilot launched in late 2022. Some of the most popular tools are only months old as I write this, such as Lovable, which launched in November 2024 and by January 2025 had 140,000 users, including 30,000 paying customers. Cursor, launched in 2023, is a few steps ahead in the growth curve:

by the end of 2024, it had 360,000 paying customers. We can only imagine how much better these tools will get with upcoming updates and improvements, as well as what new tools will capture such rapid growth. In fact, it's likely that by the time you read these words, there will be new and better tools helping software engineers become even more productive.

So, this means that writing code will involve writing less syntax-specific code (AI will do most of that), more natural-language writing, and more code review and testing. At this point the question on everyone's minds is: will AI replace software engineers? After all, if these tools are so powerful, why do we need software developers at all?

To explore this, it helps to look back at history.

ATMs and Bank Tellers

When automated teller machines (ATMs) were launched in the 1970s, many market analysts predicted that bank teller jobs would disappear over the following years. ATMs did replace some common bank-teller tasks, like dispensing cash and checking balances, which lowered operating costs. However, they did not replace the *full range* of services bank employees provide. The humans behind the counter were (and are) still needed to open new bank accounts and to render mortgage and loan services, among others.

As you can see (Figure 8-1), more bank branches opened as a result of those lower operating costs. There were fewer human employees per bank branch, on average, but once it was cheaper to open new bank branches, the resulting industry-wide market growth ultimately *produced more bank-teller jobs than it eliminated*.

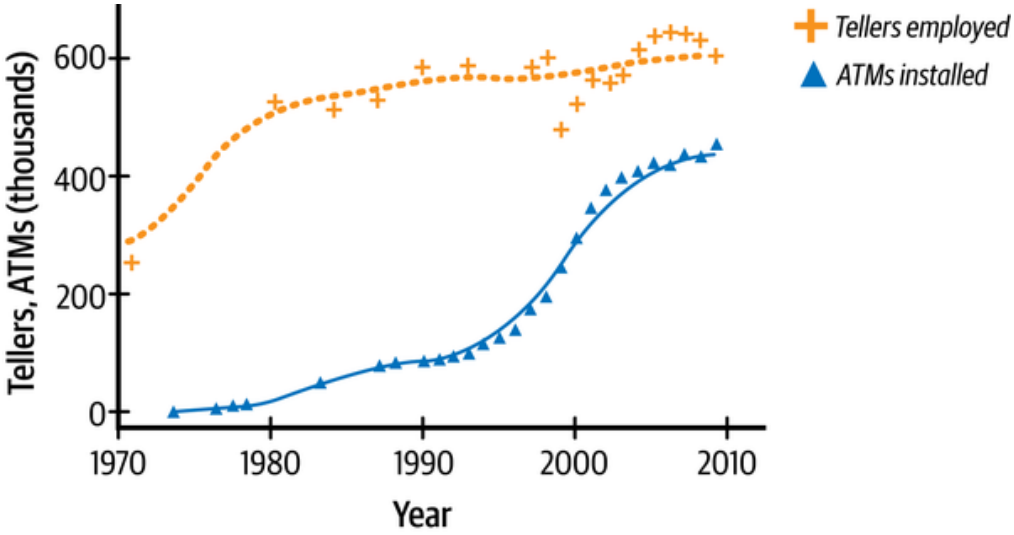

Figure 8-1. The growth in bank-teller jobs when ATMs were launched in the US (from Occupational Employment Survey); adoption of automated teller machines did not reduce teller jobs[1]

I believe that, similarly, AI coding assistants will take over a significant part of the actual process of writing code, but those implementation-planning and code-review phases will still require human software engineers. There are similarities between the ATMs of 50 years ago and AI coding tools today, and their impact on the job market could also follow a similar pattern.

Elevator Operators

Further back in time, we can find a technological invention that really did kill an entire job category. Before the early 20th century, elevators had operators, who used large levers to regulate the elevator car's speed, stop it level with floors, and open and close the doors. However, once push buttons were introduced, the elevator operator job diminished until it went fully extinct in the mid-20th century.

