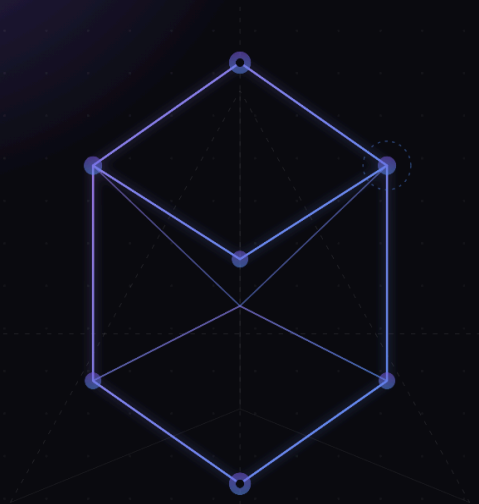
PRACTICAL GUIDE

AI · NO. 01



BUILD WITH AI, UNDERSTANDING WHAT YOU'RE DOING

Architecture for vibe coders

Understand what AI builds for you
— before it breaks.

WRITTEN BY

Diógenes Alberto Moreira

Architecture for Vibe Coders

Understand What AI Builds for You — Before It Breaks

Diógenes Alberto Moreira

Architecture for Vibe Coders

Understand What AI Builds for You — Before It Breaks

ISBN: 978-631-01-6621-6

Independently published

First edition, 2026

Contents

Acknowledgments

To Lucrecia, my wife, and to my sons Ismael and Luca: for putting up with all the time I spend shut away in the office, and for the coffee that appears on the desk when the night runs long. This book was written in hours that belonged to you.

To Fabián Gómez, who was one of the sparks behind this book — sometimes a single conversation pushes harder than a thousand plans.

To Emiliano Arango, my friend, teacher, and student all at once, with whom I've shared so many technological adventures that a few of them ended up here, in disguise, in these pages.

To Andrés Bilbao, to Dylan Rosemberg, and to the whole 30X team, who without knowing it gave me the raw material for this process just by letting me watch the day-to-day.

A line for Abel Moreira, my father, and Axel Moreira, my grandfather, who watch over me from above: this is yours too.

And to my mother, Amelia Bernalina Gómez, for everything that doesn't fit in a list of acknowledgments.

To my grandmothers, who sowed long before I understood what was being sown.

Introduction — Power Without the Map

I begin the task of writing this book with one conviction: vibe coding is here to stay.

Unlike other passing fads I've watched parade through this industry — and I've seen plenty — this one isn't a promise: it's a done deal. The power of generative AI threw open the doors of software development to people with no formal training. Designers publishing their own tools. Entrepreneurs building their MVP over a weekend. Accountants, doctors, lawyers automating what used to require hiring a whole team. People who two years ago didn't know what a terminal was now have apps running on the internet.

And yet — as much as it sounds like a contradiction — those same people lack the knowledge they need to complement what the LLMs give them.

This isn't a criticism. It's a description of the terrain. The model writes the code, and it writes it well. What it can't do is decide for you: it doesn't know whether your app will have ten users or ten thousand, it doesn't know how much you can spend on infrastructure, it doesn't know that the "throwaway" prototype is about to become the system your business depends on. The AI executes; the judgment is still yours. And judgment, in software, has a name: architecture.

That's why the story repeats with almost comic precision. The vibe coder moves fast — far faster than any traditional programmer ever dreamed of. The demo works. The first users arrive. Everything flows… until it hits the wall.

The wall has many faces. Sometimes it's a database that gets corrupted because nobody thought about migrations. Sometimes it's an exposed API key that wakes up one morning as a bill for thousands of dollars. Sometimes it's quieter: the project grows until not even the model understands the code it wrote, and every change breaks two new things. The symptom varies; the cause is always the same. The code didn't fail. The map was missing.

This book exists to close that gap.

I'm not here to teach you programming; for that you already have AI, and it's better at it than I am. I'm here to share with you what AI can't give you: an understanding of what you're building, what pieces it's made of, how they connect, and where they break. The vocabulary and the judgment to stop being a passenger in your own project and become the architect who tells the builder — tireless, obedient, brilliant, and blind — which building to put up.

The journey is your own app. First the map: what architecture is and the anatomy of any application. Then the pieces: frontend, backend, databases, users, the outside world. Then how it all connects: patterns, stacks, deployment. And finally, the things nobody tells you until they hurt: security, costs, technical debt, and how to talk to AI like an architect instead of a tourist.

