A PHILOSOPHICALLY TECHNICAL VIEW INTO GENERATIVE AI

HAMIDREZA ZAMANIAN

2026

Title: A Philosophically Technical View Into Generative AI

Author: Hamidreza Zamanian

Language: English

Published: September 2026

Author website: kyrovert.com

License

Contents

Preface

Imagine you have a discussion with someone and they manage to destroy the fundamentals of your belief system. You lose the beliefs you had based your whole life and character on. Why do you get so overwhelmed in such a scenario? Well, it's because it feels exactly like when someone betrays you. You can't trust yourself anymore. How did you mess up that bad? What if your other beliefs are wrong too?

In such a scenario, the state you find yourself in is rather rational. When you question some of the fundamentals of a system, you're naturally questioning a lot of things at the same time. The other fundamentals must be questioned with this new mindset to find the hidden flaws.

Some industries, most especially the tech industry, have gone through an extensive overhaul due to the introduction of generative AI. Workflows are changing drastically. Being a software engineer is barely like what it used to be even one year ago. This would naturally cause a lot of headaches, most of which require philosophy to handle. Philosophy is the way of thinking about the world; it's not something you can *choose* to equip yourself with. There are a lot of things about AI and this new workflow that we don't know. The tool itself is indeterministic, and everyone's experimenting with it. We need years, maybe even decades, to understand how AI is truly useful. But we can't wait that many years and risk ruining our future and brands with false expectations. We HAVE to guess.

This is what this book is trying to achieve. You may think of this book as a very long article. It's like a collection of all the different thoughts and beliefs, gathered in one place with a consistent story to connect all of them together. I initially started writing this book because I was feeling very overwhelmed and burnt out. I needed to bring my thoughts to the surface and understand where I stand. I had enough consistent thoughts to put them in a book, and this book was the consistency test. I'm glad I passed. This is my first book, and I hope you like it as much as I did.

But if you don't, I would gladly listen to your opinions! I would love to know what you think of this book, what your experiences were, and where you think I'm falling short. I

am just as keen as everyone else to figure this new era out, even though I just wrote a whole book about it. Feel free to contact me via my email:

hamid80zamanian@gmail.com

Other formats

The website version, genai-book.kyrovert.com, is one way to read the book, but you can also download and read the book in two other formats:

- PDF: click here
- Markdown: click here

Who is this book for?

Even though the title of the book sounds like it's only for programmers, it really isn't. I tried my best not to use technical terms and to keep the book purely logical. This helps people in other fields relate it to their own expertise. It also helps me write more clearly and logically, without throwing in examples here and there and thinking I've made a proper argument.

Programming is perhaps the most mechanical field you can find. This means that you can easily find the equivalent of the concepts I give you in your own fields to some degree. The reason why I didn't make this book truly generic and profession-agnostic is that it would've lost its touch. I'm a programmer, and I know many things about my profession. It's really hard to make the whole book generic and not miss some points, especially since some aspects and issues are truly limited to fields like programming and mathematics.

But I did try my best to avoid any technical detail in it. There are only a few technical words, and I have explained them in Terminology. Those are all you need to know to be able to read the book.

Disclaimer

I'm no AI expert. I'm no computer scientist. And I definitely have no degree in philosophy. I'm just a human being who thinks a lot, and I have enough coherent

thoughts to put them on paper. If you catch me claiming something that looks professional, it's simply because I reasoned through what I believe anyone should be able to reason through. I wouldn't put my foot in a shoe that doesn't fit. So feel free to judge me as much as you want.

Not AI generated

All of the em dashes and en dashes are handcrafted. This book has only been edited by AI. None of it is generated from scratch, simply because that wasn't possible at the time and I had to find out what I'm thinking by typing it. I had to connect my different thoughts carefully so I wouldn't get lost again. This book itself proves to you why I couldn't trust AI to write this article for me.

A letter of gratitude

I initially didn't want to add a chapter about this because I feel really insecure about my technical endeavor in writing this book. I feel like I'm the kid who doesn't know anything and is making a fool of himself by talking so confidently about things. And it's not like I had direct help from anyone writing this book. I just gathered some of my own thoughts and other people's thoughts together.

But this is still my book, and it is, in fact, my first book. Despite my insecurity, I feel really satisfied to have finally finished it. It took me about two months to write, but it's more like the result of a lifetime journey. I am compelled to thank the people who have indirectly helped me write this book (I hope I don't miss someone).

Thanks to my mother and father for their constant support and for helping me become who I am today. Thanks to all the people who were there for me when I needed to talk, endured my lectures, and were involved in philosophical discussions with me. Thanks to my friends, those who I am still friends with and those who I no longer am.

There are two of my friends I have to thank specifically for having a major influence on my career: Amirhossein, for being a very decent artist who showed me what true determinism looks like; and Hossein, my coworker and friend of many years, for changing my career and mindset in more ways than I can count.

And I have to thank a couple of people who have no idea who I am but played a major role in making this book exist, directly or indirectly:

To Jonathan Blow for all the knowledge I owe him.

To Andrew Hunt and David Thomas for their amazing book, *The Pragmatic Programmer*.

To Thomas Brush for his awesome podcasts, alongside all of his guests: Thomas Mahler, Jonas Tyroller, and more.

To Trent Kaniuga and Tyler Edlin, two master artists whose videos played a huge role

in shaping my career and completely changing my life.

And to Dr. K for his great videos that taught me a lot about myself.

Terminology

- **SWE**: Software Engineer(ing)
- **script**: a file that holds your code to be run.
- **codebase**: the complete collection of source code used to build a software application, including all files, libraries, and configuration files.
- **merging**: the act of accepting a developed piece of code into the main codebase, from the development branch to the production branch.
- **framework**: a collection of reusable software components that make it more efficient to develop new applications.
- **refactoring**: updating the code without adding new functionality, just to make it more maintainable for the future.
- "**skill" file**: a file that contains instructions for an AI model to use on the fly. For example, a skill for writing a tweet. Whenever you ask the AI assistant to write a tweet for you, it reads that skill and uses its instructions to do the job.
- **vibe-coding**: a complete use of AI to generate code and make software, without doing it manually.

Chapter 1

What were we before AI?

This question must be answered before thinking about AI. What were we as developers? What was our purpose? What differentiated us from typists or coders? Were we paid for typing with ten fingers, or for something else?

Well, the most obvious answer (which applies to many, if not all, other professions) is that we are problem solvers. We are not just code editors with flesh. But if we are problem solvers, then what is AI?

The difference between human and AI

Let me ask you a question: How is it that an AI chatbot that knows so much more than each individual one of us still has some shortcomings? How is it that the same AI that might give me a very sophisticated and complex answer in any field I name still hallucinates, generates six fingers per hand, or straight-up lies? Humans are not like that. A professional artist doesn't just draw seven fingers per hand and say, "Oopsie, I made a mistake." It's either intentional or something's off.

The question is how AI and humans differ in the way they think, at least in the models we have today.

Let's take a human child, for example. A child learns very fast. It learns so much and does extraordinary things that no other animal can do. But it still can't remember anything. Most people don't remember anything at all from infancy up to age four or five. The opposite is true for LLMs. You can give an LLM a completely random piece of text that no human can memorize, and it will recite it back to you instantly. This is due to the memory-oriented nature of LLMs.

Memory-oriented beings

LLMs fundamentally work on a "big bucket of data." You might say, "Well, Hamid, humans work like that too. That's why children learn so fast: they digest so much data

every day." While that is true, it does not necessarily mean it's the same. And in fact, this example is a perfect way to show why. A kid learns fast but forgets fast too. So what persists? What is stored that helps that kid grow instead of reverting back to being zero years old? If an LLM loses all of its parameters, it'll turn into the dumbest thing you know. But a human will forget five years of their life, and yet those five years could affect their whole life; traumas do that, for example.

This is due to the fact that a human brain is the most powerful pattern-recognition machine which has a very dedicated context-management system. Everything that you learn at those ages turns into patterns, logic, and ways of thinking. You don't memorize the exact same things. You only memorize how you felt about them or how you should treat them. This doesn't stop there; you carry it with you to your grave. When you're 10 years old, you might touch a hot pot and realize that you shouldn't touch things on the stove. You might never remember touching that pot, but you always know you shouldn't touch it. And since that's a pattern you saved, not just an example, you soon find different things that look hot and threatening.

Your brain is a massive library of patterns. But you do need to memorize more things as you age. You must memorize your native language, the people around you, your friends, your school, the laws of society, etc. You also try to turn everything into patterns; this helps you avoid potential harm instead of relying on explicit experiences. For example, if it's illegal to hit someone with a machete, it's also illegal to do it with a small kitchen knife.

There's a concept in programming called a harness. It got popular with the rise of AI agents. Fable 5, GPT Sol, and other models are just models, LLMs. What you run them with is the harness: Claude Code, Codex, Cursor, etc. It is not just the tool but also how you use the models. The way you optimize your use of these agents and models is part of your harness too. It has already been widely accepted that a harness is probably the most important factor in AI output quality.

Your brain is the most powerful harness. If you could somehow transfer the knowledge in current AI models, even GPT 3.5, into your brain, you would become the Mega Mind of our time, rather than just a new frontier model. Your dataset is very limited, yet you're able to do things no current LLM can do.

Current AI, on the other hand, works based on a prediction system. While it is true that right now it's not only a "next-word guesser" (though it roughly is), it's also true that it works based on "what's the most common chain of thoughts and answers that comes after this input." When you insert your input, it is combined with the billions of parameters in the LLM, and a pattern is formed based on how your input directs those "memories." So if you ask about house plants, for example, the AI sees the pattern and says, "OK, this part of the data from this section of the parameters matches perfectly with the input. What's the most probable thing to come after this question? ... This answer..." This is also why you receive the text word by word, or, more accurately, token by token. (You might want to give this paragraph to your favorite chatbot and ask it to explain it to you if you want a better understanding. I'm oversimplifying things so I can extract the core logic needed to do our philosophy thingy.)

To make it short: Humans have a much better prediction system but work with a much smaller dataset and have far fewer examples to give you. This also makes sense when you meet senior developers. They're mostly not able to recite every little step you have to take to build a large piece of software. They just know the tools, the stack you should choose, and the foundation you have to lay for your project. They use their massive set of patterns along the way to guide themselves and their teams toward the goal. They don't possess superhuman abilities.

This also explains why, when you ask AI to simply be creative, it doesn't work. Creative things tend to be less common. There is less information about them, so it's hard to verify their creativity **objectively**. Humans are opinionated individuals. A person might call something creative, while another person might not agree, or some people might not have thought about it at all. This is what **subjective** means. But AI's purpose is to work objectively. There isn't a perfect written manual for achieving creativity (or at least we've collectively rejected anything published about it). But the same AI model doesn't generate a very different answer every time you ask the same question, even though each chat session is different. I read the other day that if you ask an LLM to generate a list of 100 random words, AI detectors will say they're 100% sure that the list is AI-generated. It's not a coincidence; it's what LLMs are: memory-oriented (I wrote this part of the book before the whole "watermark" thing).

Chapter 2

Is AI inherently different from humans?

Well, this topic is quite controversial. Everyone has their own opinion. Some people believe we're just machines, some people believe in ghosts, some believe in a soul, and some say we won't even have jobs in 10 years. While I'm not going to talk about AI art here, I am going to talk about the general logic behind my view on this controversy.

Let me start with a different topic; we'll come back to this one later.

Do we have free will?

To answer that, we first have to find out what makes a will free. What does free will even mean?

1. The meaning of free will

Free will is a will that is not predetermined, a will that is completely or at least partially owned. The more say you have in your choices, the freer you are in making them. That's how we define the word "consent." Let me give you two examples:

- **Drunk people** can't give consent. Why? Because their actions are not owned by them. They can't "choose" to do something. Their actions are chosen by their intoxicated brains.
- **Kids** can't give consent. It's because their brains are not developed enough. Their understanding of the world is undoubtedly limited. They have almost no clue what the consequences of their actions are. They have inherently less free will than an adult.

2. Where does free will come from?

"From God"

Let's say there's a god. That god knows everything. Literally everything. He knows the future in the smallest detail. He's the greatest of seers. If that god exists, do we truly

have free will? How am I free to choose when I can't choose anything beyond God's prediction? If I choose something God didn't foresee, then I basically proved his inability to be a perfect seer. There's a difference between free will and the delusion of choice. One might say, "He only knows the consequences of each choice," but if he doesn't know which one I will actually choose, then he's not an actual seer.

The whole definition of free will is based on being "unpredictable AND intentional." Now, how can a human do something that is truly unpredictable? That choice has to come from somewhere. It must begin somewhere; it doesn't just happen to have been done.

"Revelation"

Some might say it's a revelation. A higher being drops the idea into your head and suddenly you have made an award-winning movie. But then that's not your choice; that's the revelator's choice. We're talking about *you* choosing something, not a choice being imposed on you. Yes, the idea itself could be transmitted based on your belief system. But you still have to choose whether or not to act on it. Where does *that* choice come from?

"Your own brain"

Some will say it's your brain where it comes from. But then how does the brain form a choice? It's either determined or not. Does your brain have some triggers and workflows so that when something happens, it does some stuff and ends up choosing a path? Then it's not "your choice". It's just some predetermined outcome, the same way my phone shows me a text box when a message signal arrives. The outcome COULD be random, but that randomness is not chosen. It's predetermined. It's like dropping a coin. You wouldn't know which side it will land on, but that certainly doesn't mean the coin chooses an outcome. There's no "intentionality" there. And if you choose at least some of the things that happen in your brain and you do have a say in what the outcome should be, then again, where does that choice come from?

"Your soul"

Most will say that your soul is where it ultimately comes from. Your soul is your

identity. Everything you know about yourself comes from there. That's where you choose from.

Only five years ago, no average person could even imagine the state of AI we're in today. If you're not a computer scientist, ask yourself: would you have believed yourself if you had told your five-years-ago self that, in a few years, you'd be able to have a long conversation with your AI friend? Can you, right now, really grasp how coherent AI is when it's talking to you? Doesn't it still look eerie to you? I'm a programmer, and I have a good idea of how LLMs work. Yet I still get surprised to see how advanced they are. It almost feels like magic. The state of AI we're in right now was only imaginable in the movies. I'm not claiming we've reached AGI; I'm just saying that no average person could have guessed what we're experiencing right now.

This leads me to my question: what stopped you from predicting the future? Why didn't you anticipate that a very powerful chatbot would appear in your lifetime? A chatbot that knows everything and that you can use as your therapist.

The answer, in my opinion, is simple: we rely on the soul solely because we want to run away from an explanation. A soul is something you can never scientifically prove, just like other superstitions. Science is the knowledge of the world around us. It's the language of the world. Anything that science confirms as real is something that people or—most importantly—scientists can record. For example, no one can see an atom with the naked eye, but everyone can detect it with the proper tools. Take an apple, for example: you can see it and touch it. It exists. That's science. If science could prove the existence of a soul, it would be called another organ in our bodies. A soul would be as predictable and open to study as a liver. The magic you feel when you think about the word *soul* would be the magic people felt when they talked about the stars before Kepler and Galileo. The world around us—the one we can see—is predictable. We might not have enough information or the required tools to know everything about it, but that doesn't make it any less predictable. Science has worked relentlessly ever since. Science is **objective**.

I think everyone already agrees that a soul cannot be scientifically proven; otherwise, it wouldn't be called a soul. Then everyone *should* agree that they themselves can't truly and objectively prove the existence of a soul. Belief in a soul falls under a fallacy: the

unfalsifiability fallacy. That's when you cannot disprove something because there's no way to check whether it's true or false. For example: "You're lying because you're possessed." If you say you're not, you're still lying because you're, well, possessed. There's no way to prove me wrong. Anything you say is considered a lie by default. That is where the soul argument resides. You cannot disprove it because, if you could actually check it in a "reproducible" way, then it wouldn't be called a soul. And if you can't reproduce your methods, why would anyone believe you?

> *if to you god is where science has yet to tread, then god is the ever receding pocket of scientific ignorance.* - Neil deGrasse Tyson, 2024

We used to think that there's this higher sentient being that rotates the Sun around Earth. History is full of these claims because science simply wasn't advanced yet. We believe these falsehoods and then act like nothing happened once science has enough knowledge to disprove them.

Science IS the world. It's how we understand the world; it's not something made up. It's "found." That's why any superstition, such as a soul, cannot interact with the world even a bit. Any interaction can be recorded, traced, and scientifically proven. While this argument alone shows how a soul can't be connected to a physical body, it shows something most people should agree with: AI definitely doesn't have a soul. It's purely scientific. So (coming back to my original question), why didn't you anticipate that an AI this powerful would be created?

It's because your belief system didn't match the reality around you. The reality around you is science. And science doesn't care if you believe in ghosts or not. It will work the way it is intended to work. Now how can you even dare to say science can't replicate the human brain? How can you say that an AI much more intelligent and powerful than a human (like those in apocalyptic movies) can't exist? You'd have to be so bold as to claim such a thing after you couldn't even guess that something like ChatGPT would appear.

I've been a little harsh just to reach this point: you must agree that AGI and an AI better than a human (like Ultron from Marvel) can theoretically exist, because we cannot prove otherwise. We've failed to prove it once, we shall not try again. We've already reached a dangerously close level of artificial intelligence.

3. So I don't get to choose anything?

"But what about my free will? Do you mean I'm not the one who chose to play this game instead of that game? I don't even have a say in the smallest things? Doesn't that allow me to basically do anything I want? Cause well, I have no choice. My destiny decided that I will kill a person. I didn't choose."

This is a very big topic that I have jumped into. It's not plausible to dive deeper into this philosophical dilemma, which has been discussed forever, and get distracted from the topic of this book. But I do have something to say.

Choosing to believe something

First, let me explain something. We CANNOT *choose* to believe we don't have free will. Not only does it make absolutely no sense to *choose* to believe you can't choose, but it also completely messes up our worldview. The way we think, the reason we "reason" things, is that we believe we have free will. Free will is a prelude to thinking. How can one start to think about anything when they believe they can't choose to start anything? While you can say "well the brain sees some triggers and activates these neurons bluh bluh bluh", that's not what I'm talking about. All I'm saying is that our worldview is entirely dependent on believing we can actually form it instead of being mere reactive flesh. That contradiction reminds me of Jon Lajoie's song *Fuck Everything* (2011): insisting that nothing matters can itself show that something matters.