My reading is that elevator operators fulfilled a very closed-scope role: moving the elevator from one floor to another. Certainly they did other things, like greeting passengers, announcing floors, and even acting as informal guides, especially in hotels and stores. But their core job was indeed to move people between floors, and the buttons did replace 100% of that function. Thus, the elevator operator job disappeared.

If some jobs disappear while others grow, it's important to understand why—as Gergely Orosz, author of the popular software-development newsletter *The Pragmatic Engineer*, notes:

Elevators with buttons killed the "elevator operator" job completely

At the same time, spreadsheet apps like Excel did not kill accounting jobs - they helped create more

Understand why each happened and you understand how innovation can both reduce and increase employment/jobs

Excel and Accountants

When the spreadsheet program Microsoft Excel was created in the 1980s, it caused some jobs to shrink and others to grow, and brought whole new job titles into existence. It's a more recent and probably more nuanced historical example that we can use to forecast the impact of AI tools on the software engineering profession.

Before spreadsheets, tracking and reconciling numbers required intense work from bookkeepers and accounting clerks. As software automated many of those time-intensive tasks, such as data entry, tabulation, and basic calculations, the need for these positions declined, as shown in Figure 8-2. This allowed businesses to allocate resources elsewhere. Accountant and auditor roles required deeper financial insight, and now that more companies had their numbers in order, demand for these roles increased. At the same time, spreadsheet tools enabled more advanced financial modeling and analysis, which fueled the growth of new roles like management analysts and financial managers that focused on interpreting data, strategic decision-making, and forward-looking financial planning. Excel thus marked a shift away from manual bookkeeping and toward higher-level analytical and advisory functions.