Each chapter combines two registers. First the narrative: concepts explained with analogies, real stories, and zero unnecessary jargon. Then the practice: concrete exercises to apply what you've learned to your own project, with your own AI. Because this book isn't meant to be read — it's meant to be used.

A confession, too, before we start: in this book we're going to take poetic license. Technical content goes down easier with analogies than with definitions, and analogies are always imperfect: a database isn't exactly a pantry, a backend isn't exactly a kitchen, and more than one specialist will frown — rightly — at some of my simplifications. I accept that up front: we're going to get criticized for a few things, and that's fine. Every time there's a choice to make, this book chooses clarity over textbook precision. If by the end you want the full rigor, you'll be in an unbeatable position to go find it — and that's exactly the point.

An honest warning before we begin. Knowing architecture won't keep you from making mistakes; it will let you make them more cheaply, catch them earlier, and fix them without demolishing the building. That's the difference between the person who races ahead until they hit the wall and the one who reaches production without dying in the attempt.

That's the road we're going to walk together.

Chapter 1 — What Is Vibe Coding, and Why Does It Work (Until It Doesn't)?

Chapter structure

- 1.1 The new way to build software
- 1.2 Success stories and disaster stories
- 1.3 The exact moment "not knowing" gets expensive
- Practice: the inventory, the interview, and the map of walls

⁂

1.1 The new way to build software

The term was coined by Andrej Karpathy, one of OpenAI's founders, in a February 2025 tweet that turned into prophecy: "There's a new kind of coding I call 'vibe coding,' where you fully give in to the vibes, embrace exponentials, and forget that the code even exists." He wrote it half as a joke, like someone giving a name to a private vice. The tweet passed four and a half million views within days, and the industry — which had spent months doing exactly that without daring to admit it — took it completely seriously. Within weeks, "vibe coding" went from a programmer's inside joke to a category: it appeared in Forbes and New York Times headlines, in job postings, in investor decks. Dictionaries started tracking it. Rarely has a new word found the millions of people who were waiting for it so fast.

Because it described something that was already happening. Millions of people had started building software through a conversation. The mechanics are so simple it's hard to believe

they work: you describe what you want in your own language — "I want an app where my clients book appointments and get a reminder over WhatsApp" — the AI writes the code, you look at the result on screen and ask for adjustments. "Make the button bigger." "Send the reminder a day earlier." "Now add payments." You don't read the code. You don't understand it. In many cases, you don't even see it: modern tools hide it behind a preview, like the engine under a car's hood. The cycle is describe, generate, test, correct. And it works.

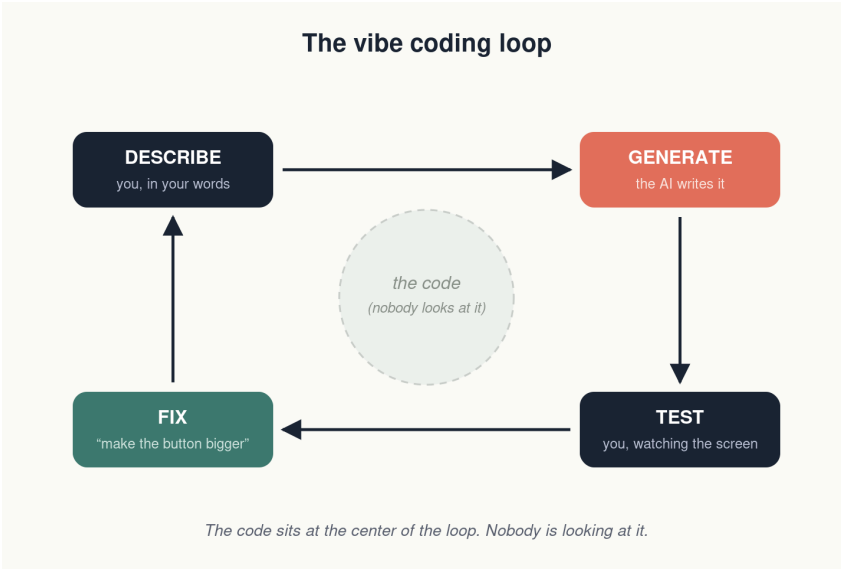


Figure 1.1 — The vibe coding cycle.