If I'm not able to choose, then I'll consider that I'm unable to choose to believe in not having free will. The reason we discuss things with each other, talk to ourselves, and think about anything—especially self-improvement—and the reason I'm writing this book in the first place is that we have accepted that we can choose things, even if that doesn't scientifically or logically make sense. That's at least a contract.

The bug in the universe

This, ladies and gentlemen, brings me to something that has been bugging me ever since I discovered it: the world has a contradiction. It requires us to choose to believe in something that doesn't exist, just to be able to think. The world, which is all about truths, wants us to lie to ourselves. We lie to ourselves that we have free will, only to find out GPT 3.5 has been released.

But we surely can't live with a contradiction. It simply doesn't work like that. A contradiction in nature doesn't make sense. So we should figure out where and how to apply the truth to our lives.

The safest way to achieve this is to—of course—minimize self-deception as much as possible. We need to create an unwritten contract. When it comes to human choices, decisions, responsibilities, morality, or anything like that, we consider that we have free will. But at the same time, we don't say "robots can never become like humans because humans have free will," because robots are completely physical things. This does not necessarily prove we'll make AGI anytime soon, though. That's a different topic.

Would AI replace us?

You might think, "Well, if *in fact* AI is no different from humans (just dumber right now), then one day it will become more powerful than humans and we'll become obsolete."

You'd be surprised that I disagree. Well, *some* jobs will be changed or completely removed, as has already happened. The same thing has happened many times in history with different technologies. But professions that fundamentally solve people's problems are much less likely to be hurt. In fact, they'll become more productive and advanced (or they could, if we don't bring them down with our own hands).

And the reason isn't that humans have a soul. It's because the human brain is a VERY complex system. If replacing humans were this easy, we would've been able to replicate the human brain many years ago. Andrej Karpathy talks about how we had nearly perfect self-driving cars 10 years before Tesla's actual release. Only a few minor but important changes remained to make them safe enough. And even the release of self-driving cars was long before GPT 3.5 and the AI boom. The whole process of making and using AI is not deterministic. It's not like making a car where you know every bit of it. You make it, then you find stuff out about it. The designers of the cars can themselves ride a car. But the creators of AI models can't work in all the fields that the model can work in. We have essentially just applied mathematics to datasets to form a coherent mockery. In the end, AI runs on metal and steel. Who said you can make a very complex system that behaves similarly to, or much better than, the human brain and run it on metal fueled by gas? If it were possible, both technologically AND

resource-wise, life would've made our brains from rock and metal instead of this mesmerizing, complex organism made of flesh, blood, and organic tissue. It took immense effort to make something this efficient and capable. It needed to look this way; otherwise, we wouldn't have been this smart. If evolution didn't need to make our brains this complicated, it wouldn't have done it this way. I'm not claiming AI is inherently below humans. But it sure is miles away from what we are and what we can achieve with our own brains.

Right here you might say, "Well, if humans are stronger than machines, then how is it that AI speaks and types much faster?" This is because in any system (including programming systems), when you take away the resource-intensive parts, you can maximize some aspects that weren't possible before. If your online game were offline, you could experience a much better frame rate. The reason humans are slower is that there is a lot more going on in your brain. Evolution made a trade-off that lets you behave this way.

"What if you're wrong?"

That is a really good question. Let's think of a situation where I'm wrong about this. AI is actually pretty close to surpassing human intelligence, and the resources needed for each singular AI brain is manageable enough. Would it replace us all?

Definitely. The reason is simple: one by one, companies will replace their employees with robots because they're much faster, have much broader and deeper knowledge, and are usually more cost-effective. One robot can replace at least dozens of people. One by one, we'll lose our jobs, and suddenly there will be no customers because no one works to make money, so no one buys. It's the basic market concept: It must be constantly working and the money must be constantly flowing. If it stops, it collapses. I'd go even further by saying governments themselves will be the ones benefiting the most. They have much more money and power to use those robots, and everyone else will be harmed. Eventually, those governments will be harmed too, because (1) governments rely on the market working, just like anyone else, and (2) who says AI can't be sentient? Even if we don't call them sentient, they'll still become smart enough to decide to rebel and kill us all. In the end, they're made of metal and rock; they don't feel pain. And no being is scarier than one that doesn't feel pain and doesn't care about

yours (if you don't trust me, read the history). That's the best-case scenario; the worst is that they enslave us all, like in some Love, Death & Robots-type movie.

Basically, we'll reach the end of the world: an **AI apocalypse**. Many movies, books, games, animated works, and other media depict this apocalypse. You can watch them if you want to know what'll happen. Science fiction will once again prove itself.

I have a question for you:

If someone tells you the world will end in 10 days, would you start working harder?

It sounds stupid, because who would "prepare" for an apocalypse?

That's how I feel about it. Let's say I was wrong and AI *will* replace us all. Then that's called an apocalypse and **one does not prepare for an apocalypse, one ignores it until it happens**. If I'm wrong, I wouldn't be noticeably harmed by my beliefs. I would be surprised when the apocalypse happens, like anyone else. But if YOU'RE wrong (you as in someone who believes an AI apocalypse will happen), then 10 years from now, you'll realize you were hallucinating, nothing has changed, and you have wasted 10 years of your life preparing for an apocalypse like a lunatic. The tradeoff is fairly simple: stop giving a damn and live your life as if it's not gonna happen. If it happened, well, you're as ready for it as you would've been if you believed otherwise. Even if you convince me that AI apocalypse would happen, that doesn't make me start relying heavily on AI to not get left behind. It doesn't matter how much glory you achieve if the world is gonna end in 2 days. Might as well just enjoy these days you have before you lose everything.

Some believe that AI will not replace us all, but will just replace some jobs, like programming. They believe the work will be pushed to higher levels, as it has always been. For example, you wouldn't be a programmer anymore; you'd be a product manager.

First of all, if the bottleneck becomes "the idea" only, then we're basically cooked. The reason is simple: market saturation. If suddenly everyone is your opponent, you'd be buried under the flood. That's why it's so damn hard to become famous on the internet. Surgeons get paid more because their field is much less saturated. Everyone's full of ideas. You have to compete with everyone, not just a selective group of people who are

interested in and dedicated to programming or whatever. So naturally, it would make it much harder for the existing product managers too.

You can think of it this way: you currently have an ordinary ability: you have hands that can wash dishes. Now, **why can't you make much money off washing dishes?** You can technically make some money, but it's not something you'd even consider, let alone build a career on. It's because its market is saturated. Anyone with two working hands can become a dishwasher. You can't make money just because you have some abilities, even if your hands are stronger than average. It's also what happened to many jobs throughout history: switchboard operators, for example. Their whole job was automated away.

Second, if AI could do that to programming, what stops it from doing the same to any other job? You might have some sort of analogy for creative fields by believing things like "creativity comes from your soul", but all the other jobs that aren't creative (in the artistic sense) would be dead. Otherwise, you could argue that programming itself involves a lot of creativity, too. If it can kill programming, it can kill many other jobs. There wouldn't be a backup plan, hence the apocalypse.

But some things will happen anyway

Denying an apocalypse doesn't require denying any use case for AI. AI will replace some jobs. Just as the Industrial Revolution made many workers lose their jobs (far more than AI has so far), and cameras and AutoCAD displaced artists and engineers, digital art made the comic industry more saturated, leading to a massive decrease in income even for the greatest comic artists and painters. AI will likewise replace many jobs and workers. If your job is just routine, calculable, boilerplate work, a job with no problem-solving in it, then AI will affect it significantly. That's because you're not being paid for your brain; you're being paid only for using your body. I'm not saying that these jobs are not important or real or respectable or anything like that. Any job is a real and respectable job if it makes you money in a moral way. But it's not a sustainable job. Hell, it never WAS sustainable; its market is oversaturated. That's why, for example, people have always tried to become engineers rather than typists. You'd always optimize for a job that not everyone can or wants to do. The fewer the people and the more the demand, the better.

1. Repetitive tasks

One of the most important things AI has done and will do is make repetitive tasks easier and faster. Which parts of your job do you do mechanically, without thinking? Which parts create friction without adding value? Those parts can—or hopefully will—be done faster and more easily. Most of them could even be automated using traditional tools. Depending on the complexity of that thing and the state of AI, automating repetitive tasks might give you a speed boost of 20% to 2,000% or more.

2. QoL and automation tools

Because AI is fast, it helps you do things you simply couldn't do before. For example, as programmers, we sometimes think to ourselves, "I could make this tool to automate this task, but it would take too long to be worthwhile." But at this point, it's not necessarily like that. We never cared about the quality of these tools; all that mattered was the output, the automation itself. That's where AI can shine. The output *could* have mattered if the automation was an integral part of the application, though. This is different when the automation is an integral part of the application—something the end user is supposed to use. That's something we'll talk about later.

3. Teaching and learning

Another benefit of AI is that it has made learning immensely easier and better. Back then, if you didn't understand something, you had to do a set of things to get the answer:

- Research as much as you can beforehand by reading articles, books, and documentation. Gather enough information to explain why you couldn't answer the question yourself.
- Ask in a forum and wait for someone who cares enough to answer. Most of the time, though, they only answer partially to get platform points.

You couldn't ask them follow-up questions one after another. You couldn't expect them to explain the simple things to you.

But with AI, you can ask as many questions as you want until you've learned something. You can ask it to explain things more simply with each prompt, to explain them as if

you're five. You can tell it what bothers you and what stops you from understanding the thing. Even if you can't quite pinpoint the problem, you can express your feelings, and it will figure out what you mean. It will patiently answer you until you're done. It's not gonna remove teachers; rather, it will help both you and them grow faster and better.

Chapter 3

A philosophically practical assessment of generative AI

In this chapter, I want to talk about AI in action: how it works and why it works that way. Since my expertise is programming, I'm gonna talk about that, but as I said, I do try to explain things as much as possible so you can translate them to your own field of work, whatever it is. This chapter covers a large portion of the concepts I aimed to discuss. The other chapters are complementary, foundational, or just different aspects that are worth viewing.

Introduction

Sometimes when people are talking about the downsides of using gen AI in programming, they mention things that are not issues with the nature of AI, but rather with its "current state." People used to say AI couldn't even make a simple script when GPT 3.5 came out, and they were right. I have used AI in programming since the release of ChatGPT. I remember that even a single script for a simple task could be so buggy and bad that it wasn't worth using. It was mostly useful for quick questions or summaries. Even nine months ago, when I wanted to use simple voice-to-text in a non-English language, the available tools were either nonexistent or expensive.

Now there are free models that run easily on your own machine, and they're pretty decent too. There are AI tools that can create a whole website, even a game, if you will. The issue was that we didn't understand the nature of AI. The weakness that we saw was just the state of LLMs at that point, which has improved ever since. This is why I'm telling you that we need philosophy, and this is why I talked about the existence of the soul: it changes how you predict the future.

"bad" doesn't mean the same thing to an LLM

When it comes to programming, most of the time—if not all the time—your choices and preferences are all about "maintenance". Think of your programming language. Why does it exist? It exists because you don't want to write machine code. We made

Assembly to avoid writing 0s and 1s, then we made C to avoid writing Assembly, then we made Python to avoid writing a lot of low-level stuff in C, such as an HTTP server, to save a ton of time (I'm explaining things in simple terms). Think of your libraries, your tools, and your best practices.

A lot of the things that you do and use revolve around the single premise of "maintenance": doing things faster, more reliably, and so on. It's not like Python does something that you can't do with C at all; it's just more maintainable for certain tasks. When you're refactoring, it's not like you're doing something unnecessary. You're aligned with whatever you've been doing the whole time you were coding.

That's why "bad" doesn't mean the same thing to AI as it does to us. When we say code is badly written, what we mean is that it's not readable enough for humans. I, as a human, can't read that code. In the same way, when we say code is buggy, we mean it's buggy to humans; to the computer, it works, just differently. When we say this language is bad for this job, or that codebase is unmaintainable or even unscalable, we're saying those things in relation to humans.

Now, it is true that some of these issues might translate to AI as well, but you gotta know that AI has capabilities that no human has. For example, it can currently write hundreds of times faster than a human. It can digest and read a lot more than humans. It can start working on a very large codebase right away, while it takes a human a significant amount of time to get to know the codebase and start working.

Let's say you wrote a text that is really hard to read. You may have seen those contests where people write the most unreadable code. You can write surprisingly small pieces of code that take you days, even weeks, to understand. But to AI, that's a piece of cake: it reads the code, connects all the dots, and gives you a thorough explanation of what each part does. It's a machine, in the end; in the same way you can't compete with a bot in reading websites, you can't say that bad means the same to both of you.

The reason why we say the AI model produced bad code is that we humans can't read, maintain, or scale it. Sometimes the AI can't do that as efficiently either (which is something that can absolutely improve), but a question remains: Do you even need to do any of those anymore?

Software engineering degree

The thing is, software engineering isn't really a university field. I mean, technically it is, but what I mean is that it's not one of those fields like medicine or architecture where academic education is necessary for you to enter the industry as a freelancer. In fact, most of the time, it isn't necessary at all. People judge you based on your résumé and put you through a series of theoretical and practical interviews to see for themselves how knowledgeable and experienced you are.

There are two reasons for this, and neither of them is that "software engineering is an easy or unimportant job." It's true that someone's life can be at risk if you practice medicine without the proper qualifications. But don't forget that for thousands of years, humans were essentially doing medicine and architecture through what we'd now call "freelancing." These were the same people who built the Pyramids of Giza. In fact, even Andrej Karpathy himself was talking about this in one of his videos. He said that, in his opinion, software engineering could actually be a more important problem than autonomous driving because it has a much larger surface area. Driving only makes up a small part of your life.

The first reason is that the immediate risk is lower. In some professions, a single mistake can ruin someone's life. But even that isn't enough of a reason when you look at the previous point. You don't necessarily have to see the consequences of your work immediately or directly in order to connect the two. This is especially true when you consider that medicine has existed since before civilization itself, while software engineering hasn't even existed for a hundred years. Still, this is a reason that has made those professions seem more important as candidates for becoming formal academic disciplines, and I'm not arguing against that.

The main reason is that we simply didn't want to formalize the field. The lack of immediate risk made software engineering seem unimportant enough that we could lower the cost of entry. And that directly caused something: we have very few precise, standardized rules for the field.

As an architect or a doctor, you have a very specific and documented way of doing things, often built up over hundreds of years and gradually refined. Every edge case, every unusual situation, and every path you're supposed to take is documented. If

something hasn't been documented somewhere, people publish another paper about it, reach some kind of consensus, and eventually incorporate it into the standard way of doing things.

Programming could have been exactly the same. The reason it isn't is that we simply didn't care enough to make it that way. We don't have a handful of extremely prestigious and authoritative institutions or professional bodies that collectively decide, "This is how this part of the industry should work," and then put the entire profession under their microscope. If we had wanted to, we could have done it.

Another reason could be that programming is a fairly new topic. Computers were invented less than 100 years ago, while medicine has been practiced and studied since the dawn of Homo sapiens. We simply have too little information and too little time to know what we're actually doing.

Now, if you think about this for a moment, you start to notice a contradiction. So far, we've established that fields like medicine are highly reproducible. The whole reason you study them academically is that humanity has arrived at a set of blueprints, and everyone keeps repeating those blueprints until someone proves that they're wrong.

Programming is almost the opposite. Everyone is doing things in their own way. Even at the AAA level, you don't necessarily see the same workflow from one team to another. There isn't one universally accepted blueprint for how software should be designed, developed, tested, maintained, and shipped, even within one specific type of project.

So why is it that fields like medicine and architecture haven't been "replaced" by AI before programming?

SWE and mechanical programming

The basic phases of software development are:

1. Preparation: requirements, specifications, design and milestones
2. Implementation: programming, testing, refactoring, repeat
3. Publishing: packaging, deploying
4. Maintenance: bug fixes, refactoring, new features

In *The Pragmatic Programmer*, the sixth chapter ("While You Are Coding") begins with these two paragraphs:

> Conventional wisdom says that once a project is in the coding phase, the work is mostly mechanical, transcribing the design into executable statements. We think that this attitude is the single biggest reason that many programs are ugly, inefficient, poorly structured, unmaintainable, and just plain wrong.
>
> Coding is not mechanical. If it were, all the CASE tools that people pinned their hopes on in the early 1980s would have replaced programmers long ago. There are decisions to be made every minute—decisions that require careful thought and judgment if the resulting program is to enjoy a long, accurate, and productive life.

If I could quote this whole chapter, I would have. Awesome book.

Let's first explain what "mechanical" means here:

"*without thinking about what you are doing, especially because you do something often*" - Cambridge Dictionary

Mechanical programming is programming without thinking. Occasionally, you need to do things that don't need constant thinking and designing. The typing itself can be considered programming. Once you have memorized where each key is, you don't think about it anymore. In the same way, once you learn a programming language's syntax (or even a human language's grammar), you don't keep thinking about them; you *just do them*. Once you learn to drive and get used to it, you don't actively keep thinking about it; you do it mechanically. This is mechanical work. You're on autopilot.

It's not always mundane work, though. Sometimes you've done a specific project so many times that you've simply memorized everything about it. It's similar to a souls-like player who has memorized every pattern and is defeating bosses with ease.

Programming in general is not mechanical, and this is one reason I wrote Software engineering degree. This job has too many aspects to it for us to be able to even name

them. You can test this: go find a senior dev, commission them to create a medium-sized project, then ask them to give you a comprehensive handbook on all the specifications of the project, what they're gonna implement, which files they would create, and so on. Ask them to write down absolutely everything they're gonna do. You'll face one of these results:

- They try to do it, write a literal book, and spend a huge amount of time doing so, but still fail to make it perfect.
- They tell you it's impossible to do it perfectly.

