Hoang Nguyen

AI changes **the Tools**, you still own **the Craft**

A practical guide for engineers and leaders to adapt AI across the stack and the organization

AI changes the Tools, you still own the Craft

A practical guide for engineers and leaders to adapt AI across the stack and the organization

Table of contents

About me

I’m Hoang Nguyen, a software engineer who grew into engineering leadership and never stopped caring about the craft.

Today, I’m Director of Engineering at ShopBack, where I lead teams across regions to build systems that serve millions of users. Over the years, I’ve worked as an engineer, tech lead, and manager. I’ve shipped big systems, made painful mistakes, coached engineers into leadership, and learned the hard way that building good software isn’t just about code. It’s about people, clarity, and judgment.

I started writing at codeaholicguy.com as a habit for more than 10 years, a way to think out loud, reflect on how I work, and share what I’ve learned. When AI tools like ChatGPT, Copilot, and Cursor entered my daily workflow, I didn’t just see a productivity boost, I saw a shift in how we solve problems, collaborate, and define engineering itself. This book came from that shift.

It’s not about hype. It’s about how we adapt.

You can find me here:

- Blog: codeaholicguy.com
- LinkedIn: linkedin.com/in/codeaholicguy
- GitHub: github.com/codeaholicguy
- X (Formerly Twitter): x.com/codeaholicguy
- Threads: threads.net/@codeaholicguy
- Substack: codeaholicguy.substack.com
- Email: hoang@codeaholicguy.com

Preface

It started with curiosity.

I first tried using AI to help write code in early 2022 with a tool called Tabnine. It didn't impress me, it felt slow, the suggestions were often wrong, and it interrupted my focus more than it helped. But then I gave GitHub Copilot a shot when Microsoft released it in mid-2022. That changed everything. It was faster, gave better suggestions, and actually helped me move quicker through my code. That's when I realized: this kind of AI tool wasn't just a cool experiment. It was going to be a real part of a software engineer's job.

I didn't know when it would go mainstream, but I knew we had crossed some invisible line. I remember chatting with some fellow engineers about it at the time. I joked, half seriously, that maybe one day, outsourcing software services might become obsolete. If your job was just translating human requirements into machine code, you were probably at risk. Because soon, in a Microsoft data center, there could be an army of AI machines writing code 24/7, unaffected by mood swings, sleep, or distractions.

That shift, from curiosity to realization, changed how I looked at engineering, tools, and the future of our craft.

Over the past two years, I've watched the conversation evolve. I've seen engineers go from asking, "Will AI replace us?" to "How do I stay ahead?" And I've seen organizations make the same mistake over and over again: rushing to plug in tools without changing how people work, learn, or collaborate.

That's why I wrote this book.

This isn't a collection of hype predictions. It's a grounded, practical guide for people who are serious about doing great work in the age of AI.

If you're a software engineer wondering what skills to focus on next…
A tech lead trying to help your team adopt AI meaningfully…
A founder or senior leader rethinking your team's structure or product workflows…

This book is for you.

We'll cover what's changing, what's not, and what you can do to make the most of this new era, without losing your edge, your craft, or your soul as a builder.

You still own the craft. **Let's get started!**

How to read this book

This book isn't a technical manual, and it's not a hype piece. It's a practical, honest guide for engineers, tech leads, founders, and leaders who are navigating the fast-changing world of AI in software development.

You can read it straight through, or jump to the chapters that speak to your current challenges. Each chapter stands on its own, but together they form a larger narrative about how AI is shifting the way we build software and how we can respond without losing our craft.

If you're a software engineer

Start with **Chapter 2: Coding Isn't the Job** and **Chapter 3: AI as Your Junior Engineer**. These chapters explore how your role is changing, and how to stay sharp in the AI era without just chasing tools.

If you're leading a team

Chapter 5: The New Engineering Workflow and **Chapter 7: Rethinking Team Structure in the AI Era** will help you understand how to adapt your process and structure to fully leverage AI, without losing what makes your team effective.

If you're a founder, senior engineering leader or CTO

Read the **Preface** and **Introduction** for the big picture, then dive into **Chapter 6: How much faster can we really go?** and **Chapter 9: What AI needs from you (and your organization)** to understand how AI can truly impact speed and culture.

What you won't find here

You won't find silver bullets or one-size-fits-all frameworks. This book is full of stories, real-world insights, and practical advice, all written from the perspective of someone who's been in the trenches.

Whether you're just curious or deep in AI adoption, read this book the same way we now build software: with context, with curiosity, and with your judgment in the loop.

Introduction: The AI Moment

The first time I used GitHub Copilot, I expected a toy. A smarter autocomplete, maybe. But after a few sessions, something clicked. It wasn't perfect, but it made me noticeably faster, especially with boilerplate and repetitive tasks. That was the moment I knew that this wasn't just hype. It was the start of a shift.

Since then, the question has evolved. It's no longer "Can AI code?". That's been answered. Now we're asking better questions:

- How should we work with it?
- What should still be done by humans?
- Where's the edge between speed and understanding?

This is the AI moment for me.

The early hype

Back in late 2022, ChatGPT became a headline in every news outlet. In just a few months, it became the fastest-growing app in history. The buzz was everywhere:

- "AI will replace programmers."
- "You can build an app just by prompting."
- "Design, content, testing, engineering, all automated by AI!"

People dreamed big, and startups popped up overnight offering AI tools for everything: slides, UIs, marketing copy, legal docs, code generation. We saw the promise everywhere: "faster software, smaller teams, better automation".

And to be fair, some of that was real. Engineers, including myself, saw immediate boosts in productivity. We could ship things faster, explain ideas more clearly, or get problems solved with a good prompt.

But at the same time, something felt off. Many outputs were "almost right", not right. Code that ran, but wasn't elegant. Docs that sounded smart, but missed the nuance. Designs that looked okay, but weren't usable. We started to realize that AI could do stuff. But building something great? That still needed us.

The shift from excitement to baseline job expectation

What's happening now is a transition from excitement to expectation. AI is no longer a new thing; it's a baseline job expectation.

In many companies, using AI tools is becoming part of the job. Shopify's CEO started this by saying publicly that they expect every employee to use AI in their workflow. Not using AI now feels like not using Google decades ago.

And this shift isn't just about speed. It's about how we work.

- Engineers are rethinking their workflows: What parts of coding should I keep? What can I automate?
- Product teams are exploring new collaboration styles: Can PMs write specs in ways AI can understand and use?
- Leaders are revisiting team structure: Do I still need the same ratio of juniors to seniors?

The story is no longer about AI replacing engineers. It's about engineers who know how to work with AI outperforming those who don't.

How to ride the shift

If you're leading a team, building products, or writing code, here are some practical ways to move from hype to real impact:

Treat AI like a teammate, not a tool.

AI is your junior engineer, who is fast, tireless, and need explicit instruction. Don't expect it to read your mind. Give it clear instructions. Check its work. Teach it your standards.

Make AI part of your workflow, not a separate task.

The biggest gains come when AI is embedded in your daily flow. Use it to:

- Draft your merge request descriptions.
- Refactor repetitive code.
- Generate test cases from API specs.
- Summarize messy meeting notes.

Expect more, not less, from engineers.

Now, AI can write a lot of code. The real value of engineers is shifting upstream. Understanding the problem. Making trade-offs. Guiding the system design. Holding the line on quality.

Update your team's definition of "done".