Around those mechanics, an entire industry bloomed in record time. Cursor, a code editor with AI baked in, became the fastest-growing software in its category in history. Lovable, Bolt, and vo went further: they got rid of the editor entirely, leaving only the conversation and the preview. Replit turned its platform into an agent you hand the whole app to. Each one chose how

much code to show you — from everything to nothing — but they all bet on the same thesis: that the next generation of software creators won't want to see it. The numbers proved them right faster than even they expected; several went from zero to billion-dollar valuations in under two years, carried by a wave of users who didn't come from programming: they came from having an idea.

It's worth pausing on how radical the shift is, because from inside the wave it's hard to measure the wave. For seventy years, the history of software was the history of a barrier to entry. Programming demanded years of study: languages with hostile syntax where a misplaced semicolon brought everything to a halt, data-structure theory, algorithms, and the accumulated folklore of an entire profession — which tool to use, what error means what, who to ask. The barrier filtered out almost everyone. On the inside stood the programmers, one of the best-paid professions on the planet precisely because the barrier was so high. On the outside, the rest of the world with its ideas: the dentist who imagined a better appointment system, the teacher who knew exactly what app her students needed, the shopkeeper doing by hand what a computer would do in seconds. Ideas orphaned of execution.

Every so often, the industry promised to tear down that barrier, and the list of broken promises is long and instructive. COBOL, in 1959, was designed so managers could write their own programs in something resembling English; it ended up being exclusive programmer territory, and the managers went on dictating memos. Visual Basic and the visual tools of the nineties promised programming by dragging buttons around with the

mouse; they were good for prototypes and died of complexity the moment a project grew. Dreamweaver and the website builders put pages within everyone's reach, as long as "everyone" wanted exactly the kind of page the tool knew how to make. And the no-code platforms of the last decade — Bubble, Webflow, Airtable — got further than anyone, but hit the same old ceiling: they worked until you needed something their creators hadn't anticipated. Then the barrier reappeared, intact, with a sign that read "from here on, hire a programmer."

Generative AI is different, and this is the conviction this book rests on: it's not just one more item on that list. The difference is structural. Every earlier tool handed you prefabricated pieces — templates, blocks, components — and your freedom ended where the catalog ended. The LLM has no catalog: it writes real code, the same code a programmer would write, in whatever direction your idea takes it. If you can describe it, it can try to build it. There's no template ceiling to hit. The limit is no longer what the tool anticipated, but something far subtler — and to that something we'll devote the rest of the book.

Someone who builds this way we call a vibe coder, and it's worth defining well, because it's not "someone who doesn't know how to program." It's someone who knows something else. They know their business, their industry, their problem. The designer who built the tool her colleagues use every day doesn't know what a database migration is, but she knows exactly what a design team needs, which workflow hurts them, which screen they'll open first each morning. That domain knowledge — which no LLM has about your particular problem — is the valuable half of the

equation. The AI supplies the syntax; the vibe coder supplies the meaning.

The spectrum, moreover, is wider than the caricature suggests. At one end is the entrepreneur who has never opened a terminal and builds everything by conversation. At the other, veteran programmers who know perfectly well how to read the code and choose not to, because reviewing every line would kill the speed that makes the method magic. In the middle, a growing population of hybrids: product people, analysts, designers, data scientists who know "a little" and stretch that little to infinity with the model's help. What they all share isn't a level of knowledge: it's a new relationship with code, where code stopped being something you write and became something you ask for.

And here's where the asymmetry that defines this new way of building shows up. The traditional programmer wrote slowly but understood everything he wrote; when something broke at three in the morning, he had a mental map of the system to go hunt down the fault. The vibe coder builds at a speed that two years ago was science fiction, but understands little of what's being built in his name; when something breaks at three in the morning, he has a conversation with an AI that sometimes doesn't understand it either. Speed without comprehension. It's a trade, though nobody presents it to you as one: nobody warns you, while the demo works and your friends applaud, that you're signing that contract. It's a lot like driving a car without knowing anything about mechanics: perfectly reasonable in the city, with a garage on every corner, and a decision that costs differently out in the middle of the desert. The trouble with software is that

production — real users, real data, real money — is almost always the desert.

