You know your industry. The world needs the software.

REAL IMPACT
REAL OWNERSHIP
REAL LEVERAGE

TURN YOUR EXPERTISE INTO A REAL TECH PRODUCT

Solve a real problem. Help real people.

Your insight is the advantage.

Build once. Create leverage for years.

- VALIDATE THE NEED
- DESIGN THE RIGHT SOLUTION
- LAUNCH AND SCALE
- EARN AS YOU GROW

THE 10% RULE

Ten hours a week. That's all it takes.

You bring the knowledge. This brings the impact.

The software your industry is missing — built without becoming an engineer.

100+ MAJOR SOFTWARE DEVELOPMENTS ACROSS FOUNDERS AND TIER-ONE COMPANIES

RYAN RICHARDSON

Turn Your Expertise Into a Real Tech Product

The book for the expert who has the idea, the industry, and the network, and cannot get the thing built.

Ryan Richardson

Turn Your Expertise Into a Real Tech Product

The stories in this book are real. Names and identifying details of clients and ventures have been changed or left out to protect confidentiality. Results described are examples drawn from the author's own projects and engagements, not promises; outcomes vary with your situation and what you do with the material. This book is not consulting, financial, or legal advice.

First edition, 2026.

ryan@onwardslabs.io · onwardslabs.io

Who is telling you this

Ryan Richardson has built seven businesses and delivered over a hundred technical projects, from funded startups to tier-one enterprises. He started in data science and analytics, and has done technical work that touched global banks, large technology firms, and government.

He is the founder of Onwards Analytics and Onwards Labs, and the technical co-founder of an enterprise software company he helped take from an idea in an executive's head to a real product with paying enterprise customers. Across his own ventures and his clients' he has done the same thing over and over. Taken a smart person who knows their industry cold, and helped them turn it into a product that ships.

He works with a small number of domain experts at a time, as their technical partner. He lives in Australia.

Contents

The service, in a single page

Already know you want a partner to build it with you?

Then skip ahead. Most of this book is for the ninety nine in a hundred who run the method themselves. But if you want the person who wrote it in the room with you, here is the short version.

I am Ryan Richardson. Through Onwards Labs I partner with a small number of domain experts as their technical co-founder. You bring the expertise the market cannot fake. I take over the build and ship what you could not. Pricing is bespoke, because a days-long rescue and a serious enterprise build are not the same job.

Why hand you the majority? Because a hundred percent of nothing is worth far less than a minority of something that actually makes money. It only pays me if your product works.

1 **Run your idea through our decision engine.** It gives you our recommendation on how to build it, before you spend a dollar. *onwardslabs.io/decide*

2 **Then a thirty-minute conversation.** If it is a fit, we build it together. If it is not, I will tell you. *onwardslabs.io/call*

The rest of the book earns this page. It is here, in a different colour, for the reader who already knows.

This Book Is For You If

- You have an idea you have validated, people want it, and you have no idea how to actually get it built.
- You are spending money on developers or an agency and you are not sure you are getting what you paid for.
- You have been stuck at seventy percent, sometimes ninety, for months, and every fix creates two new problems.
- You want to finally ship the thing instead of watching it drift.
- You want to lead the technology side of your business with confidence, to be the person who drives it, without becoming an engineer to do it.

The gap between you and the people who build your software is costing you more than you think. Not just money. Time. Opportunity. The version of your business, and your life, that is sitting on the other side of closing it.

Foreword

> *The reason this book exists is not strategic. It is personal.*

I have built seven businesses and delivered over a hundred technical projects, from funded startups to tier-one enterprises. I have sat on both sides of the table. As the technical person who could not make the business understand what was actually possible, and as the business person who could not make the technical team understand what actually mattered.

That dual perspective is not a credential. It is a confession. The honest version of that sentence is that I have had a hundred-plus chances to watch the same problems play out over and over, from both sides, and I still find it maddening every single time.

This book exists because it drives me crazy. That is the real reason. Not because I sat down one day and decided the world needed another business framework. Because I have spent most of my career watching smart, capable, well-resourced people achieve a fraction of what they should, and the reason is almost never what anyone thinks it is.

It is not the technology. The technology is mostly fine. It is not the people. There are talented people everywhere. It is not the budget. Most of the time there is enough money to build something real.

I have seen this in startups that should be moving fast and enterprises that have every reason to get it right. In mining, in resources, in financial services, in government. The faces

change. The technologies change. The size of the numbers changes. The pattern does not.

There is a more personal thread under the frustration, and I should name it. When I was starting out I had almost no help. I had the skills and the will, and I just wanted someone to give me a shot, and it took brutally long hours to get one. Nobody sat down beside me and made the hard part make sense. I taught myself the way most people do, slowly and expensively, by getting it wrong. So watching a capable person stay stuck, when I can see exactly what they are missing, genuinely gets to me. What you wish you had is what you want to give. That line runs through everything I do now, including how I hire and who I take on, and it is a large part of why this book exists at all.

So here is what this book actually is. It is a long, structured vent from someone who has been the client writing the cheque and the person cashing it, and who has a set of hard-won observations about why this keeps going wrong and what makes it go right.