If your timeline doesn’t shift with AI in the loop, you should ask: “Why not?”. With the right integration, many tasks should move 30 to 50% faster. If that’s not happening, either the workflow is broken, or AI isn’t being used effectively.

Don’t overcorrect.

Don’t overcorrect. AI won’t triple your delivery speed. It might give you 20 to 30% gains. That’s still a huge win, but don’t fall into the trap of overpromising. Sustainable change takes new habits, not just new tools.

This moment is real. Let’s use it well!

AI won’t replace you, but engineers who use AI well will replace those who don’t.

The hype was loud, but the real opportunity is quiet: better workflows, clearer thinking, faster feedback.

Don’t just try AI. Integrate it. Make it part of how you think, build, and collaborate.

This is the shift. The AI moment is here. Now it’s your move.

Chapter 1: The Turning Point in AI

A few years ago, I never imagined I'd spend part of my day prompting an AI model to help me write code, summarize meetings, or debug a flaky test. Back then, AI felt like something reserved for research labs. Sure, you'd hear about it beating humans at Go or generating surreal images, but it felt far from the day-to-day work of building software.

Then one day, it wasn't.

When ChatGPT launched, for the first time, AI wasn't hidden behind a PhD or a research API. Anyone could type into a box and get real, helpful answers.

That was the shift.

AI has been here all along

Let's be clear: AI isn't new. The term's been around since the 1950s. Machine Learning has powered things like fraud detection, recommendations, and search results for years. But most of it lived behind the scenes. If you weren't working at Google or Meta, you weren't building AI, you were just consuming it.

Before ChatGPT, most AI has been narrow and specialized. You'd train a model to do one thing well: classify spam, detect faces, translate text. These systems were powerful, but they were expensive, required tons of data, and needed ML experts to build and maintain. The bar to entry was high.

So for most engineers? AI was something other people worked on.

The Rise of LLMs

In 2017, researchers came up with a new design for AI models called the Transformer. It made a big difference. Instead of just guessing the next word, it starts generating the next words with context awareness. That's what made today's AI feel a lot more helpful, more than just fancy autocomplete.

GPUs got more powerful and affordable, so training big models wasn't just for huge tech companies anymore. At the same time, scraping large chunks of the internet became easier. That helped models like GPT could be trained on everything from books to Wikipedia to random blog posts, forum threads, and public code.

In 2020, OpenAI released GPT-3. It was impressive, but still mostly a toy for techies.

In late 2022, ChatGPT changed everything. Same core model, but better tuned and packaged with a simple chat interface. You didn’t need to understand all the complex Machine Learning concepts to use it. You just asked it a question.

The adoption number was wild. 100 million users in two months. It is faster than any app in history.

Suddenly, everyone, whether you were a student, a teacher, a marketer, a software engineer, or someone with no tech background at all, you could ask it a question or describe a task and see results. For the first time, people could personally experience what modern AI could do, without needing to know how it worked under the hood. This brought AI into the everyday lives of millions.

Chapter 2: Why this matters for Engineers and Teams

So what do we do with this moment? Here are a few takeaways I've seen play out firsthand:

This is not a passing trend

Think of it like the early days of electricity. At first, people just replaced steam engines with electric motors, same workflows, slightly better efficiency. But over time, the factory was redesigned around electricity application. We're now at the AI equivalent of that phase.

The true power of electricity wasn't just in generating energy, but in enabling new tools, light bulbs, refrigerators, and everything else that changed daily life. AI is similar. Its value shows up when people build useful things with it. Applications, tools, and services that solve problems in new ways. This will be the next turning point.

Coding became the perfect entry point for AI

Interestingly, coding emerged as an early AI success. Tools like Bolt, Cursor, and Lovable saw quick adoption, while big AI players released their own coding agents, such as OpenAI's Codex, Anthropic's Claude Code, and GitHub Copilot Agent. They are pushing AI-powered development into the mainstream. But why coding?

There are few main reasons:

- Code has rules and structure, which makes it easier for models to generate syntactically correct output.
- Code can be tested, if it runs, it's probably doing something right.
- Coding is one of the fastest ways to create real impact, because it leads to working software that people can actually use. It's different from math or science, where even a great idea might take years of testing, validation, and approval before the world sees its value. With code, you can build something today and have someone using it tomorrow.

Additionally, abundant training data of public code repositories accelerated learning.

But AI-generated code isn't always good code. It's trained on tons of open-source code, and not all of it is high quality. A lot of that code might be outdated, messy, or full of workarounds. Even if the AI gives you something that runs, it might not be secure,

clean, or built to last. That's why human engineers still matter, we're the ones who check the work, fix the rough edges, and make sure it actually holds up in the real world.

Accessibility is the real revolution

LLMs didn't just get smarter, they were used by everyone. That's what changed everything. You didn't need to understand machine learning or write custom models anymore. AI is now in the hands of every engineer, designer, marketer, and student. That's what makes this the turning point. You no longer need to build the model. You just need to use it well and build with it.

You can't ignore it anymore.

If you're still treating AI like a nice-to-have, you're already behind. It's not a future skill. It's a now skill. And it's becoming a baseline expectation for how teams work, think, and build.

You don't need to master AI to keep up. But you must try it, see what it's good at, and where it still messes up. Think of it like a fast but forgetful teammate, you'll get more out of it when you know when to help steer it.

AI isn't here to steal your job. It's just another tool. The people who'll do well are the ones who aren't afraid to try, learn, and adjust. Keep an open mind. Make it work for you.

Chapter 3: Coding isn’t the job

I still remember the first time someone asked me, “Now that AI can write code, does that mean we can build things 10x faster?”

I laughed. Not because it was wrong, but because, honestly, that’s how it looks from the outside.

Yes, AI can write code. But writing code isn’t the hard part. It never was.

But here’s the truth: If all I did was write code, I’d be out of a job soon. Coding is a tool. Problem-solving is the job.

Real engineering is messy. You spend more time understanding problems than typing solutions. You debate trade-offs, argue with your past self during code reviews, and sometimes just stare at a whiteboard trying to make sense of complexity. The code? It’s just the last step.

Coding was the foundation of building functional software and solving technical problems. Before the AI wave, “coding skill” was the badge of honor. Interviews tested algorithms, engineers measured themselves by the number of merge requests. The more code you pushed, the more valuable you seemed.

And for a while, that made sense. Tools were limited. If you couldn’t code it, you couldn’t build it.

But even back then, the best engineers weren’t the ones who knew the most syntax. They were the ones who understood the why behind the what. They dug into business needs, understood users, and found elegant solutions that stood the test of time. They also had deep knowledge of the tools and languages they used, knew when to apply specific techniques, and were able to make the best decisions based on the context they were in.

Then AI showed up and made this truth impossible to ignore.

AI can now generate decent code with a good enough prompt. It can scaffold apps, write tests, convert between frameworks. What it can’t do well yet is solving problems in context. It doesn’t know your product goals. It doesn’t talk to stakeholders. It hasn’t learned from the pain of fixing bugs or the cost of releasing a change that caused incident on production.

This means that the job of a software engineer is no longer about being the fastest coder in the room, and in truth, it never really was.

The real job of a software engineer is the ability to:

- Frame the problem clearly
- Understand trade-offs
- Guide AI (or teammates) toward a working solution
- Own the result, not just the code

Coding is part of it, but it's not the center.

So, how do you thrive in this new landscape?

Get good at understanding the problem

Don't start with "How do I build this?".