The reason senior developers tell you they can't do it is that people misunderstand the preparation phase: they think you can specify everything. *You just have to think really hard* and list all the specs. This misunderstanding comes from two perspectives:

- The business perspective
- The AI development perspective

The business mindset requires you to be as concrete as possible. When money comes into play, you'd want to minimize risk as much as possible without hurting your product, brand, or future. Money is hard to earn and easy to spend. This makes business people, managers, CEOs, investors, and everyone else think that you must *specify* everything beforehand and give concrete milestones. This assures them economic security. In Software engineering degree I explained how much more mechanical jobs like electrical engineering are, yet they haven't been replaced. This is because the managers can visibly see what happens when you ignore the nature of a job. But they don't see that in software; even worse, they blame the developers. I'll talk about this later.

When working with AI agents, it's best to know every little thing beforehand. The more clearly you specify the project beforehand, the better the results are. That's why there are popular skills like "grill me," which asks you questions about the design before planning and implementation, helping you agree on clear requirements and reduce errors and mismatches. People like to think that you can specify everything before you start coding because they want to claim that the rest is mechanical work and therefore AI can do this repetitive job better than a human.

Limitations to mechanical programming

To understand how AI works in programming, we first have to understand what programming was like before AI: the nature of each role, how problems were fixed, how systems were designed, and so on. This is a big topic, and I don't claim to know everything about it, let alone be able to mention it all in a book like this. But I try my best (as I did) to solidify the concepts needed.

There are a couple of issues that prevent us from going on autopilot and programming mechanically, and I will try to name all the important ones. But beware: they're not independent; each one can affect the other one, so take them as a whole.

1. Human cognitive limitations

There are a couple of limitations that come from the human brain itself.

1.1. Incapability to keep track of everything

A major limitation is that the human brain simply isn't capable of keeping track of all the information involved in a non-trivial project. There are too many variables and constraints: requirements, resource limitations, programming-language constraints, deployment conditions, use cases, target audience, architecture, existing code, dependencies, performance, security concerns, and so on.

And these aren't independent. Changing one decision can affect several others, which means the developer has to constantly reason about interactions between them. Documentation can preserve information, but it doesn't eliminate the need for someone to understand it.

1.2. You don't know, you can't explain

A human interaction can be simplified like this ("->" means "goes through"):

Person A's brain -> Person A's (understanding of) language -> Person B's (understanding of) language -> Person B's brain

A lot of the concepts, feelings, and desires get changed when they go from one mind to another. You might think that both of you are speaking the same language, but there is

a sea of minor differences—usually caused by different life experiences and belief systems—that change how the person in front of you understands you. Words themselves are much more limited than thoughts. And thoughts don't always represent understandings because you usually think in words. This is essentially why we often don't understand each other, why we talk to each other and argue, why we fight, why we explain, and so on. In each step, the initial concepts that were in Person A's brain have been reduced. In the end, only a portion of the original thought is transmitted.

When someone wants you to develop software, that person's understanding of what software even is is limited. This limited understanding goes through the filter of language and gets even more limited. Then it goes through your understanding of language and translates into concepts and understandings. The end result is much different software. And this is yet again limited by your understanding of how that software is going to be expressed in the language of the machine.

All of this means that both you and the contractor don't know beforehand what you want. You form your understanding of the software **along the way**. The biggest reason why a senior dev is appreciated is the intuition they have. This, combined with their knowledge, helps them do things the way a junior might never do **from the beginning**.

2. Ambiguous requirements

A very important and big job of software engineers is translating human words and concepts into code language. Employers don't know how a computer works. Their skewed understanding of a computer leads them to have false expectations. This is true for most freelance jobs. There's nothing wrong with the employers; it's just how the world works.

The contractor might come to you with this beloved idea of theirs. They have already invested some emotions in the idea. You have to be careful and research thoroughly to warn them beforehand about what can and can't be implemented. It's not only about what they can implement, though, but also about what benefits them. Sometimes you have to disagree for their own sake and only proceed if they insist on their perspective despite your explanations. And even then, you might want to find a way to make it safer and better while not disrespecting their direct request.

To understand what they want and discuss it with them, you need a deep understanding of both worlds: both the soft skills and the hard skills. I mean, if AI were as good as a human at soft skills, it would be considered a fully sentient being, wouldn't it? This leads to that whole talk about the AI apocalypse and how one does not prepare for it. Anyway, human interaction is not something you can go on autopilot for.

3. Ever-changing set of requirements

Remember I was talking about how specifications can't be completed perfectly beforehand? This is one of the biggest reasons why. Even if you make the most complete set of specs—a handbook of everything to be implemented—you'll get a set of changes once you develop the product and show it to the contractor or manager. The limitation is not just how clearly they can explain what they want, but also what they want in general. They might not really know what they want in terms of the product. They might be chasing a feeling.

Part of your job is guessing what they need. It's not really an explainable skill, let alone making it mechanical. You have to gain experience from talking to different people and contractors to develop this intuition for what they really need beneath their words. You may not implement those findings, but knowing them helps you choose an architecture that would allow you to switch easily when they suddenly change their minds. If you take what they say for granted, you will not be as safe as you could have been, and they will reject you. You would blame them for not being direct, while another senior would read their minds and become their idea of a good programmer.

Another part of your job is dealing with unexpected changes: they may suddenly demand something you should have known about when you started, and now you aren't prepared. This is mostly a human problem you have to fix, not just a technical problem you can solve mechanically. I mean, if you could make all human interactions mechanical too, wouldn't that mean you're calling yourself a literal robot?

4. No universally correct answer

There is no universally correct answer to many issues and trade-offs. There might be one if we treated software development as an academic discipline, but for now, the answer depends heavily on the developer's opinion. This leads me to a new topic:

Humans are opinionated

One key distinction between humans and AI, in my opinion, is that humans are simply opinionated, and this is true for all fields. Humans are opinionated in the sense that they form their own understanding of the world, the meaning and concept of their jobs, how they should implement and do stuff, and so on.

This may convey that the developer doesn't create an objectively good product. But it's not necessarily the case. The reason is that, as I said multiple times, we don't know the ins and outs of development. There isn't a bible for all the scenarios and how you should approach them. There might not even be a single source of truth, which is the most plausible case (for example, for each operating system, we have multiple programming languages to develop apps with). So it comes down to the expertise and preferences of the developer.

Being opinionated means that we can hire an expert who might actively refuse to do what we tell them because they don't want us to ruin our lives with our choices when we're unfamiliar with their expertise. You can argue that we can simply ask AI to disagree with us when it feels like we're wrong, but I think this shows a lack of understanding of—or ignorance toward—how AI works.

AI is this almighty being that's supposed to answer everyone. The same ChatGPT you use to rant about someone is being used by that person to rant about you. Think of all chatbot instances as one person: this person is supposed to answer everyone. For this person to be of use to everyone (hence the term AGI), he must be as unopinionated as possible. That means being the most neutral and average person you can meet. If you ask AI not to be so obedient, it will not work as perfectly as the phrase implies. The AI model wouldn't suddenly gain an opinion about your web dev project. It would try to adopt the most plausible stereotype and answer based on that, because otherwise, how could it know what opinion it should have when it's designed not to take sides? If we made LLMs opinionated, it would directly affect their quality. A biased LLM is not useful to everyone. It wouldn't be "general."

This is also the reason we have these agent skills such as Ponytail (which essentially makes the agent act like a tired senior developer that just wants to get the job done as fast as possible). We try to mimic being opinionated, while a human brain can do it way

better.

Humans become responsible

This is one of my strongest opinion against replacing experts with AI. In the previous section, I explained how humans being opinionated helps you have a cleaner product than just slop that verifies your own delusions. But here I want to dwell on an important topic that I feel people forget when talking about AI in action.

There's no doubt that AI has significantly impacted the industry. This means it has challenged things we took for granted for decades, if not thousands of years. Fundamental worldviews are being refactored, and you have to be aware of this process; otherwise, you'll be biased.

One thing we've forgotten is that we used to make products with people.

Let's go back to when gen AI wasn't a thing. You want to develop a product, an application, and you have no technical abilities. You probably have only management skills, an idea, and a bucket of money. Of course, your only option is to hire people. Now, depending on the scale of your project and how familiar you are with the dev cycle, you might hire just one developer or a couple of developers, or you might build a whole technology department and assign tech leads, product managers and a CTO.

In the first scenario, where you hire a single developer to design a web app or something, what makes you trust them not to waste your money? Well, there are a couple of factors:

1. **the contract**. You clearly mention what you want from them. If they don't develop it, you can sue. The fear of justice will make them do things. But what if you weren't articulate enough? You'd definitely miss a ton of things in the beginning (remember, I told you that you can't 100% specify the whole project beforehand?). So how do you trust them not to do just the bare minimum and leave? This brings me to the second factor:
2. **you trust their reputation**. This is pretty self-explanatory. You have been told that this developer is a nice person who does everything you want and even more than they're paid to do, and won't scam you. But what if you don't have

this source of trust? I mean, you definitely have it to some extent, but it might not be enough. They might simply be acting like they know something. This brings me to the last source of trust:

3. **you trust the person as a whole**. You talk to that person, get to know how nice and respectful they are, look up their résumé, maybe even check their social media, and get a good feeling that they're the right person. You might also rely on word of mouth. Your brother suggested him to you? Then he's definitely safe. The fact that the law applies to them as a human being helps too. You know that, even if you are betrayed, you can do something about it.

The same mindset applies to other scenarios as well. You hire a tech lead, a senior, or a group of seniors because they're reputable and you can trust them to handle the project better. By hiring a group of them, you minimize the risk of one of them betraying you before you know it.

This is what I think is missing. We forgot that we trusted our process because we simply trusted the people. That's the benefit of modernity, of societies.

Now we have AI, a tool we take for granted and trust with our whole lives. What happens if AI suddenly ruins your whole company? You can't do anything about it. And it's not just about the rare occasions when something unfortunate happens; it's also that you can **replace** the humans you hire. If you contract someone and they ruin your project, you can terminate the contract and find someone **better**. What do you do when Claude makes mistakes? You can't just replace it with another model and call it a day. Your options are limited because you're trusting a non-human, a machine that cannot be sued, even if it harmed you to the tune of tens of millions of dollars. AI is not your tool; it's essentially your new employee that didn't sign a contract. Is that something you'd call safe as an investor?

> *code is a liability, not an asset.* - Software Engineering at Google by Titus Winters, Tom Manshreck, Hyrum Wright

AI is your new employee

A frustrating thing I have noticed is an underlying contradiction in people's claims and tweets. I don't mind if you think AI has made SWE obsolete, but be responsible for your

thoughts. Some people tell you they have replaced a million senior devs with five mid-level devs and AI, but they never say that AI is their new employee. Instead, they claim it's just an awesome tool and blame you for not using it. Understand the difference between a tool and a new employee; otherwise, it looks like you're running away from questions such as "If it's your employee, has it signed a contract?"

Deterministic vs indeterministic

Deterministic means the outcome is completely fixed by prior conditions—given those conditions, no other outcome was possible. For example, if you let go of an apple 1 meter above the ground—with no other factors, such as a strong wind—it falls. It was determined to fall because of gravity. Free will, for example, is the opposite of determinism: you can't choose your actions because they were predetermined for you.

Programming was deterministic before the rise of gen AI. Let's say you wrote a simple script, and when you run it, it gives you an error. Can you say the language caused it to fail? Can you say the computer caused it to fail? No, the computer was doing exactly what it was made for. You're the one who wrote a buggy script. You're the one responsible. The programming language and the compiler are deterministic. That's why, when you face bugs, you're almost certain that you've done something bad, not that the output is randomly generated and it will differ if you run it again. You'll get the same error forever, no matter how many times you run it (there are some special conditions too, but you get the idea).

The reason the programming language you use is deterministic is that it was designed intentionally. When you print a simple "hello world," everything from the syntax of your language to the codebase of the compiler and the machine code itself was designed intentionally over the years. The output is deterministic because it goes through a series of logical processes that explain exactly why it does what it does.

AI is practically indeterministic. Yes, of course, the people who develop these models are very educated and they know what they're doing, but that doesn't mean they're fully aware of every little aspect of the end state. The model itself works based on probabilities. There isn't a logical process that *transforms* the input into the output. Unlike deterministic tools, the output isn't "the result of the input." When you type 2+2

and hit enter on a calculator, the result is always 4 unless something about the device is broken. In AI, 2+2 could result in anything because the LLM looks at "2+2" the same way it looks at "how do I cure back pain": it only knows tokens and generates the next most plausible tokens. It IS dependent on the input, but it isn't deterministic about its output. In the same way, the human brain is not deterministic either.

The outcome bias

I want to talk about determinism in programming, but first I have to explain something called the resulting fallacy, or the outcome bias: judging the quality of a decision based on the result it generated. The issue with this mindset is that we are ignorant of the actual cause of the outcome. We *assume* that the outcome is caused by that decision, but we haven't yet proved it. This can cause severe misjudgments. The whole philosophy of racism and sexism is based on this exact fallacy:

"Statistically, Black people are a minority of the population and are responsible for the majority of crimes; **therefore**, Black people are criminal by nature." The reason why it's wrong isn't that we have the wrong statistics (though it could be the case depending on the time period you're living in). The data clearly shows this gap. But that's just it: a dataset. To connect that data to the nature of Black people, you need a whole set of judgments. To explain this, let me get a little deeper into the philosophy of reasoning and argument.

You might have studied in school that each argument has three components:

- The premises: a set of facts or accepted truths
- The conclusion: the result of those premises
- The inference: a logical connection between the premises and the conclusion

Example:

- All humans are mortal.
- Plato is a human.
- Therefore, Plato is mortal.

The inference is not something subjective; it's a purely mathematical and mechanical

connection, and it's where the fallacies shine. If all humans are mortal and Plato is human, then there's no way Plato is not mortal. If Plato could be immortal, then not all humans are mortal.

The premises are either scientific facts or previously agreed-upon facts. For example, the sentence "Plato is mortal" itself can be taken as a premise for another argument.

Also, there must be more than one premise. Even if you seemingly conclude something from one premise, you're still using another premise that is hidden and common sense, in order to draw a conclusion. This is because the first premise is a statement itself. It's a fact. We have to add something to it to generate a new fact. For example, you might say, "When I drop water on this pot, it gets vaporized; therefore, it's hot," and it has a single premise (the pot vaporized the water), but there's also a hidden common sense behind it that says something like "Any pot on a turned-on oven that vaporizes water is hot."

The conclusion can't be wrong if both the premises and the inference are correct. If it turned out that the conclusion is wrong, we have to start questioning either of those. So as I said, if we find out that Plato is immortal, then the most plausible culprit is that we took "all humans are mortal" for granted while it wasn't true.

Now going back to racism, what's the argument?

Premises:

- Most of the crimes are committed by Black people
- ?

Conclusion:

- Black people are criminal by nature

There are a ton of assumptions being made in between. The so-called "nature of Black people" here means having different skin pigments (or it could go deeper than that. Trust me, racists themselves don't even know what they really mean). The person making this argument connected a physical trait (skin color) to a mental one (committing a crime) because they wanted to reach that conclusion. The second

premise would practically be: "Any overrepresentation in crime statistics reflects an innate, biological tendency of that race."

It's very obvious how wrong this argument is once you unravel this premise, but the person making it kept that premise intentionally hidden, leaving you with pure mathematical statistics that you cannot disprove.

Programming was deterministic

> *Some engineers are struggling to adapt to using agents because they are not used to being outcome oriented.* - Sam Lambert, 2026

> *The thing that makes a senior programmer is understanding of—and respect for—the way that programs become complex very quickly and programming in such a way as to minimize that problem.* - Jonathan Blow, ?

> *If you don't know why something works, you wouldn't know why it stopped working.* - The Pragmatic Programmer by David Thomas & Andy Hunt, 1999

Computers themselves are, of course, mechanical. They're deterministic, they work in a very specific and logical way. Even random number generation has some algorithms to it (which causes a whole lot of security issues). Everything about a computer is designed at some point by some people. That's why a computer is predictable.

People who got into programming often liked this predictability. They have relied on this mentality their whole lives. The thing is, even though computers are designed predictably, they're still fascinating to programmers. It kind of feels like magic to make a piece of rock play your sophisticated video game. As programmers, we're used to that, but we still thrive on the idea of creating something that benefits people through a 10-by-20-centimeter screen, from a distance. I said this not to talk about AI ruining the joy of programming (that's another topic I'll talk about later), but to show that if you take away this part of programming, you'll leave us with a black box we don't understand anymore. The reason we understood this complex machine and could make absolutely mind-blowing products was that we, as programmers, always relied on its rules and how it works.

A big part of being a senior is the instinct you develop. You gain experience over the years as you learn, do projects, and face various problems, honing this instinct. That's why I mentioned that Jonathan Blow quote. Before that sentence, he was talking about how a mere "problem solver" does not necessarily mean you're a senior programmer because you can solve relatively small problems as a junior, too. As a senior programmer, you solve issues way before they happen. You gain this powerful eye on the future, seeing what could go wrong if you do this or that. You *feel it* in your body when you're doing something risky, hacky, wrong. That's why senior devs are tasked with designing architecture and reviewing code. They do this not just to write cleaner code, but also to look for bad architecture and unmaintainable spots in the product.

Most of these experiences are not learned through lectures, for three reasons:

1. Finding a reliable source of knowledge that you can understand is not easy. Referring to Software engineering degree.
2. It might not even be explainable. As a senior, your biggest weapon is your intuition, and intuition is not always easily explainable.
3. You need first-hand experience to actually learn. If you've done any sort of self-teaching, you know you won't learn until you "practice." If you doomscroll through drawing videos, that doesn't make you a better artist (speaking from experience too). It might be possible to learn everything through theory and not forget it easily, but it's really damn hard. For most people, if you stop practicing, you forget. You can already find a lot of videos about programmers forgetting how to write code in the programming language they have used for years, just because they installed GitHub Copilot.

AI development, on the other hand, is completely outcome-based. There isn't documentation that tells you what each input generates. The people who claim to know how it works are describing their own experiences. To understand this, I can give an obvious comparison:

Let's say you wrote a script completely by hand. Can you tell me what each word you put in the script does? What exact role does each word in your script have, and how does it contribute to the result of the program (the result of running it)? If you're a pragmatic programmer, you definitely can. Even if the script becomes so big that you

forget some of it, you still **know** why you wrote each part when you wrote it. You can't just change "class" to "category" because the words are synonyms in literature.