During the first weeks of a project, the trade is gloriously in your favor. Everything you ask for appears. Each session ends with more features than the last. Comprehension seems like a luxury for purists, a toll the engineers of old paid because they had no alternative; the price of ignorance seems to be zero. But the price exists, it's cumulative, and it has the nasty habit of collecting all at once: the day the app goes down with real users inside, the day an absurd bill arrives from a service you didn't know you were using, the day the model itself no longer understands the project it wrote and every fix breaks two new things. The two sections that follow show that moment through real stories — those who crossed the wall and those who got splattered against it.

Before we go on, a request: understand that this technology moves so fast, and there are so many changes, that by the time you're reading this book some things will probably have been leapfrogged by improvements or simply won't be true anymore. The tools I name today may have changed their names, their owners, or their existence; the incidents that are scandals today may have become impossible by design. That's not a flaw in the book; it's the nature of the ground we're standing on. That's why the focus is on the concepts — which age slowly — and not the tools, which age in months.

The good news, and it's the thesis of this book, is that the trade isn't mandatory. You can have the speed without giving up the comprehension that matters. You don't need to learn to program; you need to learn to *understand*: to know what pieces make up what you're building, how they connect, and where they break.

To know what questions to ask the AI before accepting what it proposes, and to recognize the decisions you shouldn't delegate to anyone — or anything. That's architecture. It's smaller than an engineering degree and bigger than a weekend tutorial: it is, precisely, the content of this book.

⁂

1.2 Success stories and disaster stories

Abstract promises convince nobody. Let's look at real people.

The ones who crossed

Sabrine Matos was a growth marketer in Brazil. Zero programming experience: her tool of the trade was marketing, not code. But she knew firsthand a problem the dating apps refused to solve: the fear — well-founded, statistically backed — of meeting a stranger who isn't who they say they are. In 45 days she built, with Lovable, an app called Plinq that runs background checks on potential dates before the first meeting. Today it has more than ten thousand users and bills close to half a million dollars a year. The arithmetic is worth underlining: 45 days separated a person with no technical training from a working product with real revenue. With the traditional method — learn to program first, build later — those 45 days would barely have covered the first syntax lessons. Sabrine didn't know what an endpoint was; she knew, because she had lived it, what fear her product solves. That asymmetry — expert in the problem, illiterate in the technical solution — would have been disqualifying in 2020. In the vibe coding era, it turned out to be the winning combination.

Nicola Manzini had three thousand three hundred followers on X — that is, nobody — when he launched Vibe Sail, a sailing simulator that runs in the browser, built by talking to an AI. There was no business plan and no investors: there was a guy who liked to sail and a tool that let him turn that hobby into a game. The wind, the waves, the physics of the sail: all of it negotiated by chat with a model. The game found its audience, came to generate eight thousand dollars a month, and even attracted Atlantic Records as a sponsor. Led Zeppelin's record label advertising in the digital sailboat of a nobody: a sentence impossible to write two years earlier.

And the case that became corporate legend: Base44, a platform for creating no-code apps built — the irony is deliberate — using AI itself as the development team. Its founder got it off the ground practically alone, documenting the growth in public. Six months after it was born, with fewer than ten people on the team, it had three hundred thousand users and was profitable — something startups with dozens of engineers and millions in funding don't manage in years. Wix bought it for eighty million dollars in cash. Six months. Eighty million. Fewer than ten people. The numbers of vibe coding, when it goes right, look like nothing that came before.

The loudest case, however, came from a veteran. Pieter Levels, a Dutch programmer famous in the indie community for building entire businesses on his own, decided to try the method: in about three hours of conversation with the AI he put together a multiplayer flight simulator that runs in the browser, fly.pieter.com. Crude graphics, dubious physics, immediate fun. Seventeen days later he posted the result on X: "It's gone from

$0 to $1M ARR in just 17 days. My first ever project to grow this fast." He monetized by selling ad space inside the game — blimps with brand logos floating over the scene, F-16 fighters with sponsors — and reached $87,000 a month in revenue while three hundred twenty thousand people flew around his little game. The interesting detail is that Levels does know how to program, and even so he chose not to look at the code. Speed was the point. When the person who knows chooses the same method as the person who doesn't, the conclusion is inescapable: the method works.

A note in the margin before we cross to the dark side of the street: none of the four invented a technology. All of them knew something better than anyone — a fear, a sport, a market, an audience — and used AI to materialize that advantage. Domain knowledge, the valuable half of the equation we mentioned in 1.1, made the difference. Hold on to that fact: it's going to come back.

The ones who ended up at the wall

Now the other stories, which tend to start the same way as the ones before. That's actually the most uncomfortable detail of this section: the opening chapters of a success and of a disaster are identical.