Start with "Why does this matter?".

Talk to PMs. Shadow users. Sketch flows. The more context you absorb, the more precise your solutions and your AI prompts become.

Treat coding like communication

Good code isn't the flashiest or the most clever. It's the one people can understand later without swearing at their screen.

Sure, AI can help you type it faster. But it's still your job to make sure the structure makes sense, the logic holds up, and your teammates (or future you) won't need a map to figure out what's going on.

Practice making trade-offs

AI can give you ideas, but the final decision is still on you.

Do you pick the quick fix or the long-term solution? Add a new table or reuse one that already exists?

A great engineer knows when to keep things simple and when the extra complexity is worth it.

Guide the AI

AI isn't magic. Think of it like a smart but inexperienced teammate. It can help a lot, but only if you give it clear direction.

Break the problem down. Be specific. Check its work. Fix things when they’re off. And don’t be afraid to start over if needed.

Keep learning, but shift your focus

Learn to debug, read systems, and model domains. Study design patterns and architecture.

It’s less about mastering new frameworks, more about knowing when and why to use them. AI can generate the how, you’re still responsible for the why.

In this new world, the engineers who rise are the ones who think like product builders, not just code writers. If that’s you, you’re not replaceable.

Chapter 4: AI as Your Junior Engineer

The first time I really used AI for coding, it reminded me of mentoring a smart but inexperienced junior engineer.

I'd describe a problem in plain English, and it would suggest code. Sometimes it worked. Sometimes it missed the point entirely. But it was fast. Surprisingly fast. And just like a junior engineer, it needed guidance, context, and review.

In a traditional team setup, junior engineers are the ones who:

- Take on smaller, well-defined tasks
- Learn the codebase by contributing incrementally
- Ask a lot of questions (sometimes too many)
- Deliver quick wins when given clear instructions
- Need guardrails to avoid breaking things
- Eventually grow into mid and senior-level engineers by working under mentorship

They aren't expected to make big architectural decisions or weigh complex trade-offs. They're expected to learn, contribute, and grow.

That's not so different from how we're starting to work with AI.

AI today, whether it's GitHub Copilot, ChatGPT, Cursor, or Claude Code. They are becoming our new junior teammate.

It can:

- Scaffold boilerplate code
- Generate tests from API specs
- Suggest solutions in real-time
- Refactor small chunks of code
- Summarize documentation or logs

But it can't:

- Weigh the long-term trade-offs of a decision
- Understand the full system context unless we feed it
- Guarantee maintainability, performance, or elegance
- Spot subtle bugs that require real-world experience
- Design systems with intention

AI becomes a contributor in our source code. And just like with a junior engineer, **the output is only as good as the guidance.**

Treating AI like a junior engineer isn't just a metaphor, it's a practical mindset shift.

Here's how you can get the most out of it:

Give clear instructions

Just like a new hire, AI thrives when it has clear context. The more context you give it, like data structures, constraints, goals, the better the result. Don't expect magic from a vague prompt.

Review everything

AI might suggest 20 lines of working code, but you're responsible for what ships. Skim it at your own risk. Review for performance, security, readability, and fit with your existing architecture.

Use it to unblock, not autopilot

Use AI to support debugging, understand unfamiliar syntax, or automate repetitive tasks. But don't let it turn you into a passive reviewer. Keep your problem-solving muscle strong.

Teach it as you go

Use feedback loops. If a suggestion is wrong, correct it and explain why. Over time, you'll learn to prompt better, and the AI will give better results in return.

Stay the decision-maker

Even if AI can scaffold a feature, ask yourself:

- Does this meet user needs?
- Is it scalable?
- Will my teammates understand and maintain this?

These are human questions. Keep asking them.

At the end of the day, the best engineers will know how to:

- Delegate effectively to AI
- Fill in the gaps it can't see

- Make the final call when it matters

AI vs. Junior Engineer

Task / Trait	Junior Engineer	AI
Writes code	✅ With supervision	✅ With guidance
Understands business context	❌ Needs mentoring	❌ Needs detailed input
Makes architectural decisions	❌ Not expected to	❌ Can provide ideas
Learns and grows with experience	✅ Over time	❌ Stays static (until retrained or better model)
Handles boilerplate and repetitive work	⚠️ Might take time	✅ Fast and efficient
Explains trade-offs	⚠️ With guidance	❌ Doesn't understand consequences
Communicates and asks clarifying questions	✅ Part of growth	❌ Only when prompted properly
Needs code review	✅ Always	✅ Always
Can be mentored	✅ Yes	⚠️ Only indirectly via better prompts
Can become a senior engineer	✅ That's the goal	❌ Never
Suggests solutions	✅ Sometimes, with limited perspective	✅ Often, based on pattern recognition
Write your opinion	…	…
Write your opinion	…	…
Write your opinion	…	…

Chapter 5: Where AI helps (and where it doesn't)

Someone on X said, "With AI, you can do the work of ten engineers".

And sure, it sounds exciting. But let me ask you this: "Have you ever worked on a real-life product and felt like writing code was the only bottleneck?" Probably not.

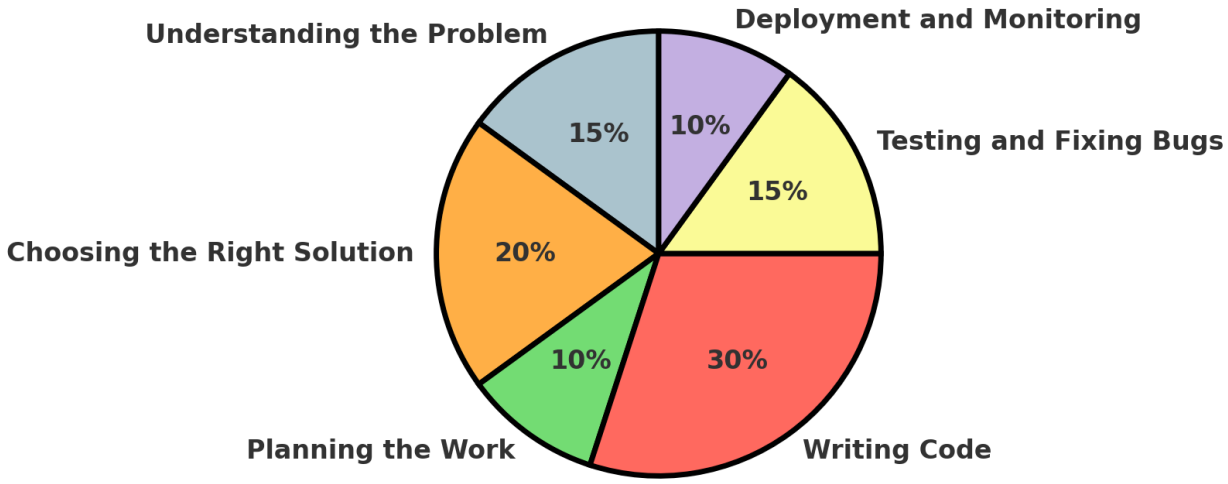
It's tempting to believe AI is a magic multiplier. And in some ways, it is. But if you zoom out and look at the full picture what software engineers actually do, it becomes clearer: AI is a great helper in some parts of the job, and a questionable assistant in others.

Let's break it down.

What do engineers actually spend time on?