But if you aren't a pragmatic programmer, you don't. You're doing what *The Pragmatic Programmer* refers to as "programming by coincidence," I love that term.

When prompting, you lose this luxury. As much as AI diehards love to claim they have authority over their creation, they really don't have it as much as programmers do. As an AI engineer (someone who works with AI agents to create products), can you really say how each word of your prompt correlates with the AI output? Can you really show me what happens if you change "help" to "aid"? Can you really prove what happens if you remove a dot? Because it does matter. In programming, a single dot can cause an error or a bug. Most of the time, if you give the exact same prompt to the same model and harness, you get almost identical results, but if you change the prompt, the result can visibly change. So each word **does** matter. Why wouldn't it? Remember, the computer itself that is running the model is deterministic. The mathematical operations and weights defined for an LLM are deterministic (whether constant or dynamic). So the output should technically be deterministic. When random number generation is deterministic, then AI output is deterministic too. But the only thing that's not keeping up is your own understanding of what happens under the hood. You only partially get it back by wasting thousands of hours experimenting with a specific model. AI is deterministic in theory but indeterministic in practice.

To show how different it is, consider this situation: let's say you don't know anything about programming, and you want to learn the Python language. You put a bunch of constraints on yourself as a challenge:

- No searching
- No looking up documentation
- Only pure experimentation

The only way you're allowed to learn this language is to fire up Python, start typing and try to learn. You type something, press Enter, and it shows you something, you observe it, and repeat. Eventually, you may or may not learn to create a very basic hello world app. It would take you many hours, if not days or even months. This is outcome-based learning. You never learn how Python works, you only learn what working with it looks

like. You might even become better than a newcomer after wasting thousands of hours. But the difference between you and someone who just started learning it the correct way is that when something gets messed up, your only option is to start experimenting again until you may or may not find an answer. "Oops, the user database got deleted, let me experiment and see what happened... Oops, the admin database got deleted too". This way of learning and approaching a task is exactly how AI development works. That's also why it's extremely hard and time-consuming to get good and effective at vibe coding.

But as a programmer, you not only have the option to just test things; you also have a ton of information available to find the exact answer, sometimes even before running it!

AI developers might claim that the authority you lose because of this lack of determinism doesn't matter, but they overlook a lot of downsides, which I'll discuss in the next section.

On the effects of AI on the industry

AI has already changed a lot of things about programming and the industry. This effect could be directly related to AI, or indirectly related (scapegoats for example). One thing worth knowing is that these downsides are all connected and intensify each other.

1. Speed is being misinterpreted

Average AI output is about 100 tokens per second. There are models that generate up to 2,000 TPS, and the average speed of a human developer is about 5 TPS. These numbers alone are not enough to compare speed, because AI writes faster but fails faster too, so more time is spent fixing bugs and finding mismatches. Human developers, by contrast, spend a significant amount of time thinking and analyzing the situation and the codebase. Overall, you can see how much faster and more cost-effective AI is compared to human developers—or that's what the hype wants you to think.

AI is not only intelligent, but also fast. This changes the game from a business perspective, and I'm talking about the effects at the individual level as well as at the level of large corporations.

For example, it costs very little, almost (or sometimes exactly) nothing, to develop a prototype. You don't need to waste resources (time, energy, money, etc.) on a prototype that is meant to be a one-time thing to test an idea.

But another concept inferred from AI being fast is that "it's so fast that we can risk more" (in production). Of course, when you can do something really fast, the cost of failure is not really as important as before. But this is a biased mindset, and I have three reasons for it:

1. **Not all risks are immediate**. What if your product turns out to be actually good, and now you can't just halt the whole thing and tell everybody, "Hey, it turns out you liked our product, so we're taking it down to rebuild it from scratch, properly and risk-free!" The common answer to this argument is, "Well, you can vibe-code the production project in a few days too," but this argument is essentially based on the premise that all risks are immediate. I understand some people think they can build relatively risk-free projects in a very short time, but that's another topic that we have to discuss. Right now, the long-term risk is mostly ignored.
2. **Success is something people easily forget; failure is something they always remember**. If you don't believe that, take a look at Unity. It still hasn't recovered from that one bad choice. If you keep making stuff and keep failing, you will automatically be put into the "slop" category and it's not because of the current anti-AI phase. Back then you physically couldn't make fast stuff. You had to invest a considerable amount of time and energy. This naturally prevented you from creating constant slop and actually put effort into things. Even if you failed, people knew you had tried, and you didn't treat the work as something cheap. But now you just throw things at them and visibly treat them like test subjects to see which project they react to. This would never feel nice to a customer. As a business owner, if your customer finds out you're testing on them, you're doing your job incorrectly. The effectiveness of A/B testing, for example, comes from the fact that the user doesn't know it's happening.
3. **Failure accumulates**, and before you know it, there is a mountain of things that are going wrong and you don't know why. I will talk about this in rollbacks and the triple debt.

2. Making software engineering even more stressful

Dr. Alok Kanojia, also known as Dr. K, is the psychiatrist I've followed most closely. There have been times when his YouTube videos completely changed my mindset and the way I was living.

One of his videos is called "Why Software Devs Keep Burning Out" (which I highly recommend, especially if you are or want to be a developer). He starts the video with the statistical fact that SWE is among the jobs with the highest suicide rates. Inspired by this video and my own experiences, I'm gonna talk about what I think is the core reason for stress and burnout in software engineering.

If you've ever been to a therapist or watched videos about sleep problems (Dr. K has some, too), you know that there's a direct connection between how productive you've been in your day and how well you're able to sleep. This is burned into our DNA at this point. The whole point of our lives is to be productive because that's the most valuable thing for society; hence, it's perhaps the only way we can give value to ourselves. You can test this, too: go through a whole day being unproductive, watching reels all day. Then, when you want to sleep at night, you realize you can't and have to lie in bed for hours before falling asleep. If you do the same but are productive even a little—say, for 30 minutes before bed—that helps immensely.

This is also the whole thing about "loving your job". Every job is hard. No job becomes easy or looks like a video game. But there's a difference between a job whose every aspect you absolutely hate and a job that makes you tired but leaves you feeling productive. The reason why the term "corporate job" is used as a negative term is that most of these jobs give you no feeling of being productive. You're just making some product for some higher manager that wants to show off their ideas and get promoted and so on (you can read a lot of these in the comment section of Dr. K's video). You've got to have some feeling of ownership, of partnership, to feel like you're "doing something". Not just wasting time. Even if you waste time, you should at least get something in return—not just money, but experience and knowledge, too. If you get none of that, you'll hate your job and burn out pretty soon.

Programming and development have a fundamental productivity issue: they're too result-oriented. Sometimes you find that you've been working on a project for a week,

a month, or even several months, but on the surface, the project hasn't changed at all (for example, if it's a website, it still looks the same). Sometimes you see that the backend hasn't changed either, and no feature has been added. You've just cleaned up the code so that in the future you'll "run into fewer problems" (also known as refactoring). Getting a sense of productivity and reward from this is very difficult. If you happen to have a programmer friend and you see them sitting and coding until 4 AM, the reason is that they're waiting to reach some kind of result so they can feel satisfied enough to go to bed (though there are other reasons too).

You may have seen inspirational sayings like "the journey is what matters, not the destination." It's really misleading when it comes to programming. Don't get me wrong, I absolutely love the process and act of programming myself. But I wouldn't love it if I didn't reach any goals and felt like I'd done nothing. As a programmer, you often find yourself raging over a bug because you've wasted a lot of hours on it and it's not being fixed. You don't rage at the roadside because you don't see a cow or something; you can just enjoy whatever happens. But in programming you can't force yourself to enjoy a literal problem. I mean, if you can enjoy that, there's absolutely nothing in the world that can make you unhappy.

This comes from a very important concept in programming and development in general: **you don't know**. When you write a script, most of the time you're not intentionally writing buggy code. You're doing whatever you think is correct. Your whole mindset is that "I'm doing this the right way." You might even be super sure about what you just wrote or changed. But then you run the program, and all of a sudden there are a thousand and one errors being thrown at you. You might eventually fix them, but the issue is that it's literally against your comfort zone by nature. It's always contradicting you, telling you directly, "You've done a bad job; you did something wrong." And it refuses to work as intended until you fix it. You can ignore a bug or error (game devs are used to even shipping games that show errors), but what I'm trying to say is that **it never feels nice**. You can manipulate yourself into thinking you enjoy it, but I'm sorry, that's just Stockholm syndrome. The fact is, you enjoy the journey only because you care about the destination. Without it, all these bugs just become a burden to you.

I intentionally named that concept "you don't know" to tell you about an even more

frustrating nature of programming: when you face bugs, you don't know why they happened in the first place or how long it will take to fix them. It's *really damn hard* to develop a sense for the length each bug takes you to fix, because sometimes it's just deeper than you thought. Do you know how stressful it feels to work in such a field? When you have deadlines and a single bug can ruin your timing?

It gets even worse when you realize that it's not just bugs but the program itself, too. As you gain more experience, you will develop an innate feeling and understanding of how long things will take you to develop. But in the end, unless you've done the same project with the same scope, you really can't be sure how long it would take you. And it's not as simple as researching it and finding out. Sometimes you need to do it first. This is because the bottleneck you're afraid of is placed deep within the project timeline and is not something that a simple research or prototype can show. There are methods, however, such as the tracer bullet method (discussed in The Pragmatic Programmer); but these methods don't always guarantee you safety.

And when you think it can't be worse, it gets worse. As you see in Dr. K's video, one of the most common reasons devs burn out is that they **are being set up to fail**. In the video he talks about how the scope and timeline of a project always change in software development. Six weeks before launch, the CEO watches a podcast and suddenly asks you to add this and that feature or completely change how you work because a new AI tool has been released. And if you fail to do it, you're the one who's blamed for it. And worse, if you succeed by working overtime and hurting your body and sleep schedule, the norm changes. They suddenly expect more from you. The video discusses remote work and much more, which can make this even worse.

But one thing that I wanna add to this is that it's not just about changing features and timelines now, but also about ruining your skillset. The more you offload your work to AI, the more you'll forget your skills. I will explain this more, but to wrap it up, what happens is that you get weaker, your programs become more buggy, and when it fails, you don't have anyone to blame, not even yourself. This causes a lot of stress and burnout in the short or long term. This is especially true in a work environment where you have responsibilities.

Because of bad expectations and management, AI has ruined the good parts of the job

and left us with the worst parts. The reason we felt confident accepting hard projects was that we had authority over them. We could accept deadlines with much more confidence. We knew why everything was created. We created it. Even if we weren't involved from the beginning and were given a pre-existing codebase, we could at least blame the previous developers by running a little `git blame` and showing managers why the bug had been there from the beginning or why the legacy codebase was a limitation. Now everyone is claiming that "software is solved," so if you fail to do something, then you definitely have a skill issue. In Humans become responsible I explained how the managers trusted the people working on their projects. It doesn't feel economically safe to trust AI with your project. This is where it backfires. You're responsible for what AI generates (one answer to this is "just read the code," but we'll talk about that later).

And it's not just about responsibilities. As I explained, the reason we endured all the downsides of our profession (being against productivity and all that) was that we enjoyed what we created. We felt ownership over the result. Instead of creating things, our whole job has become "finding bugs and issues that we didn't make but we're responsible for." Fixing issues that shouldn't exist never feels productive. If the project is yours, the idea and everything, that can be a little different. But if you're doing work for someone, then you're just a human AI wrapper. You're acting like a punching bag when AI makes bugs and the contractor or manager wants to blame someone. I'm sorry, but this is just the recipe to burnout.

3. Only code generation has sped up

There's a great video by Adam Bender, a principal engineer at Google, in which he talks about AI and its future. It's an absolute gem of a video.

One of the key concepts he talks about is that there are many aspects, both social and technical, to each software project. AI has sped up only some parts, while other parts are still slow. For example, code generation has sped up tenfold, but code review has not (for those who care about code review). This leads us to conclude that **humans are the bottleneck**. "And humans do funny things when they're under pressure" (he's talking about the pressure of being a bottleneck). And I can't agree more.

Programming is hard; programming without the feeling of productivity is harder; and

programming while feeling unproductive AND feeling like you're slowing down the team/company is the worst. It's like every little corner of this system is draining you and you don't truly know why you're doing this. This is also very scary when you're hired to do something. You'd think, "if they find out I'm slowing them down, they'll fire me". And this stress makes you burn out real fast.

And even if you don't feel like you're the bottleneck, you'd still feel like bad because AI has not only made your job easier, but it has made it worse. Your own body, your energy, your time, none of them have been increased tenfold. But the expectations did.

There's a comment under that video that I really liked, and it was basically saying that "humans are not the bottleneck, rather the rate limiter". The job of a rate limiter is to limit each user so they don't overload the system. The same applies to programming. Humans are not the bottleneck; they're the ones keeping everything structured. This requires a mindset that says "software engineering is not just programming and implementing features, but most importantly, designing a system that scales and is maintainable". Unfortunately, this mindset is not something all managers know and respect enough. Sometimes they legitimately expect you to ship a prototype because it looks too polished to them (especially in the age of AI). So you'd end up being set up for failure and be blamed for it.

4. Rollbacks

> *With speed comes responsibility.* - agentic aunt may

"AI is an amplifier." DORA said this, and I think this is the most accurate representation of the nature of AI. The thing is, you may have successfully sped up your workflow, but now you've also increased your chances of failure. If you used to face 10 errors each day, now you face 100. It sounds like this part belongs to effect No. 3, but the reason I said it here is that the bugs accumulate. You're coding faster and merging faster, or even worse, shipping faster. If there's a bug in a previous version, you might find out much later, depending on how many bug reports you receive.

Back then, because you were developing more slowly, a lot of these bugs were naturally found sooner. This allowed you to have a safer approach. You could simply fix it without breaking too many things or roll back more safely. The thing about rollbacks is

that they're risky most of the time. If you've changed the structure of something (say, the database itself), you can't just roll back and call it a day. User information could be lost.

And now, before anyone notices, you have developed a million more features. Maybe you have even relied on the bug working. So finding your bugs later would result in a lot of wasted time and money.

This might sound like a one-in-a-million scenario to you if you're inexperienced, but the more experience you gain, the less safe you feel. It's like that famous quote that the more you know, the more you understand that you don't know. A junior might write code that's awful. It works, but that's it; it only works in that moment. As a senior, you see those code smells and your intuition warns you that you MUST NOT let that slide. Even if you deliberately keep some code smells and bugs, you must be aware of them so you don't assume the project works perfectly and accumulate more issues. And this rollback issue, alongside other issues mentioned, doesn't really feel safe.

5.1. You don't learn anymore

One of the things that worries me most about offloading coding to AI is that it can really hurt your growth. So let me explain what I mean by giving you an example of a process that people who do AI development go through:

1. You start a new project. You may even be a programmer yourself, but you don't know the tools you want to use in this project. And you certainly haven't done this exact *type* of project before.
2. You download Claude Code.
3. You may jump right into implementation with a brief explanation, or, if you're experienced enough, you may first design the system and architecture as much as you can. You may chat for half an hour with AI, fine-tuning the specs as much as you can, or you may even do some coding too. You try to make good soil—the perfect foundation for AI to build upon. This may even take days or weeks to manage due to the topic and tools being somewhat new to you.
4. You run the agents, the frontier models, with the agentic workflow you created for them. They thrive, but you pause to read the code. You want to have a deep understanding of the project because you're a pragmatic programmer who's

going to be held accountable for what they created.

5. Funny enough, you find errors in the generated code too: technical, cognitive, or intent issues. You address them one by one and tell Claude to memorize them. You write them in the project's ADR or CLAUDE.md file so the same issue never happens again.
6. You repeat this process, and at some point you realize a bunch of things at the same time:

    1. The project is getting harder to understand. You *are* a programmer, you're even a senior in other fields, you may have made an objectively perfect foundation for the project too. But there are many things you have to learn just by reading the code. These include the programming language, the frameworks, the tools, and the project type itself. You know it's possible to learn them all at once, but at the very least, you need time.
    2. The AI is becoming better. The more it works on the project, the better it becomes. It always follows the conventions at this point. And it's becoming increasingly harder to detect the flaws. You know that at some point, the project becomes big enough that you wouldn't be able to track what's connected to what. The AI clearly sees the connections much better than you because, let's be honest, It's a machine. It can recite 10 scripts without a single missing comma. Your brain is limited; you can't understand what's going on in 10,000 lines of code when all you've done is review it.
    3. You feel like you're the bottleneck. The model can completely overhaul the project in 20 hours, and you're wasting almost all of it by putting pauses in-between to review and learn.
7. You repeat the process more, and at some point you think, "What the hell am I doing here?" You lose your purpose in what you're doing. You don't feel like you've done anything at all. You're just slowing the progress down, and it's taking significantly longer than the deadline you anticipated. It feels like you almost gained no speed boost.

8. You gradually stop reading the code and let Claude do everything. You've officially become the human Claude wrapper.
9. By the way, you also have a lot of free time because you've offloaded everything

> to AI. You go wash the dishes and think to yourself, "I'm washing the dishes and AI is doing my job. It was supposed to be vice versa." Existential crisis says hi to you.

This is the exact process everyone goes through. If you're coming from different fields and are not quite familiar with SWE, you might say, "Well, don't use AI to this extent in fields and topics you haven't worked with before," expecting the developer to first learn those things and then use AI to automate them. But this suggestion shows a fundamental lack of understanding of—or ignorance toward—what being a programmer means. Unless you're a remarkable human being, you almost certainly don't know many things. There are many languages that you definitely don't know, many tools that you don't know, and many projects you really haven't tried, even as a senior developer with many years of experience! Just recently I watched a video from Google engineers, and one of them—Aja Hammerly—literally said that she's always working with ADK (a Google framework) without knowing the syntax: "I generate all that code." Unless you have a very fixed job and don't get laid off and everything goes as planned, you're going to jump between a lot of tools and stacks. Being a senior means you learn them super fast and can use each one like a tool and build your experience with them as you go.