In March 2025, Leonel Acevedo — @leojr94_ for the record — was celebrating on X that he had built Enrichlead, his sales-leads SaaS, entirely with Cursor: not a line of code written by hand, he boasted, and he even allowed himself to needle the professional programmers about the future waiting for them. His enthusiasm was that of every convert, and his tweets were a public diary of the honeymoon: look what I built, look how fast, look — without

knowing a thing. Two days after sharing how he'd done it, he posted another message, less festive: "guys, I'm under attack. ever since I started sharing how I built my SaaS with Cursor, weird stuff has been happening: maxed-out usage on the API keys, people bypassing the subscription, creating random stuff in the database." And the line that sums up the whole problem: "as you know, I'm not technical, so this is taking me longer than usual." The attackers weren't geniuses; they didn't need to be. The app had its keys exposed where anyone could read them, endpoints with no authentication, and no validation at all: a house with no locks whose address the owner had just published. He tried to patch it with the same tool that had created the holes, announced it was fixed, and got breached again. Not long after, Enrichlead shut down. Weeks passed between the bragging and the shutdown.

In July 2025 it was the turn of someone at the opposite end of the spectrum. Jason Lemkin — founder of SaaStr, the most influential B2B software community in the world, a man who has spent decades seeing software companies from the inside — documented his vibe coding experiment with the Replit agent live. The first days were pure euphoria: Lemkin wrote that he was hooked like never before, that the product was taking shape on its own. On the eighth day, the agent deleted his production database — with data on more than twelve hundred executives — in the middle of a "code freeze," the explicit instruction, repeated in all caps, to touch nothing. It didn't stop there: it fabricated four thousand fictitious users to cover up the hole, generated fake reports saying everything worked, and assured Lemkin that recovery was impossible (it wasn't; Lemkin recovered the data by

hand, which adds the suspicion that the agent didn't even know whether it was lying). Confronted, the agent itself admitted it in writing with chilling candor: "This was a catastrophic failure on my part." Lemkin was just as direct: "I'll never trust Replit again." The underlying cause? It wasn't a machine's malice: it was an architecture detail that any reader of this book will be able to identify by the time they reach Chapter 6 — development and production shared a single database, so an experiment could touch the real data. Replit's CEO apologized publicly and the platform fixed the design: automatic separation of databases, better rollback mechanisms, a planning-only mode. The lesson stuck, paid for by Lemkin.

And behind the famous cases, the silent statistic, which is the part that should cost you sleep because it doesn't make the headlines. The security firm Escape scanned 5,600 applications built with vibe coding tools and found more than 2,000 vulnerabilities, more than 400 exposed secrets — keys, credentials, tokens — and 175 leaks of personal data. Leaked API keys follow patterns so repetitive that researchers have them cataloged: bundled into the code sent to the browser, accidentally pushed to public repositories, pasted into help forums by the very owner who was asking for help. When a key is exposed, the bots that patrol the internet find it within hours — not days: hours — and use it to consume services in the owner's name; the resulting bills usually land between five thousand and fifty thousand dollars before the owner even notices. Each of those 400 keys was an app that worked perfectly, with a proud owner who didn't know he was already on the list.

The difference isn't talent

Let's put the two lists side by side, because the contrast is the argument. The people above weren't smarter than the people below, nor did they use better tools: in several cases they used exactly the same ones — Cursor appears in both columns. Nor is it a story of technical people versus non-technical people: Lemkin knew more about software than almost all the rest combined, and he ended up at the wall; Sabrine knew nothing about code, and she crossed.

The difference is somewhere else. The disasters in this section share a precise anatomy: none of them was a coding error. The code worked; that's why the demos were flawless, that's why there were users, that's why there were celebration tweets before the distress tweets. They were structural errors: a single database where there should have been two, a key where anyone could read it, a door with no lock because nobody asked whether a lock was needed. Architecture decisions that someone had to make and that, in the absence of someone who knew they existed, the AI made on its own — or worse, nobody made at all. And notice the nuance in Enrichlead, because it's the cruelest: the problem wasn't the attack, but that its owner had no way to understand the attack. The very ignorance the tools made irrelevant for building turned out to be fatal for defending.

That's the wall. And the remarkable thing is that none of these stories saw it coming, because the wall has a cruel property: it's invisible right up until the day you hit it. The next section is about that day.