We often think the job of engineering as "writing code". But that's just one slice of the pie. Here's a simplified view of what software engineers really spend time on in a typical project:

1. **Understanding the Problem (15%):** Talking to people, asking questions, and clearly defining the problem.
2. **Choosing the Right Solution (20%):** Considering current systems, trade-offs, and deciding the best solution that fits with the current context.
3. **Planning the Work (10%):** Writing clear, actionable implementation plans and system design documents.
4. **Writing Code (30%):** Writing code that's clean, readable, and easy to maintain.
5. **Testing and Fixing Bugs (15%):** Making sure everything works as expected and quickly fixing issues.
6. **Deployment and Monitoring (10%):** Launching the changes safely and closely monitoring its performance.



Now let's ask the real question: in which of these can AI truly help? And where does it struggle?

Where AI makes a real difference

Let's start with the good news.

Writing Code

AI really shines when it comes to writing code. Tools like GitHub Copilot, ChatGPT, Cursor, or Claude Code are great at handling the boring stuff such as setting up boilerplate, creating new files, or writing tests. If you already know what you want to build, it can seriously speed things up.

But, and this is important, it's a bit like working with a junior engineer. It might get the job done, but you still need to double-check everything. Does the code make sense for your system? Is it clean and maintainable? AI won't catch those things unless you guide it.

So yes, it helps you move faster, but it also means you'll spend time reviewing and refining.

A big part of coding is understanding the code that's already there. For years, reading and making sense of existing code has been one of the hardest things, especially for newer engineers.

Now, AI can help speed up that process. It can describe what a piece of code is doing, show you where it's being used, and help you understand how different parts fit together.

But don’t skip the hard parts entirely. The struggle of figuring things out on your own is where real learning happens. AI can support you, but to truly get better and own your craft, you still need to do the work of understanding it yourself.

Planning and Docs

Trying to write a design doc or outline a launch plan from a blank page is never fun. That’s when AI really can help. It’s like someone tossing you a rough first draft so you’re not starting from zero.

It can also help you get organized, breaking down big ideas into smaller chunks, building out timelines, or pulling tasks out of messy Slack threads.

The trick? Be specific. The clearer you are with what you need, the better AI performs.

Testing and Automation

It’s good at writing unit tests based on your code and simulating how users might interact with your app, especially if it knows your API or UI setup. So, most of the testing can be covered by AI, including Unit Testing, Integration Testing, or even End-to-End testing.

It can also help with bug fixing. If you give it enough details, it can suggest where things might be breaking or possible fixes. Give it enough context, and it’ll help trace the issue or throw out a few possible fixes. But don’t expect magic, it still needs direction. It is still a junior teammate: helpful, fast, but not always right unless you steer it properly.

Deployment and Monitoring

AI is starting to become pretty handy when it comes to keeping an eye on your systems. It can go through logs, spot things that look off, and even flag potential outages before they happen, just by learning from past incidents.

It’s not here to replace your observability tools, but it’s like having an extra set of eyes on duty all the time. A helpful backup, not the main monitor.

Where AI falls short

Now let’s talk about the areas where AI isn’t ready, or maybe never will be.

Choosing the Right Solution

Should you take a shortcut or build something solid? Split the service or keep it together? AI might suggest some options, but it doesn't really know your system, your past decisions, or what's coming next.

These aren't just coding questions, they need real judgment. And AI doesn't have that. It can't feel trade-offs or understand the people and goals behind the work. That's why your experience still matters.

Understanding the Problem

It's easy to miss how much thinking goes into understanding a problem. When you get a new task or bug, your brain does a lot behind the scenes, asking questions, noticing when something feels off, and piecing things together.

AI can't do that. It doesn't know how to dig deeper or question what it sees. At best, it copies what it's seen before. But truly making sense of a problem? That's still on you.

Designing for the Long Term

AI can generate working code, sure. But will it be maintainable? Will it respect system boundaries, follow design patterns, or play nicely with the rest of your stack? Often, no.

AI doesn't think in trade-offs. It doesn't care if your service is hard to test or has a leaky abstraction. You do.

Use AI where it helps, own what matters

So, how do you cross the balance?

- **Use AI to accelerate the known**: Tasks you've done before, patterns that are repeated, these are perfect for AI assistance.
- **Don't outsource judgment**: You still need to decide what to build, how to build it, and why. That's the core of your value as an engineer.
- **Review everything**: AI can give you speed, but you give it meaning. Review its output like you would review a new junior teammate's code.
- **Think in loops, not handoffs**: AI is best when there's a tight feedback loop. Prompt → result → edit → re-prompt. Stay in the loop.
- **Invest in context clarity**: Whether for humans or AI, better outcomes come from better input. Make your problem, goals, and constraints explicit.

Use AI to go faster, but don't forget to look up from the keyboard and ask: Are we building the right thing? That answer still comes from you.

Want something practical? Here's a simple decision matrix:

Task Type	AI-First	Human-Led
Write a unit test	✅ Yes	❌
Design a system	❌	✅ Yes
Update docs from chat	✅ Yes	❌
Decide on architecture	❌	✅ Yes
Refactor for clarity	⚠️ Maybe	✅ Yes
Write your opinion	…	…
Write your opinion	…	…
Write your opinion	…	…

Chapter 6: The New Engineering Workflow

Not long ago, building software had a clear rhythm. You'd figure out the problem, design a solution, write the code, test it, ship it, and monitor it. Then you'd do it all over again. There was satisfaction in that loop, and there was a kind of quiet pride in doing it well.

But now, things feel different. You're still solving problems and writing code, but the rhythm feels… different. Some parts are faster, some are messier.

That’s because we’re not doing it alone anymore.

AI is now part of the team.

And it's not just speeding up our typing. It's changing how we think about the whole process, from the first idea to the moment we go live.

AI changes the Shape of the Loop

Let’s recall the engineering workflow:

1. **Understanding the Problem:** Talking to people, asking questions, and clearly defining the problem.
2. **Choosing the Right Solution:** Considering current systems, trade-offs, and deciding the best solution that fits with the current context.
3. **Planning the Work:** Writing clear, actionable implementation plans and system design documents.
4. **Writing Code:** Writing code that’s clean, readable, and easy to maintain.
5. **Testing and Fixing Bugs:** Making sure everything works as expected and quickly fixing issues.
6. **Deployment and Monitoring:** Launching the changes safely and closely monitoring its performance.

Every step was human-driven.

AI doesn't just make individual steps faster, it blurs the boundaries between them.

- While you're writing code, your AI tool suggests structure, edge cases, and even documentation.
- When debugging, AI surfaces possible root causes before you’ve even hit the logs.

- Before you start coding, you can ask AI to scaffold the design doc or break down tasks in Jira.
- During QA, AI can generate test cases from API contracts or user stories, reducing manual work.

The linear path becomes more like a web. You're jumping back and forth more often, and the feedback loop tightens dramatically. Instead of waiting for code review to hear about a potential issue, AI may flag it right in the editor.

It's less "step-by-step", and more "everything, everywhere, all at once".

How to adapt

We need a new mental model and here's how engineers and teams can rethink their workflow:

Start with "How can AI help?"

Every task you pick up, ask: What part of this can AI do faster, or help you think through better?

- Writing a migration plan? Ask AI to generate a first draft.
- Building a feature? Let AI write the boring boilerplate.
- Debugging an issue? Summarize logs and ask AI for a starting point.

Move from "Write Code" to "Shape the System"

You should focus on the why and how well. Think:

- Is this code scalable?
- Are we choosing the right trade-offs?
- Will it be maintainable?