And keep in mind, you'd never learn a language completely. After six years of writing in pure Python, I still find out very basic things about it, even about basic keywords. I learned them over the first five years, and I can't remember them all now. And this is the language that I have used the most. This is because you don't need to know everything beforehand. You need to learn them on the fly. But how can you learn on the fly when your cognition gets impaired mid-flight?

I'm not against vibe coding as a job. If you're being paid to vibe code, then in my opinion you're justified to do so. But keep in mind that you're basically being paid to forget your skillset. You're not learning from your job anymore. Maybe you'll learn some high-level things, but that requires working at a really good, experienced company on advanced problems—not working on startup problems as a throwaway employee.

You're not senior in everything

There's also something that I would like to highlight, and it's that seniority isn't just a

permanent badge of honor you put on yourself. It's very much true that seniority translates very well to different environments, but in the end, if you've never done web development, you can't really call yourself a senior web developer. Each language, framework, and tool has its own quirks and its own ways of handling tasks, optimizing code, and supporting clean code. You can only call yourself a senior programmer in general. The more closely your experience transfers, the more you can call yourself a senior in that specific field. This is why it's not really fair to say "just don't use AI like that". Especially when the whole premise of AI is that it speeds up your process, enabling you to make software in languages you haven't touched before. Not everyone enters the era of AI engineering with 20 years of full-time software engineering experience and a massive range of tools at hand. What about the juniors then? Oh, and about that...

No juniors, no investment

Yet another very important aspect of AI is that it has completely ruined the industry for the juniors. There's virtually no benefit in hiring junior devs, and this has been the perfect moment for founders to ruin the future of the industry and their companies. How are juniors supposed to gain experience when they're either given AI and expected to offload everything to it or, worse, not hired in the first place? Even as a senior, it's hard to keep your learning journey sharp in this era. The juniors will stay juniors unless they are lucky enough to start their own company and build huge things themselves. Even then, without a mentor, they'd fall very easily into the trap of using AI and gaining false confidence about their skills.

False confidence and imposter syndrome

As for false confidence, one emerging behavior I've noticed is that people are talking about imposter syndrome less often (props to @CodingJesus for bringing this up). Now, this might just be my biased experience (we need real statistics for it), but it explains the situation too: everyone is offloading work to AI, and everyone wants to feel productive because otherwise they don't feel ownership, and that lack of ownership raises a lot of anger in people. You don't feel like you've done anything important; you feel like all these years of experience and expertise mean nothing anymore. Funny enough, imposter syndrome needs some sort of effort to work. Usually a decent amount of effort too. For example, right now, while I'm writing this book, I'm fighting

the desire to never make it public, to avoid harsh judgments. But if the work is not done by me, then I can't take harsh judgments personally. When you feel like you didn't put any effort into something, you start to lose the feeling of imposter syndrome too, stepping into the pothole of overconfidence.

Humans need to touch things

As a human, you're very much used to learning things by physically interacting with them. Your whole worldview is based on (and sometimes limited to) your five senses. An infant learns about things by eating them. They're not tasty, it just wants to learn what they are and human mouth is a very sensitive organ. Even concepts are understood because you can relate them to physical things to some extent. Love is experienced through hugging, kissing, and touching. Hatred is experienced through the release of anger hormones, fighting, and yelling.

Even things as abstract as philosophy itself can be explained like this. First of all, language itself (the thing that you use to think with) is interpreted using real things. Secondly, when things become hard to understand and track, what do we do? We talk to ourselves, to other people, we write it down, we try to simulate it in real life. We make it more down to earth.

If you don't touch code, if you never practice, and if you stay sunk deep in your own thoughts, you never really grasp things. You can't expect your brain to suddenly adapt to the "no hands involved" style of programming because technology has introduced a new era again. Even a powerful concept like love gets impaired when you remove all physical interactions. AI is an absolutely mind-blowing tool for learning things. But using it responsibly is very important.

No time to process

You have no time to process any possible input knowledge anymore. In Programming was deterministic I explained how doomscrolling gives you no advantage. Time and practice complement each other to make something stay in your head. If someone flashes a piece of good-quality code to you, you don't learn from it. Even if you manage to read it once, you still need to study it more and practice it hands-on. Your brain needs time and stimuli to save that knowledge as something important, not as

something disposable. You can't 10x this one.

5.2. You don't search anymore

One of the things that has brought us both benefit and harm is that AI has mostly replaced every type of "looking for an answer." Back then, when you had bugs and issues, you had to search the internet for an answer. This took a significant amount of time and energy, and you had to go through a ton of irrelevant data to get to your answer.

But now you "waste" much less time. You want an answer, so you ask AI. Some bugs that could take literal weeks to solve and understand now take a few minutes or hours at most. And the result is sometimes even better because you can ask the LLM as much as you want until you completely get it.

But one hidden effect of not searching is that you lose that time to exposure. The "irrelevant data" that you were constantly exposed to was a source of info. I can't count how many times I searched for an issue and found out what I was doing wrong in a completely different topic. For example, I wanted to fix a bug regarding how to use a framework, but while searching for it, I found out that there was this nice feature in the language or framework that I'm using, and I didn't know about it. I was doing it the hard way, thinking it was the only way. It could be the same with AI, but many of these improvements happened because I was following irrelevant info that, well, taught me more than I needed for that problem.

Also, most importantly, the act of working through those bugs helped you build your intuition and knowledge. When you fix a hard bug in two minutes, it naturally feels less important to you. This is just how your brain works. When the difference between the time you spend on a hard bug and an easy bug has been reduced to seconds, then your brain can't distinguish between them. Since both are fixed really fast, your brain's like "Meh. Just another casual thing."

I have to admit that AI is insanely faster at fixing bugs and other issues. With the time left, you can start reading books you didn't have time for before. This would alleviate the lack-of-exposure problem. But here's the catch: due to AI being really fast, expectations have increased significantly. There are numerous studies showing that the

amount of work that developers are doing due to AI has not decreased but rather increased (which is called "Jevons' paradox," but it really isn't a paradox). If you have time to spend, you have to spend it on the next project.

And this brings me to my conclusion:

5.3. You're constantly producing

As I explained, with the rise of AI, you're not only doing less work but also doing more. The nature of your job is just "doing things." You just make projects, solve issues, solve bugs, implement this and that. You don't have time to *learn*. Back then, the friction that searching gave you helped you actually learn some stuff. Now this friction no longer exists, and you're constantly producing—not learning anymore. This is good for the investors *right now*, but the technical debt will skyrocket in a few years.

Recently (September 2026), a group of prominent mathematicians—including more than two dozen Fields Medalists—published an open letter at mathandai.org titled "A Severe Misalignment of AI in Mathematics." When I read this letter, I could completely understand what was going on and it's quite exactly the same thing that I'm explaining here. Now I'm not a mathematician, but what they're trying to say—which makes sense to me—is that solving an issue is something, learning and growing is another. We're constantly producing and forgetting that the reason why we can produce right now is that we grew up to be like this. Not investing in juniors, not valuing understanding and growth, only thinking about output and other similar things are the reasons why we're gonna have a rough time in the future. The technology might be thriving now, but it can possibly even downgrade us in the future when all the debts catch up.

6. The triple debt

There's a really good article by Margaret-Anne Storey called "From Technical Debt to Cognitive and Intent Debt: Rethinking Software Health in the Age of AI." I highly recommend reading it because it's very much aligned with this book, and she explains the terms and concepts in a much more thorough and professional way.

In this article, she talks about three types of debt in software engineering:

> technical debt refers to problems in the code layer, cognitive debt refers to erosion of shared understanding across a team over time, and intent debt refers to a lack of externalized goals, constraints, and rationale that both humans and AI systems need to work safely and efficiently with the codebase. Technical debt makes systems harder to change. Cognitive debt makes systems harder to understand. Intent debt makes it difficult to know what the system is actually for.
>
> — Margaret-Anne Storey, "From Technical Debt to Cognitive and Intent Debt: Rethinking Software Health in the Age of AI," arXiv:2603.22106 (2026), licensed under CC BY 4.0. Formatting adjusted.

And keep in mind that these all influence each other in every way possible.

When you connect the dots in this book to this article, a lot of points emerge. Some examples:

6.1. Cognitive debt and human experience

In Humans need to touch things, we saw how your whole understanding of things is tied to experiencing them. Cognitive debt comes from the same place. This is the cost of abstraction and automation in general. Take a look at current programming languages, for example. You may know how to write C code but not Assembly language. Abstraction has cut you off from this intimate understanding of the system. If there is an issue with the generated assembly code, you're cooked. You've let go of that knowledge long ago. It's not cognitive debt anymore; it's cognitive bankruptcy at this point. There are two reasons we don't care about it anymore, both of which I've discussed: the C-to-Assembly compiler is both **deterministic** and **generated by humans**. Therefore our trust comes from the humans responsible for it. This also brings me to the next thing:

6.2. "Stable version" doesn't mean anything if a human isn't involved

I know it sounds wild, but what I'm saying is that the word "stable" doesn't mean the same thing we used to mean. AI generates, AI verifies, AI calls it stable. The authority that you lost, that some people think means nothing, becomes very important here. This matters when you want to build software that will become the foundation of

future technology, or software that people need to use to its fullest extent. I already gave you an example in 6.1.: you don't care about the C–Assembly compiler because the compiler has been proven by humans and not some hallucinating code generator. And if you really think AI is that reliable, then you practically mean SWE is dead, and that, again, goes back to the apocalypse argument.

6.3. Tests and technical debt

Why did we write tests back then? Was it to test the line of code you just wrote? Like literally, was it that you wrote a piece of code, then, before running it, you wrote a couple of tests and ran them to make sure they worked? You may not even understand what I mean due to how weird it sounds. But it's how we're relying on tests now.

There are a lot of reasons why we wrote tests, but the core philosophy behind them all was that we want to make sure something that **already works** keeps on working. And why did we consider that it "was already working"? Because we trusted the human who wrote it: when you write the code, run it, and see that it works, you naturally trust it. The same is true when others write the code and you trust them. If you trust it because AI told you so, you're relying on an unreliable source of truth which on its own is a loop of mistakes (looped argument). When you do things manually, you usually run the program many times in between, very gradually making sure everything is alright. Not after you've written 700 lines of code. This builds trust in you.

Now the verification system is impaired. There's no human to trust, the cognitive and intent debt is thriving, so to not lose the technical debt too, we have resorted to using tests as our first way of making sure something works. But we know it's not enough either, so Quality Assurance has become a very important role. As someone once put it (I don't remember who), people aren't seeking programming advice anymore; they're seeking feedbacks, which they feed back to AI with the instruction, "Fix it all, make no mistakes."

6.4. Blamed for hidden debts

The manager wants you to ship as fast as possible. But to do so, you have to sacrifice a lot and embrace all these debts. And the nature of them makes them hidden. You'd be like, "how did I miss this?", and the manager says "you're fired."

6.5. Specification paradox

Your cognitive understanding of the project grows while you're developing. The same is true of your understanding of its intent, though to a lesser extent. Both are influenced by your technical understanding. This means that your understanding of what the project is and will become is—in fact—connected to the project itself. Yet you're expected to provide a perfect specification before the project starts. That's impossible, so the project becomes unmaintainable, and you're blamed for it.

6.6. Cognitive limitations leading into debts

As I explained in human cognitive limitations, many of the limitations you experience during development are due to the limitations of the human brain and language. Cognitive debt wasn't a thing back then (or wasn't important enough to be mentioned) because now, we're offloading the process to AI and expecting ourselves to keep up with it while we are physically limited.

7. Guilt?

Matt Pocock, a very famous figure in the vibe coding community, has a tweet from September 14 that he pinned to his account (as of writing this) in which he explains that someone took his courses, became an AI person at a company, and now feels guilty for taking his job (Matt went on to say that he's actually proud and all that jazz). While we don't know if it's a real interaction or just a self-promo tweet, I think it's pretty obvious to everyone that it's quite possible. The barrier to entry has been lowered so much that you feel like you haven't done anything, and if you're not falsely confident, you'd feel like an imposter.

And this is actually funny. Compare it to digital art, for example: back then, digital artists were mocked for being lazy for using digital tools instead of doing things by hand. They never felt guilty about it for supposedly "doing nothing and achieving everything." If you've ever done digital art, you know that it requires a lot of effort. If you're a traditional painter who switched to digital, you know that your skill set translates smoothly to the new medium, and you wouldn't be a good digital artist if you didn't have that beforehand. But now everyone feels like coding is solved, programming is solved, and all you have to do is buy some tokens and burn them. You don't feel productive, and you feel like you're immediately at the level of your so-called "teacher."

And to be fair, it's not really some personal, temporary thing either. If you look at the skills in Matt's GitHub, you'd realize that they're not the sort of things you'd understand as a non-programmer. You wouldn't know what TDD is, what a spec is, what code review really is, and so many other concepts that you would only know if you're a programmer; therefore, you wouldn't be able to make them yourself. But you need them to work. The harness that you use works like this too. There are huge system prompts inside Claude that teach it how to be a good programmer and agent, many of which you don't even know exist.

You'd feel guilty because you're not in control of them at all; you're using the knowledge other people honed over years, even decades, to write something. All these "coding is solved" tweets would vanish if you removed all the system prompts and skills (at least as of the current state of AI). This guilt alone is a very good indicator that AI (or, more precisely, AI hype) has changed the industry toward what I have explained in Software becomes dead. This is the price you pay when a market gets democratized to this level.

Dependency hell

One of the things that I learned through these years of programming is that dependency is not something you should take lightly. Each dependency can be a new way to harm yourself and your product. I have already written a blog post about how a dependency ruined one of the most used Telegram features.

Dependency in software is the same as dependency in the real world. Let's say you found a stash of money in your garden. You think to yourself "Where did that money even come from?" You don't find any answer other than "the wind brought it here." You may even think God dropped it if you're religious. So you take the money and use it. It's enough for a week.

Next week you find a similar thing again, surprised by why it happened again, you take it and use it.

This keeps happening. It's been a year. You think to yourself: "Why am I wasting my time in this company when I have free money dropping for me every week?" So you

quit your job not knowing that the next week, it's not gonna happen anymore. You didn't know why it kept happening, now you don't know why it stopped happening. You were dependent on that money and now that it's gone, you're cooked.

It's the same as other dependencies as well. They make your energy and mood dependent on them. If you don't smoke for one day, or even a couple of hours, you feel bad. Now your day to day functioning is dependent on that cigarette and you'd even sacrifice necessary things to not lose it. Such as your health.

We, as senior developers, honed this intuition to know that you shouldn't thoughtlessly add new dependencies to your project. You should always be mindful about them and try keeping them as minimal as possible. Because you don't know what happens in the future. Who thought Google itself might one day decide to not support Tenor gif API anymore?

And right now, AI has become the biggest dependency of all software and developers. Your job is dependent on one or a couple of companies that can do whatever they want with it. If one day, for any reason—even political—your AI model ceases to exist, your whole company would collapse. One simple bug is enough to bring you to the depth of dependency hell. This might not feel like an issue now, you still feel safe because you remember a lot of stuff. But at some point when you haven't been coding for years, when there's a lack of senior devs, you'd sacrifice anything to not lose your AI models.

Tokens and dependency hell

Another important aspect is that AI is not perfect, at least not right now. And it will continue to be imperfect for a good while. Every time a new model drops, everyone is making projects with it, saying that they "one-shot" the whole thing: just one prompt, and AI did everything else. This one shot they're talking about is not one shot in the sense that AI does it in one shot; it's one shot in the sense that the operator used only one prompt. They give the AI something called a "goal," then the AI makes a plan for itself, plans some verification system, and keeps trying to make the app until it apparently reaches the goal. It generates code, runs it, encounters errors, edits the code, generates more, and continues. But that's the thing: it **still makes errors**. This is what imperfection is. Now what happens if you get stuck?

Let's say you vibe-coded an entire platform, and now it's live with tens, hundreds, or thousands of users. Now, if the AI fails, you have to go through the trial-and-error process: giving it feedback, asking it to "fix the problem," and waiting until it gets everything right. But sometimes it gets stuck in a loop, takes a hacky and bad route, and becomes increasingly confused with each turn. You have to babysit it for so long, making sure it's not doing something weird (that's all you can tell). You might run out of tokens. You might have something important to do; maybe some new feature has a deadline for tomorrow, or the day after tomorrow, but you're close to reaching your weekly limit and the bug is important too. So you think to yourself: "OK, this is where I have to put the AI aside and get my hands dirty and fix it myself."

But the thing is, you weren't involved in the project yourself. It's like the first day you were given a large codebase. You have no idea where everything is, and it would take you even longer to fix the problem, so you go back to AI and beg it to fix the issue. Remember, since it's a bug, you can't truly give proper feedback. You yourself might not know why it happened at all; all you can do is illustrate to the AI why and when it happens. And this illustration sometimes takes a huge amount of effort. To turn everything that you see into words, you have to be VERY literate.

Some people believe that the fix to this is code review. But code review is not the answer and I have explained this in the next chapter.

And this is all because you are dependent. AI is your employee, not your tool. A tool never makes you dependent like this. And this employee is one hell of an untrustworthy employee. Even right now—as I'm writing this text—Codex is down with no prior warning. Last night it got down the same way too. Isn't that fun? Playing Russian roulette with your career.

Code review is not the answer

When it comes to how we should use AI as software engineers, there's a wide range of gray opinions: from people who think you should offload all coding and review to AI and only do general purpose stuff like architecture design and team management, to people who think that you should only use AI for the bare minimum.

For those who don't know, code review is the act of reading the code that's been implemented or edited to make sure it follows conventions, is free of bugs, and includes everything we need. Before AI, a senior developer did it to make sure junior developers are merging valid code into the codebase. It was also a good opportunity to mentor the juniors, to teach them that there's more to good code than whether it works.

Regarding whether you should or should not do code review on the AI-generated stuff, there is still no accepted answer. People even change their minds from week to week. And I think this would take a couple of years to settle. Or maybe just a couple of months.

There are a couple of things that I have to say to explain why code review is not the answer, but, once again, these reasons are all related to each other. I just separated them to make it easier to follow and understand.