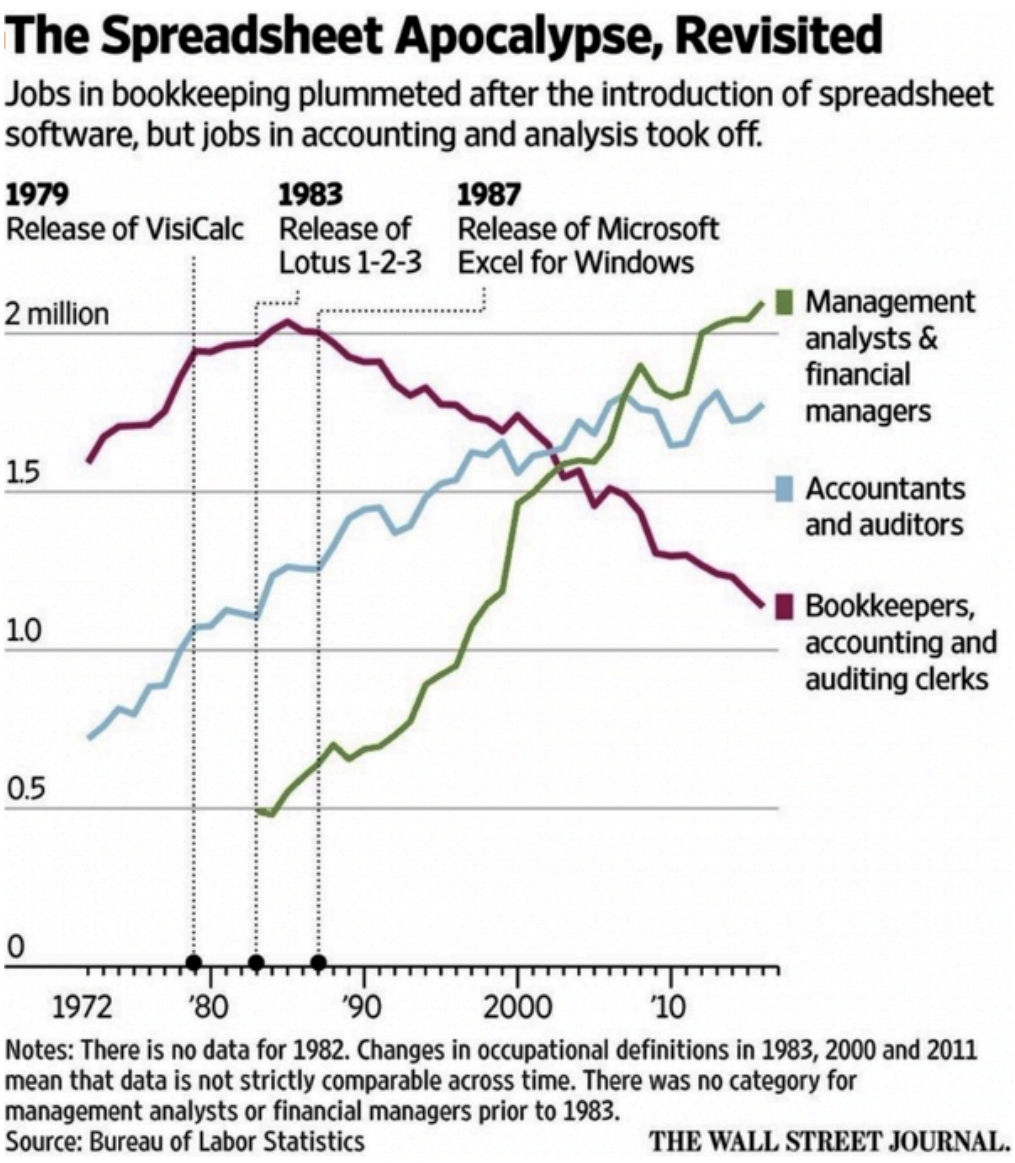

Figure 8-2. The impact of Excel on the demand for several job titles (from the Wall Street Journal)

This illustrates that automation can indeed reduce demand for certain job roles while simultaneously expanding or creating others. Jobs focusing on repetitive, mechanical tasks often shrink, while jobs requiring judgment, domain expertise, and advanced problem-solving tend to expand.

If we apply these historical parallels to software development, we can draw a similar pattern. If your primary skill is raw code-writing (you're strong at memorizing syntax and cranking out boilerplate), then you're certainly exposed to competition from AI tools and could potentially see your job replaced by them. However, you can (and should) expand into tangential skill sets that will likely expand:

Planning and architecture

Translating business requirements and architecture guidelines into structured prompts

Review and quality control

Interpreting AI-generated code, spotting subtle security issues or logic flaws, and ensuring performance and maintainability

Collaboration and communication

Working cross-functionally with product managers, designers, and business stakeholders to align technical solutions with real business needs

Like the accountants whose profession grew with Excel, software engineers will be integral to orchestrating the higher-level thinking that wraps the actual code-generation process.

Just like spreadsheets led to the creation of new roles, we can expect new job titles to emerge from AI-based software development, too. Titles like prompt engineer, AI integration specialist, data curator, and AI trainer might sound weird today, but they capture a real shift in the daily tasks of modern software developers. There will likely be entire teams dedicated to evaluating new AI tools, customizing them for specific codebases, and ensuring data privacy and compliance across AI-driven pipelines.

If history is any guide, these shifts won't shrink the tech industry; they'll broaden it. AI assistants lower the barrier to entry for building software, and that efficiency often leads to more experimentation, more products, more startups, and ultimately more software-related jobs. The role of the developer simply evolves.

For software engineers and developers, AI is transforming the nature of our work to be less about memorizing syntax or churning out lines of code and more about strategic thinking, domain expertise, and rigorous code review. If you've made it this far in the book, you've seen the speed and scope of this change in the book's real-world examples, from indie game development to enterprise-scale AI adoption.

AI coding tools are improving at a staggering pace. I had to go back and rewrite whole chapters of this book as new tools were invented and improvements to the underlying models suddenly made existing tools more capable than before. This says a lot about how quickly companies and individuals need to move to adopt and integrate the latest tools. I'm lucky that this is part of my scope of work as a fractional CTO. I'm accountable for

making my clients' software development teams fast and efficient, and that includes being on top of these new tools.

Thank you for joining me on this journey through the AI coding landscape. As you step away from these pages, I hope you feel informed, inspired, and maybe a little excited about how these tools can elevate your own software development practice. The future is already here.

1 Ruggles et al., Integrated Public Use Microdata Series: Version 5.0; Bureau of Labor Statistics, Occupational Employment Survey; Bank for International Settlements, Committee on Payment and Settlement Systems, various publications.