AI can generate code, but you're still accountable for the quality.

Treat Docs and Context as Code

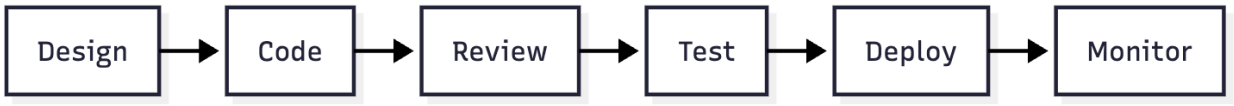
If you want AI to help meaningfully, you need to give it the right context:

- Centralize your decisions in the codebase.
- Keep README, design docs, and task descriptions up to date.
- Log the "why" behind decisions in places AI can read later.

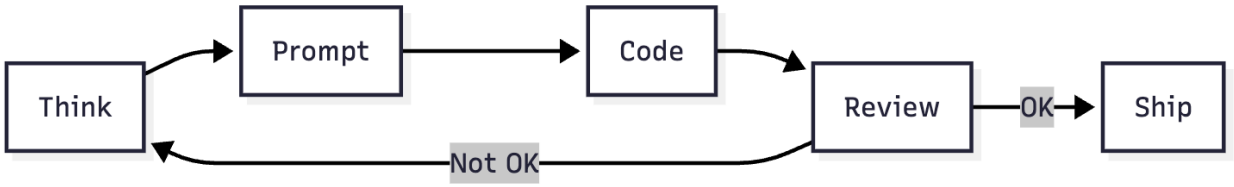
Think of this as writing for your future self and your future AI collaborator.

Tighten the Loop

Old workflow: Design → Code → Review → Test → Deploy → Monitor.



New workflow: Think → Prompt → Code → Review → Prompt again → Ship faster.



You might ask: where are Design, Test, Deploy and Monitor in the new flow? They're still there, but more blended into the process:

- **Design** happens during Think and Prompt, with AI helping shape ideas early.
- **Testing** blends into Code and Prompt, as AI suggests and runs test cases continuously.
- **Deploy and Monitor** are rolled into "Ship", once a solution passes review, AI helps push it live and monitor it right away.

These stages haven’t disappeared, they’re just no longer siloed. They happen earlier, more often, and more automatically.

This means:

- Shorter feedback cycles.
- More experimentation.
- Smaller Merge Requests, reviewed faster with AI-assisted coding.

In product development, time-to-insight becomes more important than time-to-code.

Rethink Team Planning

Instead of estimating “how long will it take to code this”, ask:

- “How long to align on the solution?”
- “How do we co-build this with AI?”
- “Where do we still need deep human judgment?”

Time saved in coding should be reinvested in better testing, tighter design, and learning.

Welcome to the new engineering workflow. It’s still your craft, but now with a jetpack.

Chapter 7: How much faster can we really go?

Right after the emergence of all the AI coding tools such as GitHub Copilot then Cursor, I started getting the same question over and over:

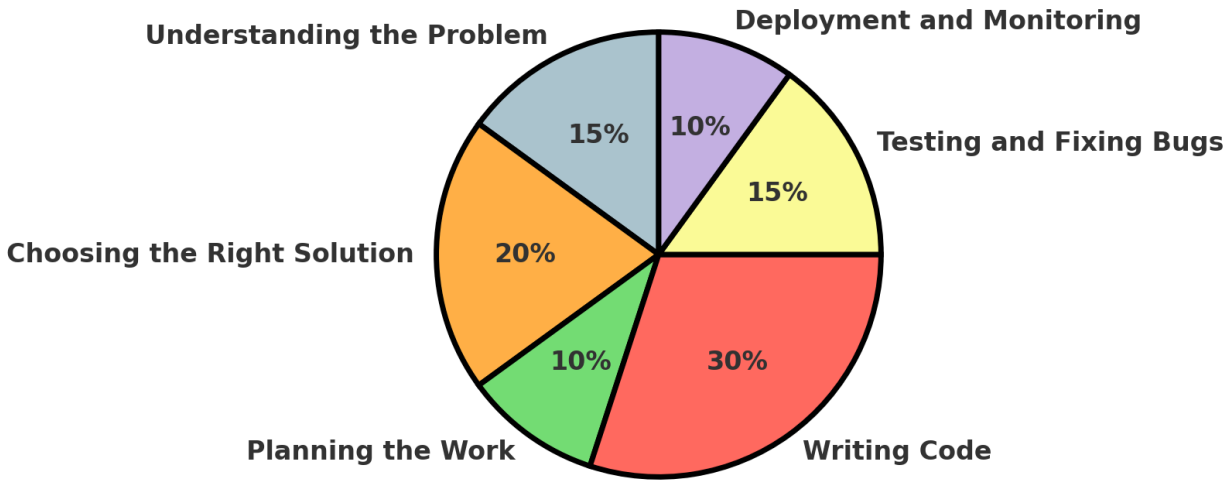
"So… does this mean engineers will be 10x faster now?"

Sometimes it was said with excitement. Other times, with concern. But no matter who asked, I gave the same answer: It depends.

And honestly, I think that’s still the most honest answer we can give.

Where AI actually speeds us up

Let’s recall the engineering workflow again:

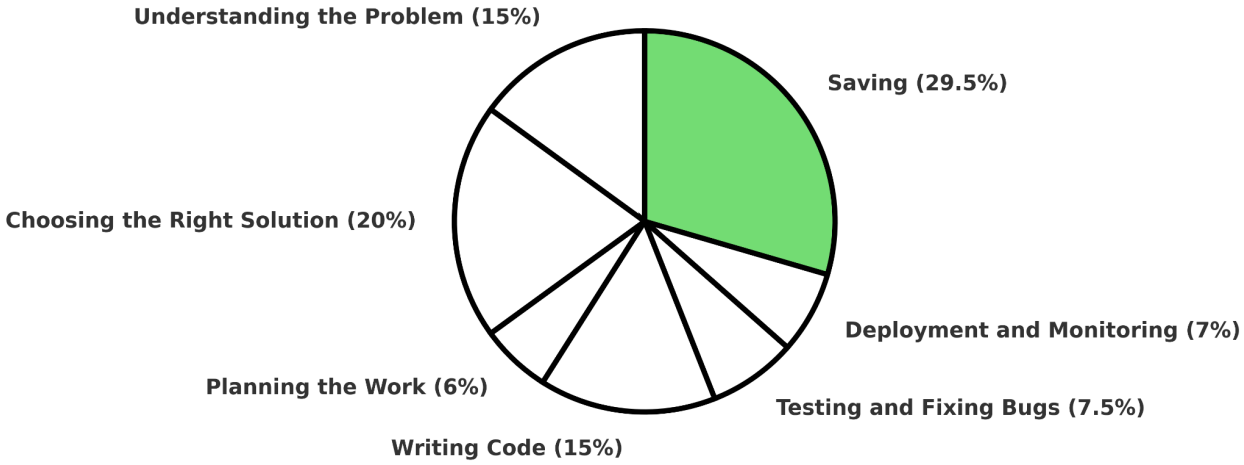


If you ask me to estimate how much time each phase takes, I’d say coding is maybe 30% of the job. AI can help with that, sure. But the other 70%? Still very human work, at least for now.

So when someone says, “AI will make us 10x faster” I like to ask: "At which part?"