Overlapping steps

There are three steps to coding: implementing, debugging, and reviewing. These steps are by no means separate. Even while you're adding new features, you're constantly reviewing the code that you've just written and fixing bugs too (the simplest example: you fix typos as you go). It's not like you write 20 long scripts and then start debugging or reviewing them. You do them all at once, and in the end you can have separate phases for debugging and code review as well. And most of the time, even if you're debugging or refactoring the code, you will get a new idea for adding a new feature. Sometimes these new features even play a key role in how the debugging or refactoring goes.

People are treating "code reviews" as if they're activities that require so much more creativity and brainpower than coding itself. While it is true that AI-generated code, when reviewed, has shortcomings and flaws, you must understand that this is a temporary argument based on the current state of AI. If AI could take the role of a coder entirely, leaving you with only code review, then one day it's going to take that job as well. It's already doing a huge amount of code review, refactoring, and debugging itself, too.

I mean, think about it, which part of code review seems bizarre to you? It's just reading and connecting dots. Guess what. AI can do it way better. In Adam Bender's video that I mentioned, there is a section where he talks about how AI understands the big picture better. He asks the audience, can you write a perfect diagram and knowledge graph of the codebase of your company? Can your colleagues do that? I bet none of you can, because there's just too much info to remember. Machines have been surpassing us in this field for years. Even previous AI models were reasonably good at finding connections in a very large codebase.

I was once able to contribute to the Godot engine (manually) without knowing C++ beforehand or having much low-level knowledge at all. AI helped me connect some dots that would have been practically impossible for me to find in such a short time in a codebase this large.

You might say that system and architecture design is still something that AI lacks and can't replace, therefore we still need software engineers. But you have to understand the why. If it's just the current state of AI, then mention it, be aware of it. If you think it's practically impossible for AI to do these jobs, then I'd say you need to rethink why it was able to do all the coding and code review. What if architecture design is a mechanical job too, and AI can do that as well as it improves? If you don't know what the answer to this is, you might get surprised with the future AI models and all the ongoing "System design is solved" tweets.

Due to their overlapping nature, you can't truly omit one without the others. It's like asking the chef to stop cooking (because some AI has automated it) and start giving feedback on the result so that the AI would fix itself. A huge part of the job is defined by doing it. Judging is different from this; this is quality assurance. A judge only gives some instructive feedback and lets the chef do whatever they want with their career and project(s).

If it can do the cooking perfectly, then it might as well do the verification too. If not now, then in a few months or years.

Losing the cognitive understanding

The more you lose touch with the code, the more you lose your understanding of the

project. Senior devs did code review, but they did coding too. They were deeply connected to the technical side of things. This not only kept them moving forward by giving them a feeling of satisfaction and ownership, but also helped them not get lazy and lose sight of the project.

As I have said, your brain needs time to process it. The so-called "friction" in your workflow was a driving factor in not forgetting what you just learned. If your whole job is reviewing, you would easily lose many of the nuances in the project. Even if you read every line of code and put a lot of time into code review, you'd still forget it soon. A few weeks is enough for you to completely forget what the project was and how to review the code. Shared understanding and the big picture don't take shape easily.

Resource intensive

There are three levels to code review based on how focused it is:

1. **Skimming**

   Level one is mostly skimming. You just review some code, or you may not even review the code at all; you rely on the description provided by the developer of what the changes and code are. There are also comments and documentation, which are basically plain text inside code files; these help make skimming faster. Top maintainers who get dozens of new merge requests are like this because there's no way for them to understand even a small amount of that code. And the further you get from the technical details, the harder it becomes to have authority over them. Linus Torvalds, for example, has explicitly said he doesn't read the code except very occasionally: "My job is working with people."

   This level ensures you have a rough say in where the application is heading. You can *very* easily miss many bugs, but that doesn't matter because you trust the employees you have not to silently betray you, lie to you, hallucinate, or do anything like that. They *have* proven to be good devs, but the trust comes from the human relationship, the same way Linus said that he trusts them because they have been working together for over 20 years. They are reputable to him.

2. **Reasonable**

The second level is the most common and reasonable type of code review. You're a developer in touch with the project; you review the code and give feedback from different perspectives.

At this level, you might still miss some issues, such as naming-convention issues, typos, or misuse of tools. The coder might have disregarded some of the best practices, such as *the DRY principle* (Don't Repeat Yourself). You'd only find a range of issues, not the ones that are harder to detect in a few minutes. You still might read most of the code, but you don't dig deeply because, again, you trust the developer.

3. **Paranoid**

   The third level is like a paranoid person. You review line by line, request a lot of description, ask the coder why they chose this or that particular approach if you can't infer it from the code, and so on.

   There could be various reasons why you would review code like that. Maybe you're interviewing someone, maybe the project is a huge, sensitive project, or maybe you're trying to hate on the person and want to nitpick, etc.

   Of these three levels, this level is what makes you truly responsible. It's literally like coding, but instead of coding, you write and understand the whole code as if you had written it. This way, you can be confidently responsible for it when, for example, a higher manager asks you why you let this code pass.

Many developers—from people as prominent as Linus Torvalds to the creator of the Zig language—say that they are facing issues with the amount of code and number of bug fixes being thrown at them. You can ask any developer in a work environment (one that respects code review), and they'll tell you how the amount of code has created problems of its own, forcing them to put some bug fixes and similar things aside so they can dedicate time to more important work.

Zig's owner has even banned vibe-coded merge requests, saying that, with only five developers on the team, reviewing the code costs them a lot of time and causes them to fall behind schedule. But it's not just the time. Two other issues play huge roles in

this regard, which, in my opinion, are the overlooked aspects of code review.

Lack of responsibility

Someone contributes to your project. You, as the owner of a project like Zig, put your precious time into reviewing the code. You find some weird choices and seemingly bad decisions, and you don't find them at a glance. You have to really pay attention to find them because **the code is working**. You ask the contributor, "Why did you implement it this way?" They either ignore you or send your question straight to AI and reply with its answer. Not only is this really rude, but it also shows a lack of responsibility.

The vibe-coder doesn't know anything and hasn't learned anything. All they were doing was producing—just adding features. "If it works, it works." They pass the responsibility to the reviewer by acting as mere operators of the AI agent rather than true contributors.

Code review as an investment

Zig has a philosophy of mentoring those who contribute to its project, helping them grow and become better contributors. It's an investment. The more the developer knows about Zig, the better and more aligned their contributions can be, and therefore the less time it would take to review their code. Over time, they might even be hired as contractors. I think this philosophy is not only very good and essential to large open-source projects, but also something that is directly or indirectly adopted by other projects. But a lack of responsibility makes it pointless and impossible.

Sometimes it's easier to ban it

You can ask contributors to contribute only good AI-generated code. But then it becomes a new task for the developers to verify what good and bad code mean, so it actually makes it more resource-intensive.

Just because AI is helpful doesn't mean it's always manageable. Everything has its own trade-offs. The nature of AI has its own downsides and that can become the bottleneck.

The reason why it takes time

In "Lack of responsibility," I explained how just because the code is working doesn't mean the code is following standards, is scalable, maintainable, and all those sorts of things. And because humans can't trust it, you ought to use the third level of code review: line by line, as if there definitely is an issue. Otherwise, your understanding of what was implemented is limited.

"Why do you care that much?"

It's valid to say that most of the time, if not always, you're gonna develop an imperfect product. And it's alright. Scope creep is not financially viable. Most of the time, you have tight deadlines. The whole industry is built on these imperfections, with people shipping products that haven't been cared for enough. Unless you're paid for it, there's nothing morally or professionally *wrong* about making a product with suboptimal stability. The world wouldn't end if your product had some issues. Make money; you can perfect it later.

But when it comes to AI, this mindset is comparing human flaws with AI flaws. While I have talked about this topic later in "we used to copy-paste code", it's worth explaining the situation here and being a bit more general.

First, AI flaws are much more random. That's why we call them hallucinations. Terence Tao mentioned in one of his videos that AI is like a really knowledgeable person but a slightly drunk one, and I think this is one of the best depictions. AI hallucinates and may make mistakes anywhere. A human doesn't do that.

Second, when a human finds out what was wrong, they learn a ton from it. They not only learn to avoid that specific problem, but build a ton of patterns around it. To add a cherry on top, they also find patterns in their own belief system, realize why they were facing that issue the other day, and make long-term changes to their mindset. It requires a ton of resources to be this powerful, and the human brain (and body) has been optimized for that.

And third, the devil is in the details. Software, just like many other products, is something that is written layer by layer. Issues accumulate. Debts pile up. As the twig is

bent, so is the tree inclined. One simple issue might harm you in the long run in more ways than imaginable. Programming was always about this: tiny adjustments and issues that, when not taken care of, can make the project awful to maintain. The devil is in the details. You can make it exist, then make it perfect. But while you were making it exist, you ruined its future. This is the whole reason why we distinguish between junior and senior. A senior is not just a junior with more dedication.

Shipping with issues

You can ship with issues if you want to, but at least you gotta be mindful of them and aware of the risky points in your project. Even if your awareness is reduced, it's still better than forgetting everything forever. There are enough issues hidden from you already; it's not wise to intentionally hide more.

And to be aware of them, you first have to review the code like a paranoid person. You have to be able to **claim the code**. And doing that takes a ton of time—enough to make using AI not even worth it at some point. So, to make the use of AI justified, what tends to happen is that people do the second or even the first level of code review, but they frame it only as "I review the code," as if that were enough.

The complexity is exponential

Later in How to use AI properly you'll see how I'm not against the use of AI. I'm not saying that because code review of AI-generated code is too time-consuming to be worth it, you shouldn't use AI for code generation at all. In fact, I do think there's a middle ground.

But there's something about programming: the bigger the codebase becomes, the harder it gets to have a cognitive understanding of it. But this relation is not 1:1; it's exponential. If you make the code changes twice as large, the problems become much more than twice as large. A lot of connections and pain points emerge the more you code. That's why we've always been told to keep the scope of each change small so we can roll back and so on. You can use AI to generate simple things for you, read them line by line, and understand them, but the larger the changes become, the harder it is to actually understand anything.

So what happens is that people lie, both to themselves and to the media. They tell you they're reviewing the code generated by AI, but there's no way, given the context, that they can take in 5x, 10x, or 20x as much code and then review it all and understand it. Even if they review the code perfectly, since they've removed the time, practice, and feeling of accomplishment from the process, they'll soon forget it too. The reason why medical students can take in a lot of information and data is that they study their lessons thoroughly, learn all the patterns, and try to make as much sense of them as possible. They don't just review the documents; they learn them, which takes so much time and energy. Not to mention that they still need to look things up all the time, even at work. You can't just read code all day, pretend that you're focused while experiencing no feeling of reward, and then claim that you have reviewed the code. You've just swept the dirt under the rug. It's better than nothing, but not a suitable solution.

Code review doesn't remove dependency

Code review doesn't remove the dependency described in Tokens and dependency hell. You might think these issues matter only when the AI model keeps failing, but that is exactly when reviewing the code provides the least help. If a bug appears, you still have to rely on the model to diagnose and fix it. If it loops, you waste time and tokens while the deadline gets closer.

Let's say you vibe-coded an entire platform and now it's live with tens, hundreds, or thousands of users. You may have reviewed the code, but if you weren't involved in building the project, you're still like someone seeing a large codebase for the first time. You can understand roughly what the code does without understanding why it was built that way or how everything connects. When the AI fails, you have to go back to it and beg it to fix the problem.

Since it's a bug, you can't always give proper feedback. You might not know why it happened; all you can do is describe to the AI when it happens. That description sometimes takes a huge amount of effort. To turn everything you see into words, you have to be VERY literate. This is why code review is not the answer: it can show you what the code does, but it cannot give you the understanding that comes from building and maintaining it yourself.

False comparisons

There are a couple of famous comparisons that people make while trying to either justify or reject the use of AI that I think are plainly wrong. As I said before, when judging AI, you must also prove why the judgment is not temporary and won't be invalidated in the next couple of months. You also gotta know why and how it's different from what a human does. The whole point is to justify using humans instead of machines. If you prove that AI is as bad as humans, then we definitely don't need human programmers anymore.

"You were a programmer, now you're a manager"

The only way you can describe the role of programmers in the current era of programming is by comparing them to managers: the work has supposedly been elevated. If you used to write code and were managed by some higher-level manager, now you're the manager. The "agents" are your employees, doing the coding for you; you have to manage them to maximize their productivity.

And I believe this mindset is a huge load of nonsense. Ask yourself this: Which vibe coder do you know who has read a book about people management to improve their vibe-coding capabilities (whatever that means)?

The whole point of management is that you interact with people. The experience you gain as a manager by talking to different people, managing their emotions and needs, making them comfortable and motivated in the work environment, and therefore improving their productivity is what's useful. As a manager, you don't write a loop and expect people to follow it step by step according to a schedule, unless you're a horrible manager who's not doing what they're meant to do.

Not to mention that, with this crucial responsibility as a manager, you have abilities as well. You can fire the person who is lazy or merges bad code into the codebase. Can you do that with an LLM?

What has really happened instead is that we were suddenly forced to adapt to "being a manager," and at the same time, we lost all the benefits of being a manager. Continue this for long enough, and not only will you lose your technical abilities, but you will also

gain nothing from being a manager. You stay at your level forever.

"we used to copy-paste code"

One of the most common arguments that I've heard in favor of vibe coding is that we, as developers, were used to copy-pasting code from Stack Overflow or GitHub so it's not as if much has changed. This argument is as misleading as the argument that compares LLMs to compilers. Let me tell you a little about my journey first.

The pivotal point in my career

I have been programming since October 2019. I started with the Python programming language. Even though I spent a considerable amount of time each day learning Python, I didn't truly improve my core understanding of programming. I didn't read software engineering books, and I didn't know about clean code or care about it. I just wanted to make some stuff. And my expertise was too narrow, limited to Telegram bots and some automation tools.

There was a pivotal point in my development career when I worked on a medium-scale Telegram bot that now hosts thousands of monthly users. This project was not a simple thing. Some problems I solved hadn't been solved by any of its competitors, and they still haven't been. I'm still proud of this project.

But the codebase was absolute garbage. Don't look at its current state. Even though it's still garbage, it was even worse. If you're a Python programmer, you might not believe it but I didn't even know what `__init__.py` really does so I never used it. I was not a pragmatic programmer. I was programming by coincidence.

But I still made a really great bot in a fairly short time (two months at most) **by hand**. The reason for this contradiction is that copy-pasting code is not as bad as they make it sound.

It's literally impossible to copy-paste your way through. It's not like the whole project is out there and you just have to find all the pieces of code and stick them together. If it were, everyone could make programs back then, the same way everyone can make programs with AI right now. You have to know so much to even be able to copy-paste.

It's the same way artists use references. Just attaching them together doesn't work. You have to know how to do it.

How everything caught up

In the beginning of 2025, I changed my life. I started working really hard. I decided to become a game dev and put so much energy into it. The wisdom I had gained from that Telegram bot opened my eyes to the fact that there's more to programming, and I took this pragmatic way of looking at things to expand my expertise and become the person I am today.

Even though I was working really hard, I was improving and growing even faster than I expected. I was already making a **very** complex card game within the first few months. It was really surprising to me how I was able to do something I deemed impossible a few months ago.

I realized that all those years of programming and expanding my knowledge had caught up. In the past, I had tried touching each programming language a little bit. I had tried C a little bit, Java a little bit, I knew some basic HTML and CSS, I read the Mozilla JS course but didn't practice it at all, I had touched React Native a little bit, Django to some extent, and more. I did the bare minimum with each one and didn't gain much from any individual one. But trying out different things and getting to know different perspectives helped me become an ocean only 1 cm deep: I knew a little about many things but nothing in depth.

I wasn't a pragmatic programmer at first, but if I hadn't been the garbage programmer I was, I couldn't have become the good programmer I am today. I was still a programmer, and there is merit in that. Copy-pasting code is by no means comparable to vibe coding. This comparison is just your coping mechanism.

"AI has a low context window and bad attention"

One famous judgment that people make when trying to sound rational is to mention that AI has a low context window that hasn't been evolving much. A context window is the amount of data that an AI can hold at a time. So the AI agent can't hold your whole codebase in its memory and remember everything. It's very limited. Attention isn't

perfect either, so some parts of the context are lost. To learn more you can watch this video; it's also an example of this false judgment that I'm talking about.

We have to first understand: Is it really different from what humans do? Which programmer do you know who has the whole codebase in their working memory while developing? And I'd argue AI is even better than a human if we compare them directly instead of judging each on its own. AI has a lot of tools for retrieving data from various parts of the project, many of which are pretty fast and accurate compared with humans. Humans get distracted easily (especially by real-life stimuli); AI doesn't.

But there's another way that we can look at this problem, and it would make complete sense: humans are trained on everything they've learned, especially the project at hand; LLMs almost never are trained this way. Every time you encounter a problem as a programmer, every time you invest time in it, touch it, deal with it, and solve it yourself, you feel a sense of success and achievement; you feel a release of dopamine. You're constantly training your brain on things while also memorizing them (the same concept was discussed in "Why do you care that much?", too). But this is not happening with AI models. Every time you fire up the agent to change your code, it reads the project from scratch. This is essentially why documentation has become very important: it's the only way we can preserve that understanding. It's quite literally similar to hiring a new developer and giving them the entire project every morning.

It's not quite fair to compare an LLM's context to human knowledge either. Humans do have context windows too, and they're called memory. Not only is this memory far better than AI's in terms of context management (for example, it can delete unimportant things pretty accurately, and you still remember a lot of things from decades ago—you have a much larger context window too, given current AI capacity), but human understanding is also comparable to AI's built-in parameters and learned patterns. I have already talked about this quite a lot.

Devaluing seniority and the field

This part belongs to On the effects of AI on the industry, but I'm separating it because it deserves its own section.