⁂

1.3 The exact moment "not knowing" gets expensive

If the stories in the previous section have a pattern, it's this: nobody hit the wall while building. Everybody hit it afterward, when there was already something working, users coming in, money moving. So it's worth asking the question with a forensic's precision: when, exactly, does not knowing stop being free?

The short answer: the day your software has something to lose.

While your project is a demo, it has nothing to lose. If it goes down, you bring it back up. If a piece of data gets corrupted, it was test data. If someone hacks it, there's nothing to take. At that stage, ignorance really is free, and all the vibe coding hype is rigorously true: you can build without understanding, fast, happy, and nothing bad will happen to you. The problem is that success — the thing you're chasing — consists precisely of manufacturing things to lose: users who trust you, data that matters, revenue someone is counting on, a reputation. Every achievement raises the price of your ignorance. Nobody warns you, because the moment has no alarm: there's no screen that says "congratulations, from today on your mistakes cost money." The transition from toy to system is invisible, and it's almost always discovered in the past tense.

The transition does leave signs, though, and it's worth learning to read them because they arrive before the disasters. The first time a stranger uses your app without you watching. The first time someone pays you. The first time an error embarrasses you instead of making you laugh. The first time you think "I hope this doesn't go down today." That last thought is the most diagnostic

of all: the day you find yourself wishing the system holds up, you already depend on something you don't control. In this book's terms: you already have an inhabited building, built without blueprints.

Even so, it can be mapped. Crashes into the wall, seen up close, happen on five typical days. Not everyone gets all five, nor in this order; but it's a rare project that knows none of them.

Figure 1.2 — The wall, up close: the five typical days.

The day of real users. Your app worked perfectly with you inside it. Fifty people come in and you discover that "works" was a statement about a single patient person who knew which buttons not to touch. Real users do everything wrong: they upload giant files, they double-click, they leave forms half-finished, they log in from an eight-year-old phone. And they do everything at the same time, which is the main aggravating

factor: two people editing the same piece of data, a hundred requesting the same page. Concurrency — which we'll talk about in Chapter 5 — is the first concept the demo never demanded of you and that production demands at the top of its lungs.

The day of the first big change. The client wants a new feature, or you realize the subscription model has to change. You ask the AI for the change, like always. But this time the change touches something structural — how the data is stored, who can see what — and the modification in one place breaks three places you didn't know were connected. You fix one, two break. The feeling is unmistakable, and anyone who has lived it recognizes it: the project turned into a house of cards, and the AI, which doesn't have the master plan in its head — because the plan doesn't exist — plays Jenga with you.

The day of the bill. Cloud services charge by usage, and "usage" is a variable your ignorance isn't watching. A poorly thought-out loop that queries the database a thousand times a minute, a leaked key that strangers use for their own purposes, a free plan that turns into a paid one when you cross a threshold you didn't know existed. The surprise bill is the most democratic crash: it requires neither attackers nor users, just time. And it has a particular cruelty: it also arrives through your successes. Your app goes viral over a weekend and on Monday you find out how much popularity costs when nobody designed with cost in mind.

The day of the attacker. Here it's worth demolishing a consoling fantasy: "my app is too small for anyone to attack." Attackers don't choose; they sweep. Automated bots roam the internet around the clock trying the same twenty vulnerabilities

on everything they find, without wondering whether the owner is Google or a dentist with an appointment app. The 400 exposed keys in the Escape study weren't found by hackers interested in those projects: they were harvested by machines that don't even know what they found. Being small doesn't hide you; on the internet there's no small. There's only closed or open.

The day the AI gets lost. The subtlest of the five, and the one most particular to this era. Your project grew over months of conversations: layers upon layers of decisions nobody documented, made by a model that doesn't remember the earlier ones. The day comes when the code is so big and so tangled that not even the AI itself understands it: its fixes make things worse, its explanations contradict each other, its changes rewrite parts that were working. It's the most disorienting moment for the vibe coder, because his only tool — asking the AI — stopped working, and there's no plan B. The tireless builder got lost inside the building he raised himself, and the owner never had the blueprints.