Now let’s get practical. Based on what I shared, here’s where AI really helps:

- Planning (10%): AI can generate initial task breakdowns, fill in documentation templates, and even draft basic system design outlines. You still need to think, but the scaffolding is faster. Let's say a 40% time savings here.
- Coding (30%): With tools like GitHub Copilot, ChatGPT, Cursor, or Claude Code, Based on our team evaluation with medium size code base, it was around 50% speed-up, when used well. AI can scaffold code, handle boilerplate, and suggest fixes, but humans still need to review, structure, and decide.
- Testing (15%): AI can now generate test cases from API specs, simulate user interactions, and suggest likely breakpoints. We experienced 50% savings in writing automation end-to-end test.
- Deployment & Monitoring (10%): AI helps by analyzing logs and errors, alerting before things go wrong. Not perfect, but let's call it a 30% improvement.



If you do the math, you might end up with roughly 30% total time saved on a full engineering workflow, assuming your team knows how to use AI effectively.

That's not 10x. But 30% is no joke.

Let's put that into perspective.

If your project took 10 weeks before, now it might take 7. That's 3 extra weeks for polishing UX, cleaning up tech debt, or just shipping more. Multiply that across teams and quarters, and yes, that's real impact.

But only if you work differently.

You can’t just add AI and hope

Here’s the hard truth: AI won’t magically make your team faster just because you installed Copilot or Cursor. You need to change the way you work.

You need to change the way you work. Here’s how:

- **Rethink timelines:** When planning projects, start with the assumption that AI will assist, then reflect it in the delivery estimates. If your team isn’t seeing any time savings yet, ask why.
- **Level up AI usage:** Make AI part of your team’s daily workflow. Use it for stubbing out tests, writing documentation, translating requirements into code. Treat AI as a junior teammate, not just a code autocomplete.
- **Eliminate repetitive work:** Repetitive documentation, meeting summaries, incident reports? Automate them. If something boring happens twice, ask: Can AI do this next time?
- **Encourage AI-first thinking:** Before you start any task, ask: “How can AI help me here?”. Don’t wait for someone to tell you. Be curious. Try things.
- **Don’t forget the craft:** Speed is great, but speed without clarity or maintainability is tech debt waiting to explode. Fast code still needs to be good code. Vibe coding is not the excuse for bad code.

Let’s wrap this up with three key lessons:

- AI can make you faster, but only if you change how you work.
- Expect around 30% time savings, not 10x, unless you redefine the whole process.
- Use AI to eliminate boring tasks, but keep human judgment where it matters.

Speed isn’t about typing faster. It’s about thinking better. And that’s still your job.

Chapter 8: Rethinking Team Structure in the AI Era

Not long ago, one of our interns asked me during a coffee break, "Do you think AI means we'll need fewer engineers in the future?"

I paused, thought for a second, then replied, "Maybe not fewer, but definitely different".

It wasn't a deep conversation. But it stayed in my head. Because it reflected a bigger truth I'd been noticing across projects, planning meetings, and performance reviews: we're not just shifting tasks, we're shifting team dynamics.

We're not just adding AI to our toolbox. We're rewriting the assumptions about how many people you need to build and maintain software. It's time to rethink how teams are shaped.

The Old Model: Scaling with People

For decades, software teams scaled by headcount. More features? Hire more devs. More quality support? Add more QA. Need speed? Spin up a dedicated squad.

Most teams followed a familiar shape: a couple of senior engineers, a bunch of junior or mid-level engineers, and maybe a tech lead or architect steering the ship. The seniors made architectural decisions, reviewed merge requests, mentored the juniors, and unblocked problems. The juniors wrote a lot of the code.

It made sense. Coding was manual. You needed hands on keyboards.

But when AI can now scaffold entire features, write tests, and even summarize documentation, do you still need that same ratio?

AI isn't replacing the team, but it's reshaping it

AI won't replace software engineers. But it will absolutely change what your team looks like and what each person is responsible for.

Here's the emerging pattern I've seen:

- Instead of 2 senior and 4 juniors, you might have 2 seniors, 2 juniors, and AI doing the rest.
- The seniors aren't just reviewers, they're AI enablers. They guide the tooling, ensure quality, and stay responsible for long-term architecture.

- The juniors don’t write every line from scratch, they learn by prompting, reading, reviewing, and improving AI-generated output.
- The repetitive work? That’s AI’s job now.

In the example above, the team size was reduced by 33%, and it is real, we’ve already seen projects delivered faster, with better quality, and with fewer people when AI is integrated effectively into the workflow.

And the good thing is that it frees up the humans to focus on what humans are great at: judgment, creativity, and thinking ahead.

So, will AI take over all junior engineering jobs? No, not at all.

If we stop helping junior engineers grow, we’re giving up on the people who will become our future tech leads and problem solvers. AI can’t replace their potential to think deeply, solve complex issues, or make smart choices.

But what we expect from juniors is changing. We don’t need people just to write simple, repetitive code anymore. What we do need are juniors who are curious, who want to learn how to work with AI, who ask good questions, and who try to understand why a solution works, not just how to write it.

That means we need to change how we train them. Instead of just teaching syntax or how to follow instructions, we should help them understand how to think through problems, use AI tools wisely, and ask the right questions when something doesn’t make sense.

Their job is to learn, to get better with help from seniors, and to grow into engineers who know how to guide both code and AI tools in the right direction. The junior path is still here, but it’s not just about typing anymore. It’s about learning to think like an engineer in a new kind of team.

How to Rethink Your Team

If you’re leading a team or building one, here’s how to rethink your structure in the AI era:

Hire for Thinking, not Typing

Prioritize engineers who can evaluate trade-offs, guide AI tools, and adapt quickly.

It’s not about who can write the most lines of code, it’s who can make sense of AI’s output and steer it toward a solid solution.

Elevate Senior Engineers into AI Coaches

Your seniors should spend less time writing every piece of code, more time curating AI prompts, reviewing, mentoring, and setting standards.

Think of them as system architects and feedback loop maintainers.

Still invest in Juniors, but train them differently

Junior engineers still matter. But their job starts differently now. They'll need to learn how to think through problems first, then use AI to accelerate solutions.

Pair them with seniors to develop judgment. Let them correct AI output, debug edge cases, and gradually own more responsibility.

Reduce role silos with shared context

In the past, PMs wrote specs, engineers wrote code, and docs were scattered across Slack, Confluence, Jira, or maybe Notion.

Now, AI works better when everything lives in the same place, code, decisions, product goals. Everyone needs to collaborate around a shared repo.

That means PMs might create merge requests. Engineers might write documentation alongside code. Context isn't optional, it's fuel for your AI stack.

Let AI be part of the team

Give AI real tasks: drafting specs, generating tests, rewriting code to follow conventions.

Review its work just like you would a new hire.

Build workflows where AI shows up in code reviews, planning sessions, and retros.

In the AI era, great teams will be:

- **Smaller** because AI handles the repetitive work.
- **Smarter** because senior engineers spend more time thinking, coaching, and shaping the system.
- **More fluid** with tighter collaboration across roles, centered around shared tools and context.
- **More valuable** because they ship faster, maintain quality, and still grow talent.

You don’t need more people to build more software. You need the right people, using AI the right way.

Let the tools do what they’re good at. Let your team focus on what really matters.

Chapter 9: Rethinking how we work with Product

Not long ago, during a feature kickoff, the product manager proposed a solution and I paused to ask, “What specific user behavior are we trying to shift? What’s the pain we’ve observed?”