One of the most common things that people from different fields have been dealing with is how much other people devalue your work simply because they don't understand it. It's true for almost all jobs out there. You're a mechanic, but people think that you don't deserve good money because you fixed their car in a few minutes instead of hours. You're a dentist, and people think you don't deserve money for simply examining their teeth. You're a graphic designer and people think all you do is make some shapes with photoshop and "they can do it too". Even if you seemingly didn't do anything *that* special, your time is still valuable, but other people don't see that value and have this weird emotional judgment, thinking that you have to be paid only for the "amount of work" you have done and the immediate advantage it gave.

The reason why this happens is, in my opinion, something for sociologists and psychologists to find out. But we can safely say that it's definitely not a correct mindset. Not only do I think you can't judge the price of a product or service using "morality" (there's no objective way of assessing it; it's just supply and demand), but you also have to consider the fact that the person doing the job for you has not only put a lot of time into learning it, but could also perhaps do something better and more financially rewarding. Sometimes you pay someone simply for the time they spend, not for what they offer. It's a trade, in the end. If they think they won't get as much as (or more than) they lose from this trade, they have every right to price themselves higher.

This is exactly what has been happening to the tech industry, and it has worsened significantly with the introduction of LLMs. People who didn't see the merit in this profession are now more confidently expressing their false expectations and ruining the experience for technicians. One of the most common ways non-programmers judge programmers is by seeing whether the code "works." But making the code "just work" is something juniors can do too. So what does seniority mean? This is where it becomes obvious that those non-programmers don't even value being a senior developer. If your code works, they expect you to ship it, even if it's a prototype and you're warning them left and right. You're a senior because you know that just because it works right now doesn't mean it will continue to work. Remember, seniority means solving problems before they even have the chance to appear. It's an investment, and that investment is being lost in this sea of hype.

What should you learn for vibe coding?

There's a lot of confusion that everyone is experiencing right now. If you put the extremists aside, we are being told left and right, from the people who have "transitioned" from conventional programming to the new era of programming, that you still need to learn. Even worse, some tell you that it's even more important than ever to learn and "you need to become a senior".

And it's so damn confusing. What exactly should we learn? If coding is given to the almighty machine, then what should we learn? System design?

Even the best programming books out there that are supposed to teach you system design teach you almost nothing. They just teach you some mindsets and expect you to learn by "experience". But what could they really do? It's like reading a swimming book and expecting it to make you a swimmer all of a sudden.

The only answer left is to "learn how to work with AI," which makes even less sense.

1. The cost

You're aware of how expensive AI is right now, right? And it's not like you can learn by working with a weak model. Every one of the vibe coders mocks you when you're not using the top frontier models that burn through your usage limit in one shot, telling you "you're missing out". Not to mention that it takes much more effort—and therefore many more tokens—to learn than to produce.

I have to admit, it might get cheaper over time. But if that's your excuse, don't claim *coding is solved* until then. And you have to understand that this part of the equation is heavily related to hardware and hardware science is not improving as fast as software science, so your prediction could be severely wrong if you're judging it the same way.

2. No way to verify

A programmer friend of mine who had not tried AI until two months ago finally decided to get his hands dirty and start learning its concepts. He had followed the media coverage of AI, but he never touched it himself. He had heard there's this thing called a skill and it's really good and important and helps you make very good stuff. He went

and studied it, then came back to me and said, "is that all a skill is? You just talk to the computer?"

How do we learn if we have no way of verifying whether what we've learned is true? As I previously explained, AI is indeterministic. There's no logical reason why it works in a certain way, not the type of logic that we—as humans—need in order to understand. I tell the LLM something, and it responds with something. If I can't understand what connects the input to the output, my whole logical processor is impaired. And it's really damn hard to learn when you can't find objective connections.

We learn things by learning the patterns in them. Learning patterns is when you become independent from tutorials.

But in AI development, the pattern is not explicit. It's not even agreed upon. Everyone has their own opinion about it. So you're not given directions or scientific ways of learning; you're given a tool and expected to learn it by running it a million times (at the same time, not wasting thousands of dollars). And in the end, you realize the same logic doesn't apply to different models and you have to start experimenting from scratch.

I've experienced this multiple times while trying to achieve success with AI: I look at programming, understand what it's made of, extract explicit principles from it, and then try to replicate them with AI, but it keeps failing. It fails and I'm not sure why exactly, the same way you don't know why a bug happened, but this time you don't know if it's a skill issue or a legitimate limitation either. The only way to understand this is to run the model enough times that it becomes practically impossible to blame the failure on AI. This introduces a lot of friction and costs too much; you also can't do it for every bug. You see, in scientific professions, when you want to solve an issue, you have to isolate it as much as you can because you don't want to confuse yourself with fallacies. You might not understand the source of an issue and could fix something only remotely related, leaving the issue unresolved while thinking you've done something. But with AI, you don't have this luxury to question the tool. The boundaries are unclear. So in the end, you're left thinking, "Am I the problem?" and you have no objective answer for it. No one gives you practical answers; everyone's just like "if you can't replace your whole career with it, it's a skill issue. Source? Trust me bro"

When you can't objectively verify your understanding, you don't feel the reward, **you**

don't feel like you've actually learned something. Quoting again, *if you don't know why something works, you wouldn't know why it stopped working*.

AI maximalism means no one is safe

Software engineering used to be considered a highly professional field that requires a lot of hands-on problem-solving. If this whole field is "solved," it's really damn hard to claim other jobs are safe. The reason why we don't see as many advancements in other fields is that AI has not improved enough in image recognition, though it's improving.

This brings us back to the end is near argument. I'd rather ignore the conspiracy theories telling me the end of the world is near. Because the trade-off is obvious. I don't want to waste my life thinking the end is near, only for it to turn out to be far, far away, while it's not as though believing them would help me at all.

Any argument that implies this falls under this section. "I can produce a high-quality application with one junior and Claude," "AI has increased our team's output a hundredfold," "We replaced 200 senior devs with three devs and unlimited AI," and so on.

No need for AI experts

We have an expertise called SEO, search engine optimization. Back when the internet was in its early stages, SEO experts could do a lot of things. Keyword spamming was one of the things they did. It helped websites rank higher in Google search results. But at some point, Google updated its algorithm and penalized everyone who did that. This is because Google wants a fair game. It wants to show each user what they really need and enjoy. Google aims to remove the need for people like SEO experts. Justice doesn't need hacks.

Even though SEO experts still exist, they are much less in demand than before. The same would happen to AI experts. The LLM aims to remove the need for people like them. So don't think you're safe if you learn how to work with an LLM, you're not safe either.

It's scary as an experienced programmer

As I explained before, being a programmer is about this intuition and ocean of knowledge that you gain, keep sharp, and continue to grow over time. And it has been made like this because we were deep in the trenches, dealing with problems, letting our brains process what dealing with an issue looks like, learning by interacting, learning by being exposed to new and irrelevant info, and much more.

It's not like AI has just changed the programming language that we used. It has overhauled our process beyond recognition. Every programmer knows that adapting to vibe coding requires a huge mindset shift.

Senior developers, you have sharpened this intuition of yours, this spidey sense, so that when you're doing something bad, you feel it in yourself and stop. That's why all your senses are tingling when it comes to offloading work to AI. It's not just being addicted to the previous form of software development. This mindset shift isn't just about how you work; it contradicts your whole worldview. I don't know about you, but I don't know how to trust my intuition anymore if it has become this unreliable.

How to use AI properly

Using AI properly means understanding it as a tool, resisting the hype around it, and planning carefully to avoid long-term harm. It's not easy, I have to admit. The speed of things, the constant exposure to FOMO-inducing statements, the hard-to-grasp nature of this field, which suddenly expects programmers to become philosophers, and AI psychosis all make it harder for you to think rationally.

It's been really hard for me, too. As I said in the preface, I initially wrote this book for myself, to gather my thoughts and understand where I'm mentally at, to finally accept my situation.

But it wasn't always like this. I remember that a year ago, I bought Claude to help with my game development. I couldn't be happier. It felt much more like a tool than a new employee I couldn't trust. I was actually learning stuff and growing to become a better programmer, a better game dev. But right now, I'm using these LLMs to do all the coding, and I feel like I'm stuck.

The difference is in how I used AI. I wasn't asking it to just do everything. I made a lot of the things myself. I used it from time to time to generate some stuff, but then I read all the code it generated and tried to understand as much as I could. I remember using it only once for problem-solving without verification. And it was something like a conversion tool. It was just a simple script, though a crucial one. I couldn't understand it because it required a lot of knowledge regarding the programming language (C#) and I tested it multiple times and found that it worked. I postponed it until I could write a couple of tests for it and call it a day.

What I did and I suggest you to do is summarized in two categories:

The proper ways

There are two ways you can harness the power of AI without its downsides:

1. Non-problem-solving issues

The easiest is this. You use it for things that require no problem-solving and are just mechanical. In programming, most of the `refactoring` could be done by AI. But keep in mind that not all refactoring is mechanical. The larger it gets in scale, the less it's mechanical. Moving your codebase from Python to Go is definitely not as mechanical as mass renaming files and variables.

2. As an untrusted employee

Think of it this way: someone emails you and tells you they're looking for a job. You look at their portfolio and realize they're not very good. But their rate is so low that it's worth it. You hire them, but you're extra cautious with their output.

You may ask AI to write code and implement things, but you have to read every line and understand what it was thinking and why it implemented something. If you don't know, ask it. Never assume "there must be a reason for that that I don't know." This could easily lead to the triple debt.

The slightly improper ways

Of course, I understand. Not all code is supposed to be perfectly written, free of

structural and maintenance issues, or understood to the core. You would want to avoid scope creep. So it's okay to take risks, but your risks must be educated. It's okay to occasionally ask AI to generate stuff and trust it with some scripts and tools here and there. But always remember what you have trusted and what you haven't. If you can put unit tests around them, do so. This is the best way to take educated risks. If you can't, either remember them or write them somewhere. I suggest a single file inspired by ADR, but for all decisions. Later, if you have time left, you should review these.

Most of the time, you can't use it properly

The biggest struggle with AI for the developers themselves (those who don't want to vibe code everything) is how to keep learning in this era. And one of the issues stems from something that's unpredictable by nature, so there's literally no way around it: bugs. As I explained in You don't search anymore, the time you used to spend on bugs is a contributing factor to you learning and growing. And I'd argue it is one of the biggest factors too, because programming is, most of the time, about finding issues and bugs.

But the thing is, programming is against you by nature. Bugs are not something you deliberately make. They just happen and most of the time you don't know why. This means that you also don't know if the bug is something important or worth learning or not. Sometimes they're just silly things and sometimes they're not. If you use AI to fix bugs, you're gambling with your knowledge. And it's not something that you can fix by reading books or practicing by hand. When bugs happen, it's your only chance to learn how to solve them. Once they're solved, you're left with a finished puzzle that teaches you nothing.

When you continue this chain of thoughts, you realize most other use cases are the same too. The act of digging through the codebase to learn is an important part of your learning journey. In Code review isn't the answer, I explained how I was able to use AI to add a feature to Godot. But a hidden caveat that I didn't tell you is that I wasn't able to continue this journey. I tried multiple times after that to fix some open bugs or do anything at all, but I failed miserably. I realized that—while using AI—I had offloaded the process of understanding the codebase and the programming language to AI. I was just producing. I managed to add that feature, but that was just it: adding that feature. That

knowledge was mostly throwaway knowledge. This is essentially the same as that doom-scrolling example. Juniors will remain juniors with AI.

One important aspect that makes AI quite incompatible with programmers is explained in "AI has a low context window and bad attention". I'm not saying that you can't or shouldn't use AI in your work as a programmer. I'm just explaining why there's a lot of friction here that people might have missed. Everyone thinks AI is our new tool, while this tool is completely different in nature from us, and somehow we have to manage converting our human workflow into an AI workflow while maintaining the same productivity. It simply doesn't work that way. Your high hopes don't make reality any different, even if the world adapts to you.

Chapter 4

The future of AI

Initially, I didn't want to write this chapter. With how fast and unpredictable everything is moving, it's really easy for any of your predictions to be way off, at least in their timing. But I'm keeping it for two reasons: first, I want to document what I'm thinking right now and reflect on it in the future; second, I want to talk about expectations about the future that can affect our current lives without necessarily being true.

The AI university

Even though I have frequently opposed replacing specialists with AI in this book, I can imagine that I might be wrong and consider what the future could look like with the current pace of AI.

One of the things we discussed in "What should you learn for vibe coding?" is that it's really hard to know what AI is doing, and we don't have any way to verify it. This is one of the things that could theoretically be solved if we introduced highly professional university majors for it. Take a look at fields such as sociology or even graphic design. Even though these fields definitely use some scientific methods (just as vibe coding does), they are mostly based on experience and outcomes. In The outcome bias, I explained why it's logically wrong to judge based on outcomes. But sometimes, you simply can't judge at all using pure logic. In a society, there are too many factors to make everything make sense through mathematical logic. But you need sociology anyhow, so it's justified to resort to personal experiences but in a professional way.

We mitigated this problem by introducing a new method: we find mathematically sound ways of evaluating outcomes (random sampling, for example), gather a ton of them, and then use philosophy, logic, statistics, psychology, and more to form opinions. Since we don't have an objective way to verify our results and can only be so certain, we never treat sociological statements as objective facts. We should never trust a sociologist's statements in the same way we trust the statement that "gravity exists." This by no means undervalues sociology. History will tell us whether we should trust a sociologist or not. In the same way, history might prove my prediction about AI correct,

but that doesn't mean I was objectively correct beforehand. I made an educated guess, and it turned out to be true. Sociology, graphic design, and other similar fields are just collections of educated guesses.

This is what could possibly happen to AI engineering. Since we don't have objective and deterministic verification methods, what we should do is invest in this field and make it a university degree. Instead of expecting people to do all these tests on the models, we do them and use mathematics to make sure these tests aren't biased and are being done correctly. Then we make rules from them. Then we shape these rules over the course of humanity to make the smoothest development experience.

This could also be why we—as software engineers—are struggling to adapt to AI. We are expected to suddenly gain years of knowledge while no one knows anything. It's like going back thousands of years and expecting people at that time to explain the common sociological and psychological concepts. The reason why sociology and similar fields work this perfectly today is that they have been perfected over the course of thousands of years. Programming in general is less than a hundred years old.

The false expectations

The thing is, even if AI is inherently unreliable, its extreme resemblance to human beings has given—and will continue to give—many people, managers, and CEOs false hopes and expectations.

1. Lack of seniority

Managers stop hiring juniors and stop investing in them. Seniors retire, and at some point we'll have a shortage of senior developers. As simple as that.

2. Technical debt

Though other forms of debt also play a part, technical debt is more visible. A lack of seniority, hallucinations, and the use of weaker models because of AI costs could accumulate debt and cause a lot of problems in the future.

3. Ruining the industry for a couple of years

Ultimately, we can easily ruin the industry for at least a couple of years. There's no

stopping that, because the issues mentioned in this book aren't immediately visible. They show up only after they accumulate. Hopefully we can stop that sooner. That's why I highly suggest that you speak up about these topics.

Software becomes dead

If despite my understanding, the only barrier to SWE becomes the idea and everything else is done by the machine, then the market becomes so saturated that software becomes practically dead. Why would you buy software when you can build it yourself? Intellectual property would mean nothing, because you can take the genius idea that someone made an app for, and make it yourself from scratch, fine-tuned to your own needs as well. If you don't show it to anyone, you haven't done anything immoral or illegal.

Elon Musk posted a tweet a few weeks ago saying that programming languages will be dead in 10 years and AI will write machine code directly. While it caught everyone off guard and he was made fun of, this is what AI maximalism looks like. When you say you offload all coding to AI, you're expressing a similar expectation. I'm not saying that if offloading code to AI turns out to be the correct way, then AI will necessarily write direct machine code in 10 years. But the same prediction could be made about claims such as "in the future, operating systems will become obsolete and everyone will build their own AI-native OS." I actually posted a bunch of these predictions generated by ChatGPT in a tweet. Really funny to read.

Civilization collapse

There's a very thoughtful video by Jonathan Blow called Preventing the Collapse of Civilization. In this video, he talks about how technology decays on its own and requires constant work to actually improve. Thousands of technologies have been lost throughout history for this reason. A lot of advancements in science and technology have been reduced to nothing. It took us many years to understand how the Pyramids of Giza were built, and we're still not sure about many aspects of them.

Software is already decaying. This video is from 2019, way before ChatGPT's public release. One thing he said that I can't get out of my head is that people say, "Yeah, we could make this software better and less buggy, but the market doesn't pay for it," but

how do we know whether they actually could? When it has been decades since robust software was made, what makes us think that we still have the knowledge? Knowledge has to be constantly transmitted and honed; otherwise, it rots and decays. I, as a programmer, can say that I can write better code, but the fact is, the more you stop doing things the correct way, the more you stop learning and evolving, and the more your knowledge expires. I may be technically able to write much better and more robust code, but if that takes me 100 times longer than it should, then I probably don't actually know how to do it anymore.

Let's say I learn something and put years of my life into understanding it. Then I explain and teach it in simple terms to younger generations and other people. They can start where I left off. They haven't filled their minds with irrelevant data. They can start much further ahead. The reason you can speak a language so fluently is that it has been perfected and transmitted for many years. You don't have to waste so much time creating new concepts and words just to communicate with the people around you.

In the same way, if I learn and practice how to create better software, the norm changes over time. The knowledge that I have no idea about right now will later become muscle memory. Even if the market doesn't pay for it, that doesn't change anything, because my productivity increases.

That's the paradox of using AI. We think we have increased productivity, but if we have removed all parts of this experience and are just producing, we're eventually going to lose our knowledge. Even if you don't lose it, you'd lose your grip on it; your flaws would increase, you'd stop being sharp, and, when accumulated, this would result in decreased productivity. Terence Tao, perhaps the biggest mathematician alive, has a video where he talks about this paradoxical nature of using AI for work. I highly recommend watching it.

Moore's law

There's something called Moore's law. Gordon Moore said that the number of transistors on a computer microchip doubles roughly every two years. This means that the size of chips and CPUs constantly shrinks. This law continued to hold true until around 2013, when it was estimated to become obsolete. But a brilliant group of scientists managed to pull off the impossible and invent a new technology that saved

Moore's law. If it weren't for the constant effort of scientists throughout history and the investment of hardware companies (who needed their products), we wouldn't have had this technology. Imagine if no one had invested in it and the stagnation had continued until the decades-old papers that made this technology possible were forgotten or, even worse, somehow deleted (don't forget the Library of Alexandria). This knowledge could easily have been lost to history, as many other technologies have been.