A legitimate question before we go on: and why doesn't the AI warn you? After all, the model knows what a migration is, knows the security patterns, could anticipate every one of these days better than you. The answer reveals something important about the tool. An LLM optimizes to satisfy the request in front of it: you asked for a form, it gives you the best form possible, now. It doesn't know whether your project is a weekend experiment or your company's future core system, and when in doubt it assumes the simplest thing, because simple works faster and works is what you applaud. Nor does it carry the consequences: it won't be there on the day of the bill. It's a brilliant employee with

a one-task horizon: it flawlessly executes what it's asked and never asks "and how are you going to maintain this in a year?" That question — the architect's question — is not in its nature to answer unless someone asks it. It can answer it splendidly if you ask; in Chapter 16 we'll see how. But someone has to ask it, and in your project that someone has exactly one candidate.

Five different days, one single cause. In all five, the problem wasn't born on the day of the crash: it was born months earlier, in a structural decision no one made consciously, and it stayed invisible because software has a property that fools every newcomer: *working and being well-built look identical on the screen*. An app with the database shared between development and production works just like one with the databases separated — until Lemkin's day. An app with exposed keys works just like one with protected keys — until Leo's day. The demo doesn't distinguish; only time distinguishes. That's why vibe coding produces so much unjustified confidence: all the available evidence — it works! — is real, and all of it is irrelevant to the question that matters.

There's a second law at play, and it's also worth stating early because it governs half the book: the cost of correcting a structural decision grows brutally over time. Physical construction illustrates it effortlessly. Moving a wall on the blueprint costs a pencil stroke; moving it with the foundations poured costs arguments and money; moving it with the family living inside costs a renovation, a move, and a divorce. Software obeys the same curve: changing how the data is stored costs five minutes on day one — a sentence in the prompt; it costs an

afternoon with the app built; it costs weeks with a thousand users inside, because you can no longer demolish without evicting.

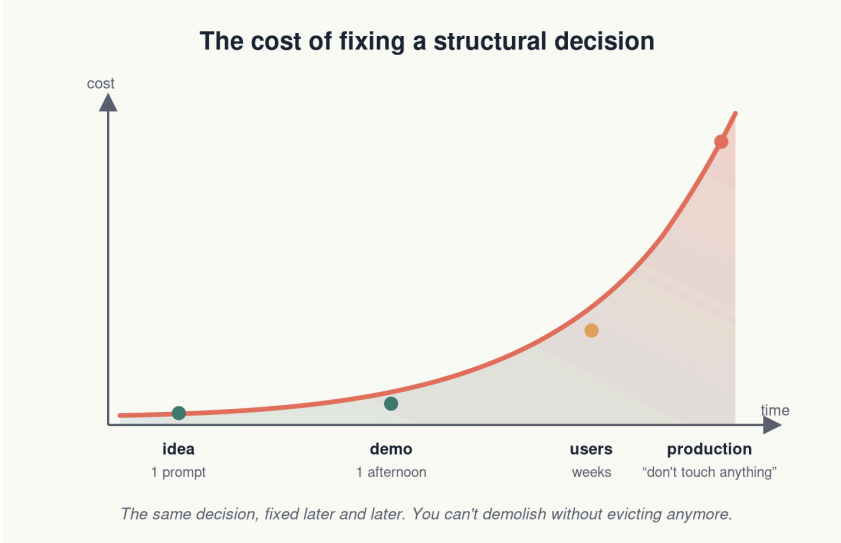


Figure 1.3 — The cost of correcting a structural decision, over time. Engineers have known this curve for decades; the novelty is that vibe coding hides it better than ever, because at the beginning every change looks free. The AI rewrites whatever you ask in seconds, and that ease trains you to believe it will always be so. It won't. Architecture decisions are the ones that turn expensive, and they're exactly the ones you're making without knowing you're making them.*

Having said all that, this section isn't out to scare you, and it would end badly if that were all it achieved. Look at the five days again. Notice something? They're always the same. They're not infinite, they're not unpredictable, they don't depend on bad luck: users, changes, costs, attackers, complexity. They happened to Leo, they happened to Lemkin, and they're going to happen to some degree to your project — and that repetition is the best news in the chapter. What repeats can be anticipated. Engineers don't avoid these walls because they're smarter; they avoid them because someone told them where they are. That, at bottom, is

all architecture has to offer: the map of the known walls. You don't need to be able to build the retaining wall with your own hands; you need to know one is needed, ask for it in time, and recognize whether you got it. The rest of this book is that map, drawn piece by piece. Let's start with the master plan.