We ended up having a great conversation that led to digging into user feedback and support tickets. We kept asking question, then it became clear the initial idea didn’t really address the core problem. We had to take a step back, look at it from a different angle, then ended up with a simpler and much more effective approach.

That moment stuck with me. Good engineering isn't just about building what’s asked, it’s about helping the team think clearly. Sometimes that means pushing back, challenging assumptions, and guiding the conversation toward what truly solves the problem.

Now that AI can handle more of the busywork, it makes this kind of critical thinking even more important.

The Old Way

Traditionally, product and engineering collaborate a careful two-step:

- Product writes the spec.
- Engineering builds the feature.

It was transactional. Product defines the “what” and “why” engineering executes the “how”.

This split made sense when coding was time consuming and expensive, and when execution was the primary bottleneck.

But the truth is that it’s no longer the bottleneck.

With AI speeding up implementation, the slowest part of the loop isn’t writing the code. It’s understanding the problem, aligning on the solution, and keeping context from getting lost in translation.

The old way of working doesn't just slow us down, it fragments our thinking.

The Changes

AI changes the shape of the work. Engineers no longer need to wait for fully polished product specs. We can prototype ideas, generate scaffolding, and even explore trade-offs before a single meeting ends.

So what does that mean?

It means the walls between “product” and “engineering” need to come down.

In this new era:

- Engineers must become better product thinkers.
- Product managers must become more hands-on with tools and repositories.
- AI becomes a shared assistant, helping both sides move faster if they speak a common language.

This doesn’t mean PMs should write production code. But it does mean the merge request might come from the product side, not just the engineering one.

And that’s a good thing.

Because the faster we align on the “why” and “what” the better the “how” will be.

The New Collaboration

Here are some practical ways to rethink collaboration:

Work in the same repository

Keep product docs, specs, diagrams, and code in one place. It is your repo. This creates a shared context for both humans and AI tools. When AI can see the whole picture, its suggestions are smarter.

Start with Problem Statements

Don’t begin with features. Start with a clear articulation of the user problem. Engineers and AI tools are both much more effective when they know the “why”.

Create early Prototypes together

With AI tools, engineers can spin up wireframes, flows, or backend scaffolding in hours. Invite product teammates into the loop early to review and reshape.

Adopt an AI-First Mindset

Before breaking down the work, ask: “What can AI help with here?”

This includes generating user stories, test cases, even data queries. Product folks can drive this just as much as engineers.

Define the Guardrails for AI

Both need to agree on the level of quality, scalability, and maintainability expected from AI-generated output. Otherwise, you risk treating AI like magic and getting burned by shortcuts.

Encourage lightweight Merge Requests from PMs

Don’t gatekeep the repo. Empower product to propose changes, text updates, config tweaks, even mock JSON responses. Engineering still owns the final quality, but speed improves when everyone can contribute.

Shift from hand-offs to shared problem solving

Instead of a ticket that says “build X”, co-create the solution. Use Figma, Notion, Confluence or whatever tool your team prefers, but ensure you’re solving problems together, not passing them across walls.

In the AI era, speed comes not just from automation, but from speed of alignment.

Three things to remember:

- AI accelerates execution, so collaboration becomes the new bottleneck.
- Product and engineering must work in one loop, not two lanes.
- The best teams aren’t faster because they code faster. They’re faster because they think together from day one.

Chapter 10: What AI needs from you (and your organization)

The first time I tried using an AI assistant at work, I thought it would save us a lot of time. But instead, it just sat there, waiting for instructions. It reminded me of when a new teammate joins the team, smart, eager, but totally lost because no one explained how things work.

That's what AI looks like in many companies today.

People say, "Use AI to move faster!". But AI is like a new junior teammate. It doesn't know what's important, what's old, or what parts of your code are tricky. It doesn't know your goals, naming rules, or who to ask about that one unstable service. Unless you give it good information, it won't be able to help much.

If you want AI to help your team, you have to set it up for success first.

The real blocker isn't the model

Most companies aren't being held back by weak models. They're being held back by weak context.

It's tempting to believe that just plugging AI such as Copilot or ChatGPT into your workflows will instantly make your team faster, more productive, more competitive. But the models already work pretty well.

The real question is: are you giving them the environment they need to thrive?

Let me put it plainly. AI tools don't replace your systems. They rely on them.

- If your knowledge is scattered across Slack, Jira, Notion, and people's heads, AI won't find it.
- If your documentation is outdated or vague, AI will guess and often guess wrong.
- If your workflows are clunky or manual, AI can't magically fix them without guidance.

You wouldn't hire an engineer and then leave them blind. So why do we do that to AI?

AI is like a teammate who learns fast, never sleeps, and never complains but only if you onboard them well. That means shifting your mindset from "what can AI do?" to "what do we need to give AI so it can help us?"

This is where a lot of teams get stuck.

They try AI. It gives weird answers. They blame the tool. But in reality, it's often a data and context problem.

Here's the truth most people miss: **AI needs structure, clarity, and shared context to work well.** Just like humans do.

To make that happen, your organization has to start treating its information as a product not just stuff thrown into Confluence or dumped into code comments. That means:

- Centralizing decisions, docs, and code into one place.
- Using semantic structure such as clear labels, tags, linked data to make information meaningful.
- Asking: "can a new engineer (or AI) pick this up and understand it without a call?"

How to set AI up for success

You want real leverage from AI? You've got to build the foundation first. Here's how.

Centralize Your Context

Your knowledge should live where your work lives. That might be a well-structured monorepo, a code-aware documentation system, or a shared workspace tied to version control.

Write for the Reader (and the Robot)

AI can't read your mind. Neither can your teammates. Whether you're writing implementation plan, README files, or system design docs, be clear, be specific, and write like someone will read this later to make a change.

Add Semantic Layers

Some companies are moving away from using Saas and third-party tools so they can better control and organize their own data. They want to prepare it for AI use and maybe even fine-tune their own models. This shows how important good data structure has become in AI work.

Some already have strong systems for handling data. But most still use setups that only create reports for humans to read. These reports often remove small details and useful connections that AI needs to work well. We need to improve these systems so they also include meaning, context, and clear links between different pieces of data.

Structure matters. Metadata, naming conventions, tagging, and ontologies, these are the breadcrumbs AI uses to navigate your world. If you want better responses, give it better signals.

Measure What Matters

Don't just ask if AI is being used ask if it's moving the needle. Are we shipping faster? Are bugs caught earlier? Are devs spending more time on high-leverage work? Use real metrics: lead time, test coverage, review quality.

If you want AI to help, you need to:

- **Give it good context.** Clear inputs make for useful outputs.
- **Design your organization to be readable.** AI learns from your systems, make them clean.
- **Make structure the default.** Semantics > scattered scraps.

Set your AI up like you do for a new engineer: patiently, intentionally, and with clear expectations.

That's the work. And when you do it right, AI won't just help, you'll wonder how you ever worked without it.

Chapter 11: Staying relevant as an Engineer in the AI Era

A junior engineer once asked me, “Will AI take over our jobs?”

I paused for a second not because I didn’t know the answer, but because I wanted to say it in a way that would actually land. “Not if you keep growing” I told them. “But if you stop learning, stop thinking, and just do what you're told then maybe.”

That might sound harsh. But it’s the truth.

This wave of AI won’t crash and take our jobs. But it will force us to evolve. The engineers who thrive won’t just be great at prompting, they’ll be the ones who know when to ignore the prompt and think for themselves.