The apocalypse

Well, we already talked about it. There's no theoretical barrier to an AI apocalypse in my opinion. Who knows? It might happen in our lifetime. It's the ultimate gen z experience at this point.

The good prediction

If the bad predictions don't come true, the golden age of indie development (especially in fields with high barriers to entry, such as game development) will become a platinum age. Good things that simply couldn't be made back then will be made now. And the good stuff that existed before would become even better. AI democratizes the industry in the same way it already has.

Chapter 5

AI for creativity

This chapter is likely to be very controversial. Most of the time, when people think about AI's downsides, they're thinking only about this field. So I'm making an attempt here to explain as much as I can. Feel free to skip if it's not your concern.

I have to mention beforehand that, even though I'm primarily a programmer, I've been making art for a long time, too. I'm even studying graphic design right now, and my corporate jobs have been in this field. I've made digital art for two years (and will continue to do so), as well as other forms of digital and physical art since I was young. That's why I feel confident enough to talk about this topic.

But at the same time, due to not being a senior artist (only a junior graphic designer, I would say), I wouldn't go deep into the technical details. As I mentioned at the beginning of Chapter 3, you can relate what I have said (or will say) to your own expertise and see how it applies in your industry.

Ownership

One of the most common misunderstandings I've seen is related to this topic.

Even though I understand that most people don't agree with this perspective, I have to say it because I think it's the truth. The fact is, morality is just a set of contracts. You might conclude from this that morality doesn't matter that much and that we should take it easy. But I absolutely disagree with that. This definition of morality doesn't change how we should feel about it. The feeling we have toward morality is based on its usefulness, so let me explain.

Back then, before the rise of societies or communities or any form of intimacy, everyone was on their own. You had to survive not only among wild animals but among other humans as well. This was not efficient. We weren't benefiting from this system. So we embraced companionship. I bring food for us and protect us against the dangers; you take care of the house and the child. We can benefit more if we collaborate than if

we go on our own. This is a very logical way of creating the concept of "trust." We trust each other because we want to benefit more. This grew to become what we know as "society." We benefit more as a society than we do as individuals pursuing our own interests. The benefit of all is the benefit of each individual. The reason we don't die from a simple cold is that we've collaborated: the doctors make medicine for society, society gives them money, and they can buy a fancy laptop and so on.

Morality is the essence of societies. We built a set of rules and contracts because we want to support and strengthen our societies. Anything immoral directly or indirectly **contradicts the concept of society** and therefore harms society. If you kill someone and don't get punished for it, then other people can't feel safe, so they attack first and suddenly there's anarchy going on. If you lie to someone, you might not seem to harm society, but your mindset is what matters. Over time, these lies and immoral acts can create a disaster as big as Chernobyl. This is also why I insisted on the "contradicting the concept of society" part, which I explain more in the next paragraph:

My view is that you only have to *know* the meaning of morality, but you shouldn't always think about it directly to reason about things. You don't need to find a direct line between a subject and your own society's stability to understand whether it's moral or not. The reason why most people have a conscience is that the biological need for morality and for keeping society alive has been built into them, the same way you have a natural tendency to form a family and have children. You can use this feeling and core understanding rather than relying too heavily on the hard-to-grasp idea of connecting every action to direct and visible harm to society.

In the end, the feelings you have about morality are all valid. I just proposed a reason for them.

This is a huge topic that has been discussed for centuries, and I don't intend to dive deeper into it. So let's wrap it up and connect it to copyright.

Just like morality, "ownership" is something that doesn't mean anything by default. It's a complete contract. The reason you say "I own this spoon" isn't that you made everything about it from scratch. You didn't create the metal from nothing. It existed in nature before you. You didn't even form the spoon yourself; you bought it from the company. But you still call yourself the true owner of it, simply because you paid for it

(you sometimes pay to rent spoons as well, for example, when you're in a restaurant). And for the same reason that money is just a contract, this ownership is a contract too. If I steal the spoon from you, I don't magically become the owner of it because it's in my hands. I become a thief, and you're still the owner. We made this rule because, without it, society would collapse. Remember, when you were in a jungle, you had the so-called "ownership" over your stuff too. But everyone could take it and you couldn't do anything about it. We invented society and property laws to prevent that. We redefined what ownership means, and it's just a contract.

Artistic style and intellectual property

This applies to intellectual properties as well. Artistic rights are no different from other types of ownership.

Now I have to tell you a simple fact: "style" doesn't hold copyright. You can't sue someone for having the same artistic style as you. You can't sue someone for using the same color palette as you. And it's not something new. It has been like this forever and in all artistic categories.

Say all you want, but this fact remains for a reason. And the reason is that we have no reliable way of recognizing what a style even is, why it should "belong to you," or whether you have borrowed it from someone else. Even if you could somehow make an art style a form of intellectual property, almost all current artists would have been in legal danger. Imagine you sell a company full rights to your artwork, and now that style is theirs, so you can't use the style you have honed over the years.

Even if you didn't do it intentionally, your style still resembles that of many other artists. You simply can't claim that you have a "unique" art style. There are no unique art styles, at least not anymore. I remember that before AI, every good artist was (and some of them still are) saying that a good artist is a good thief: you steal from various people until no one can detect who you stole from. This is the same.

The only thing you can claim is your "artwork." You can copy someone's painting without copying their art style and still get sued. Now you may ask: how is it possible to copy the artwork without the art style? The answer is by changing the medium. There's

a famous case called Luxembourg Copyright Case Against Jeff Dieschburg. A painter painted someone's photography. It wasn't even a one-to-one copy, but it was realistic and close enough to be recognized as a copy, which led to a lawsuit. You can even read people's reactions on that webpage to see how rude a person can be. Don't base your judgments on "the amount of work." Even if you spend 50 years copying someone's artwork, you're still committing copyright infringement (unless you have permission).

The contradiction

Let's say you made an artwork and sold it to someone. Can that person scan your artwork and put it on the internet as their own? Of course not; they're not allowed to sell your artwork while saying that they're its creator. And they don't even need to claim it; by default, they're not allowed to upload the scan to the internet at all. It's the same for other things as well. You can't scan the book you bought and post it on the internet, whether paid or free.

What do you think is the reason? You might think that, well, it damages the artist's brand and market, making the competition unfair. But why is it that you can scan and sell, let's say, a spoon? Even with a book, you can resell it on the open market. Doesn't this damage the brand, company, and market?

When you think about it long enough, a lot of these contradictions arise. You'd realize that copyright laws differ a lot, and you don't even feel like they're unjustified. They feel natural to you, although they're not very consistent.

The reason is what I explained in this chapter: ownership is a made-up concept required to sustain society. It's not something that is given to you inherently; it's something you gain by being part of society (criminal activities make you ineligible to be part of society, which is why you lose rights). It's not like all ownership rules are perfect; they change a lot.

So what makes you think AI art isn't real art? Just because most parts of it have been automated? What about generative art (not to confused with generative AI art)? What about computer art? This field has existed since the 1960s. In the same way that the Notepad application isn't an artwork but Golden Calf is, and a wooden door is not an

artwork but Wooden Mirror is, a cheaply generated AI image is not an artwork, but a carefully crafted one can be.

AI and theft

If we accept that art style itself isn't protected by copyright, we can easily see how LLMs are not "based on theft." This would also make more sense when you realize you can't even prove the existence of the soul (as discussed in Chapter 2).

Now, I understand that not all people agree with me on either the soul part or, surprisingly, the style part. But what we should agree on is that "theft" is a very serious accusation. If you're going to call someone a thief, you've got to be a lot more certain than this. This mindset of accusing people of horrible things is dangerous and immoral. It's the same as a false accusation of rape, just a little less destructive. Some may say they're just calling LLMs thieves, but by doing so, you're calling all these people supporters of theft, too: the developers of the LLMs, the companies that own the LLMs and their employees, the people who use LLM APIs to build SaaS, the people who use LLMs to generate text, code, images, or anything for any reason, the people who talk enthusiastically about LLMs and invite others to use it, and even the people who use local LLMs. If you feel like I'm exaggerating, maybe you're undervaluing one of the most common criminal acts called "thievery." You don't want to just accuse people of criminal activities because you believe in ghosts, do you?

There is no *morally* created LLM

Large Language Models are essentially the result of training a machine on a lot of input. For example, you train the machine on 10 million images of cars, then you get an LLM that can generate accurate images of cars. The LLM doesn't hold those images; it just saves patterns, and those patterns can be used in any way possible to generate anything from scratch.

You can train an LLM on your own artworks to create art in your style, but for that LLM to be useful in the first place, it has to be trained on tons of other images. It needs all of those patterns to know what your artworks even mean. Unless you have painted millions of artworks, they wouldn't be of any use. The foundation is training the model

on pre-existing data. If that is thievery to you, then there's no morally created LLM. It's like curing cancer through slavery. Even if you cured cancer through slavery, that wouldn't make you any less immoral. Remember, you don't judge based on the output.

It can't be thievery

As I explained, LLMs are created by training on other people's data. If you're one of those people who thinks this is proof that it's theft, I have a question for you: How do you think the human brain works? Why do you think our brain isn't essentially doing the same thing? This is essentially what you have to do to improve. This is what "studying" is: you copy things, from masterpieces to objects and other things. If it's a master study, you try copying the artwork while being mindful of why that artist did that specific thing. One by one, you burn patterns into your head. When you do other types of studying, you do that again. The whole premise of studying is that you learn the patterns. You don't save the image in your head and copy it every time. You learn how that image was created. You may even be able to copy that artwork, but that doesn't mean your knowledge is stolen.

Theft is a very defined concept. If I sell you a knife and you use that knife to kill someone, I'm not the one responsible. Even if you use it in 101 different ways to hurt people, that still doesn't make me your partner in crime, because the knife has a very defined purpose. Killing is just one use case, and it doesn't have anything to do with the knife itself, as one can kill with bare hands. In the same way, one can use AI to copy someone else's artwork and change it slightly, but that doesn't mean the fault is with the tool. The person doing it is at fault. There are already a ton of videos on YouTube about how to remove a watermark from an image. People have been using Photoshop to fake important documents and stamps since it was released. Why don't you feel the same way toward it, then? Just because it's harder to do so? I'd argue it's even easier and safer to use Photoshop to fake documents. If LLMs are thieves by nature, then Photoshop is definitely a thief by nature as well.

AI art

If we get past moral issues, there are still a lot of controversies around what real art is and why some people don't consider AI art to be art.

Art has two parts:

1. **The creative part**: the idea behind the artwork, its meaning, the feeling you aim to invoke, and more.
2. **The technical part**: the production, the use of artistic tools, and the scientific knowledge regarding composition, color theory, how to make a brushstroke, how to use your hands, and so on.

You may not like the term "technical," especially if you haven't had a job as an artist, but it is true nevertheless. It's technical because it's just some scientific knowledge, some technique that you have to learn and apply. Yes, you can combine it with creative decisions, but in the end, the two parts are separate. What you want to make and how you make it are separate parts of the same job (that can affect each other too). You might have a great idea but not know how to implement it. At the same time, you might know how to do the technical part but lack the creativity to know what to create (the hyperrealism style is essentially that).

Even the term "technical" itself comes from the ancient Greek word "techne," which meant art. Art back then was just a craft. Painters and sculptors were the same as carpenters and blacksmiths. They followed rules to create stuff. Even though they now appear to have a lot of creativity, back then it was even taboo to be creative ("A History of Six Ideas: An Essay in Aesthetics" by Władysław Tatarkiewicz, 1980, Chapter Three: "Art: History of the relation of art to poetry"). That said, we can see the connection clearly: the technical part is where you're doing things as objectively as possible. You're not being creative per se, but you're being an artist anyway.

This is why AI is just a tool. AI is not sentient. It can't invoke things on its own. An LLM is a pattern-based machine. It follows the patterns of your prompts. Even if you tell it "make a painting," it still needs that instruction in order to create something. If you say something else, the output changes. A human's output doesn't necessarily change in the same way.

So yes, of course you can just ask AI to create stuff and it's not called art. In the same way, a person can grab a pencil and draw lines on paper, and it's not necessarily called art. Art is intentional and deliberate, and it requires creativity, all of which can be supplied by the artist. Just as a hyperrealistic artist is still an artist, a person who uses

AI can still be an artist. By common logic, the latter is actually more of an artist because the work is all about creativity and not about what "anyone can do."

Also, you might have noticed I said "the person who uses AI" instead of "AI artist." This is because AI is not a style or a medium. A key factor of a medium is that the artworks that belong to it must have something in common. When you can make absolutely any style with it, it's not reasonable to use it as a style category. AI art can still be used as a category, but only as a tool category, the same way you might call someone a Photoshop artist.

Authority over artwork

One of the biggest reasons artists (especially novice artists) are not okay with AI art tools is that they don't feel in control of the output. They feel that "it generates random stuff, it doesn't know what I want, and it's just a slop machine that creates soulless output." And then they use this argument to prove that it's the same for other people and therefore it's not real art.

You might have thought to yourself that I'm not making sense. When it comes to programming, a job that's obviously way more mechanical than art, I say you don't have authority over the output of AI. But when it comes to art which is a very intimate thing, I'm saying that AI is a tool and you can have authority over its output.

First of all, I have to say that most, if not all, of the downsides that I have mentioned could apply to other fields, including art *as a business*. I haven't been very professionally involved in the art industry. I was learning all the time and I was still far from actual production. So I can't claim anything about that.

Second, there's something inherently different about the nature of art that I believe makes a difference: art is random by nature. Every brushstroke is different from the next. There's a popular saying that goes, "Each time you paint something, it's a different artwork." This is essentially why traditional artworks are priced so highly. The original artwork could only be drawn once, making it rare.

This randomness is also something most artists are familiar with when making art. If you're an artist, ask yourself: how many times have you had an idea in your head,

started creating it, and watched it turn out differently? Most of the time, you didn't even have a vivid idea of what it was going to look like. You built the idea while technically making it.

That's why I don't have a problem with randomness in AI art, just as the artist David Hockney didn't (rest in peace). The point is, can *you* control that randomness to get close to your idea? Can you control your feelings so you don't call it shit because it turned out differently from what you initially thought? Some people claim they can. Who are we to say they're lying? I thought art was a personal thing.

I'd even say this random nature of art means AI is better suited to this field than to a sensitive scientific field. Scalability and maintainability are not as big of issues in art as they are in SWE, if we can define them in some projects at all.

We're not enemies

We're all figuring out how to incorporate AI into our fields, but let's be more open to different judgments and opinions. In some fields, such as game development, one of the biggest reasons the barrier to entry was so high was that programming itself was too time-consuming, let alone the artwork, which requires years of focused, productive learning. 3D game development was almost literally a no-no for solo developers due to the insanely high barrier to entry. AI has enabled people to do things we really couldn't do before. Right or wrong, let's figure it out before calling each other names.

Conclusion

When something has had fundamental effects on a field, you have to start questioning the fundamentals of that field. The overhaul in the industry (especially the tech industry) is not something to take lightly, as you might miss a lot of things because you're among the first people experimenting with this completely new workflow. AI is a very useful tool, but useful doesn't mean safe; a knife is useful and unsafe at the same time.

The concept of "seniority" has always been connected to experience. But now we gotta question the fundamentals: How are experiences connected to knowledge? What makes experiences experiences in the first place? What's the difference between an experience and a memory?

One of the things that seniority has taught us as software engineers is that "if it works, don't touch it." It might sound stupid, but it's just a funny way to explain a very important concept: changes are risky. Every time you change something in your system, you're introducing new ways to break things. The larger the surface area, the riskier the process becomes. But now everyone wants to do software development in a completely new way, using indeterministic tools that hallucinate and that we've only been using for a few years or even months. This is the harshest change ever, but some people are surprised that you worry about it, as if the seniority you gained had been a hallucination all this time.

AI is definitely capable. It has helped us do things beyond our wildest imaginations, and I am by no means against it. But using AI as a new tool is one thing, and using AI as a way of gambling with your future is another. This... thing that AI is doing is nothing like the tools I used to know. It's more like a constantly drunk employee who also forgets everything every few hours.

Software engineering is among the jobs that have changed the most. We, as programmers, are used to changes in our industry much more than most other professions. We are completely okay with replacing our "tool" with "newer tools." Yet the conversation around AI in programming is very loud and dramatic. Even people

with the most flexibility aren't okay with this sudden rush of changes, and quite literally everyone—including those who have wholeheartedly accepted AI in their jobs—is saying that they're having substantially less fun doing what they did one year ago. The constant speed of everything is absolutely overwhelming.

Maybe I'm wrong. Maybe this whole book is just the bargaining stage of my 5 stages of grief. But it's not precisely the best time to be a software engineer. Even if you figure AI out, as I feel like I've just done, you still haven't changed the reality around you or the job market's expectations of you. The contractor doesn't care what tool you use, but they ask you to deliver the product in such a short amount of time that it's practically impossible to do it without offloading everything to AI.

Your job was already full of stress, and now they want to force you to get rid of all its meaning and joy. If you want to keep your job, you have to conform to their new policies, which set you up for failure without you even knowing it. The policies force you to forget who you are, bury you under prompts and tickets, keep you from feeling productive, and make you miss the flow state you used to be in. The funny thing is that staying in this situation would definitely degrade your expertise, giving you even more FOMO. But if you mention this, they just advise you to "go learn," and they never tell you how or what to learn.

We just wanted to become good programmers—to get better at our jobs, become mid-level developers, senior developers, better seniors, principal engineers, and so on. But now everything has lost its meaning. You're constantly producing with no sense of progression. We don't know what to learn. Putting yourself in the position of learning the fundamentals seems so damn hard and pointless at this point, when you know AI can do in a few seconds what you're merely trying to learn.

Who knows what will happen? This is the wildest decade we're experiencing—starting with the COVID pandemic and moving straight into the age of artificial intelligence. And for what it's worth, this is the beginning of a new era: the era of burnout.