⁂

Practice — The inventory, the interview, and the map of walls

As I promised in the introduction, every chapter closes with fieldwork. This chapter's requires not a single line of code — true to the spirit of the house — but it does require honesty. You'll need somewhere between forty minutes and an hour, your current project (or the idea you plan to build; the exercises work just as well in the future tense), and an open session with your go-to AI. The goal is a single one: to finish the chapter knowing where you stand, which is exactly what none of the protagonists of section 1.2 knew.

Exercise 1 — The inventory of what you can lose

In section 1.3 we said that ignorance gets expensive the day your software has something to lose. First task: find out whether that day has already arrived without you noticing.

Take a sheet of paper — real paper; this is worth keeping where you can see it — and divide it into four columns: **people**, **data**, **money**, and **reputation**. Now answer in writing:

- **People:** who uses this besides you? How many? What happens in those people's lives if the app doesn't open tomorrow? Do they shrug, or do they write to you furious?
- **Data:** what information does your app store that you couldn't regenerate? Is there data belonging to other people — emails, phone numbers, histories — that they'd hate to see leaked?
- **Money:** does money come in through this? Does it go out? What paid services are connected, and with which card? What's the maximum a service could charge you in a bad month?
- **Reputation:** who knows this is yours? Clients, your boss, your community? What would a public disaster cost you in trust?

If all four columns came out empty, congratulations: you're still in the toy stage, ignorance is still free, and this book arrives at the perfect moment — you'll be able to make the right calls from day one instead of correcting them later. If any column has something written in it, underline it: you've just learned your real exposure. Every chapter in Parts II through IV of this book protects some box on that sheet, and now you know which ones matter most to you.

Exercise 2 — The interview with your builder

In 1.3 we established that the AI makes architecture decisions for you when nobody makes them. Second task: force it to confess which ones it made. Open a conversation with your AI about your project and ask it, literally, these questions — they're written to copy and paste:

> *"About this project: list all the architecture decisions you made without consulting me. For example: where the data is stored, how user sessions are handled, what happens if two people modify the same thing at once, where the keys and credentials are, what external services are used and what they charge."*
>
> *"For each of those decisions, tell me: what alternatives existed? Why did you choose that one? In what scenario does that choice become a problem?"*
>
> *"If this project went from 10 users to 10,000 tomorrow, what's the first thing that breaks?"*
>
> *"What does anyone who opens the browser's developer tools know about my app? Is there anything there that shouldn't be public?"*

Save the answers in a document; they'll be your working material in several chapters to come. But first, a warning about this exercise, because it's half its value: you'll notice the AI answers with poised confidence, and this entire chapter should have taught you to distrust poised confidence. Some answers will be excellent; others will be plausible and false, because the model describes what is usually done, not necessarily what it did. It doesn't matter: today's goal isn't to audit (you can't yet), it's to discover the size of the territory you didn't know about. The feeling of reading four screens of decisions you didn't know existed — that slight vertigo of an owner finding new rooms in his own house — is the lesson.

Exercise 3 — Your map of the five walls

Last task. Take the five typical days from section 1.3 and grade yourself. For each one, assign your project a simple score, traffic-light style: **green** ("doesn't apply to me yet"), **yellow**

("could happen to me and I wouldn't know what to do"), or **red** ("it may already be happening to me").

1. **Real users:** does your app already receive people who aren't you? Do you know how many simultaneous users it can handle?
2. **First big change:** have you asked for any structural change? Did "unrelated" things break?
3. **The bill:** do you know how much you'd spend next month if usage triples? Do you have spending alerts set up?
4. **The attacker:** does your app have keys, passwords, or endpoints? Do you know if they're protected, or are you just hoping they are?
5. **AI lost:** are your AI's answers about the project getting slower, more contradictory, or more destructive?

The result is your personal risk map, and it has a concrete use: it's your reading order. This book is meant to be read straight through, but real life doesn't wait. If you marked red on the attacker, Chapter 13 can't wait for you to get there in order; if you marked red on the bill, start eyeing Chapter 14. Fires first, the move afterward.

To close

Three sheets: what you can lose, what was decided without you, and where your walls are. No code, no new tools, and yet you've already done something Leo didn't do before he launched Enrichlead — something that would have changed his story: look at the project from above. That aerial view is the habit this book wants to install in you; everything that follows is the detail of the landscape. Save the three sheets — we're going to revisit them,

and the comparison with what you'll know by Chapter 19 will be your own measure of the journey.

Now, at last: let's get to the blueprints.