For a long time, being a good engineer meant you could write clean code, ship working features, and maybe debug a tricky issue once in a while. If you were solid at your craft, kept up with your language of choice, and could learn a new framework now and then you were in good shape.

But that baseline is shifting fast.

AI can now help write code, create tests, and even build simple apps. It’s not flawless, but it’s good enough to change how things work. Skills that used to make you stand out like writing code quickly or knowing all the right syntax are now expected from everyone. And some of those tasks are starting to be done by AI automatically.

And that’s not a threat. It’s a wake-up call.

The engineers who will thrive in the AI era are not just the ones who can prompt well. They’re the ones who:

- Know how to spot edge cases the AI missed
- Make judgment calls when trade-offs aren’t black and white
- Debug messy, AI-generated code and refactor it for clarity
- Collaborate deeply with product, design, and even the AI tools themselves
- Keep asking, “What are we really solving here?”

You’re no longer just the one who types the code. You’re the one who sees the problem, frames it well, and guides both humans and machines toward a thoughtful solution.

So what does staying relevant actually look like in practice? Here are things I keep coming back to:

Master the Fundamentals

Don't skip the basics. Strong coding skills still matter not because you'll always write every line yourself, but because you need to review, understand, and improve what the AI gives you.

The feedback loop between you and the machine depends on your ability to see what's right and what's wrong.

Great at Debugging

AI code isn't always clean. It's often just "reasonable-sounding". Your debugging skills, your ability to trace problems, form hypotheses, test, and fix will be more valuable than ever.

Practice Problem-First Thinking

Start with the problem, not the solution. Don't wait for someone to hand you tickets, dig into the real problem. Ask better questions. Learn to break down ambiguity. That's the kind of thinking AI can't do well yet.

Work well with different teams

You'll need to work across roles more often. Product folks might submit Merge Requests. Designers might tweak code. AI might generate specs. Your job is to hold the technical bar while collaborating tightly. Communication, documentation, and shared ownership become core skills.

Experiment and Build With AI

Don't just read about AI, build with it. Try out tools like GitHub Copilot, ChatGPT, Cursor, or Claude Code. Use AI to write tests, generate documentation, or summarize bug reports. See what works, what doesn't, and how to guide it better. The more you experiment, the sharper your instincts will become.

AI as your build engine, new type of programming

AI isn't just a tool for completing your sentences or suggesting the next line of code. We need to see it as a new way of building software. Think bigger:

- Instead of writing every line of logic, you prompt the model to fetch and format data.
- You can build apps that orchestrate LLMs to handle user queries, data processing, and even reasoning.

The real potential of LLMs isn't just making old workflows faster, but it's about changing how we build things completely.

Let's use a simple example: a weather app.

- **Before AI:** An engineer finds a weather API, reads the docs, writes code to connect to it, pulls the data, and builds a UI.
- **With basic AI help:** The engineer uses an LLM to search the API, then uses it to write parts of the code faster, like parsing the API response. It saves time, but the process is still mostly the same.
- **With a new mindset:** The engineer can ask the LLM to gather and organize weather data directly. For example: "Give me temperature, humidity, wind speed, and rain chances for the inputted location in a structured format". The LLM could return a clean JSON with all that info, skipping the need to write low-level integration code, and engineer can focus more on making a good user experience.

This shows how LLMs can take on bigger roles, not just writing code, but more. That frees up engineers to focus on what makes their app unique. Treat AI as an **engine** you can build around. Prompting becomes programming. Human creativity and context still matter, but the AI gives us a new layer of abstraction and power. This is the new infrastructure for innovation.

Here's what to remember as the landscape shifts:

- **Coding isn't going away, it's just changing form.** Strong engineers will still be needed to review, reason, and decide.
- **AI won't replace you, but someone using AI better than you might.**
- **Stay curious. Stay sharp. Stay human.**

This is not the end of your career. It's the beginning of a new chapter, where the best engineers are thinkers, problem-solvers, and guides for the tools we now work alongside.

Chapter 12: You still own the craft

It’s not whether AI will replace engineers. It’s whether we’ll remember what makes a good engineer in the first place.

Over the past couple of years, we’ve seen a flood of AI tools. It's easy to feel like we’re entering a world where AI does everything. And sure, AI can do a lot.

But when you zoom out, you realize: none of these tools actually solve problems on their own.

They don’t talk to users.
They don’t understand trade-offs.
They don’t care about long-term impact.
That’s still our job.

The core of engineering hasn’t changed:

It’s about solving real problems in thoughtful ways. Coding is just one of the tools we use. AI is now another.

Let’s be honest, AI is fast. Sometimes too fast. It can generate a working function in seconds, but whether that solution fits into your system, scales well, or makes sense for your users? That’s still on you.

We’re no longer judged by how fast we type or how many lines of code we push. We’re judged by:

How well we think.
How we debug.
How we design.
How we decide.

In this new era, engineering becomes more like coaching. You guide the machine, give it context, and shape the output.

It’s like having a junior engineer who never sleeps, but still needs supervision.

Here’s what it means to truly own the craft now:

Use AI aggressively, but review it religiously.

Let it write the boilerplate. Let it draft test cases. Let it help you move faster. But never let it write your logic without checking. Your brain is still the compiler of judgment.

Sharpen your debugging and review skills.

The fastest way to spot AI mistakes is to deeply understand what “good” looks like.

Can you catch a subtle performance issue? Can you tell when something looks off even if the test passes?

That’s the skill set that matters now.

Get closer to the problem.

The real value isn’t in executing a feature request, it’s in understanding what’s behind it.

Why does this feature matter? What are the constraints? What are the options?

That’s where engineers shine.

Document everything in your workflow.

AI needs context. If you want it to help you tomorrow, you have to leave breadcrumbs today.

Whether it’s decisions, diagrams, or notes in your code, write it down. Future-you (and your AI assistant) will thank you.

Mentor and be mentored.

AI can speed up execution, but growth still comes from human interaction. Teach junior engineers how to think. Ask your peers to challenge your ideas. Talk out loud about the why behind your decisions. That’s the craft.

The best engineers of the future won’t just know how to use AI, they’ll know when not to.

So, as this era unfolds, don’t fear the tools. Learn them. Use them. Stretch them. Break them. But never forget: It’s your thinking, your judgment, your care that turns software into something worth shipping.

The craft is still yours. Own it.

Closing

If you've made it this far, thank you. Truly. Writing this book has been a way for me to make sense of how AI is reshaping our work, and how we, as engineers, founders, and builders, can shape it right back.

This isn't meant to be the final word. It's a conversation. A starting point. And your perspective matters.

Maybe you disagree with something I wrote. Maybe you've seen AI work differently on your team. Maybe you have a better metaphor, or a real story that proves, or challenges, or an idea.

Whatever it is, I'd love to hear it.

How You Can Help

- **Tell me what resonated.** Which chapter felt the most useful or surprising?
- **Tell me what didn't.** Was there something missing, or unclear?
- **Share your story.** How is AI changing your day-to-day as an engineer, leader, or builder?

You can reach me directly at my social account that I shared, or drop a comment wherever you found this book. I will read every message.

And if this book helped you, please consider sharing it with a teammate, a friend in tech, or someone just trying to figure out where they fit in this AI moment.

We all still own the craft. Let's keep building it together.

Regards,

Hoang Nguyen