It is not a strict method you implement exactly as written and expect the same result every time. Some of it will fit your situation and help quickly. Some will be directionally right and need adapting. Some will not apply to you at all, and that is fine, skip it.

What I can tell you is that getting even half of this right does not produce a marginal improvement. The gap between the expert who can lead the build of their product and the one who cannot is not ten percent. It is not fifty. When you get it right, when the person who owns the vision actually understands the system they are trying to lead, the difference is a real product in the market instead of one that quietly faded into nothing.

I have watched it happen enough times to be certain of it. The rest of this book is the list of things I wish someone had told me earlier. Take what is useful. Leave what is not. And if it saves you even one product from quietly fading away, it was worth writing.

Ryan Richardson

1. This Product Is a Trojan Horse

I am going to do something in this book that most people in my position would never do.

I am going to show you the machine while it runs on you.

Not the polished version. The real one. The one with the wiring exposed, the labels still on, the part where I tell you what I want from you at the end before you have decided whether you like me. Most people who write a book like this hide that part. They pretend the book exists for no reason other than your enlightenment. It does not. And the whole thing works better if I say so on the first page.

So here is the trade. I show you the entire method, in full, no held-back chapter, no locked module, no the-real-stuff-is-in-the-programme. And in return you read it knowing that some of you, by the end, are going to want to build something with me. I will tell you when that part arrives. I will tell you what it is. You can take the method and walk, and that is a completely fine outcome. Or you can decide the person who wrote the method is the person you want in the room. Both of those are wins for me. Only one of them is a secret.

Sit with that for a second, because it matters more than it looks. Knowing the book is built to do a job does not change whether it does the job. If you are the right reader, you will close the last page and want help, not because I tricked you, but because you will have spent a few hours watching something true. If you are not the right reader, you will put it down somewhere in the middle, and that is the machine

working correctly as well. A good book filters as hard as it attracts.

I sat on this for a long time, and I want to be honest about why. What I am going to show you is how I take an expert's knowledge and turn it into a product that beats what agencies charge ten times as much to half-build. When something works that well, the instinct is to guard it. Why write down the recipe so someone can copy it. Why pull back the curtain when the curtain is doing a job.

But I kept watching the same thing happen to people I respect. A doctor with a better way to run a clinic. A tradesperson who has solved a scheduling problem their whole industry lives with. An executive with twenty years of judgement that only exists when they are in the room. Real experts, carrying knowledge the market should pay them far more for, sitting stuck, out-shipped by louder people who know less. The gap between how good they are and what their expertise earns them looks like a canyon. And it has got worse, not better, since everyone started shipping half-built software and calling it done. So I decided I would rather have the recipe out in the world doing good than sitting in a drawer being valuable to no one but me.

The trade, in numbers

Let me put actual numbers on it, because that is how I think about it.

Most people who read this book, call it ninety nine out of a hundred, are going to take what is in here and use it without ever speaking to me. Some will run the whole thing themselves. Some will hand it to a developer they already have and finally know what to ask for. Some will read it,

realise the product in their head is not the one worth building, and save themselves a year and a lot of money. Every one of those is a win. Not a consolation prize. A real win. I would genuinely rather a hundred experts ship good products off the back of this book than keep the method to myself and help nobody. The world gets a few more things that actually work. My name gets quietly attached to the reason. I will take that trade every day of the week, and I mean it.

And one in a hundred, maybe fewer, is going to get to the last page and decide they do not want to run it alone. They want the person who wrote the method in the room with them. That is the other win, and it is the one this whole book is built to earn.

Here is the part I want you to sit with, because it is the actual mechanism. Both numbers are the point. The book does not work if it is built to serve the one percent. It works because it is honestly, almost wastefully generous to the ninety nine. The one percent end up trusting me precisely because I spent the entire book helping the other ninety nine for free, holding nothing back, including the parts most people in my position lock behind a programme. You cannot fake that. You either give it all away or you do not. I am giving it all away. What you do with it is entirely up to you.

You are stuck

Let me tell you where you are, because I have watched enough people stand exactly here that I can describe it without knowing your name.

You have an idea. A real one. You have watched a problem up close for years and you know it can be solved, because nobody around you has solved it properly. Maybe you have

even started. Paid someone. Watched the thing crawl to eighty percent and stall. And somewhere in that, a doubt set in, quiet and corrosive, that maybe building things is just not for people like you.

That doubt is wrong, and it is the single most common reason a good idea dies in a folder. The wall between you and a finished product is not a gap in you. It is a flaw in the way this work is normally done, and I take it apart properly later, in the chapter that is all about it. For now, hold on to one thing. The story that says you are not technical enough to have the thing you can already see is the first lie to put down.

That is what this book is about. Turning what you know, the expertise the market cannot fake, into a piece of software that is genuinely worth something. Not a demo. Not a deck. A real product that works, ships, and makes money while you sleep. And yes, at the end, some of you are going to want to build it with me. I said I would flag that up front. There it is.

A taste of what this looks like when it works

Before I ask you to believe anything, I want to give you the numbers, because the numbers carry the argument better than I can.

A domain expert with a twenty-year enterprise career came to me wanting to turn that career into a product. Not a lifestyle app. A serious enterprise platform, the kind of thing a team of a dozen would normally spend three or four years and two to three million dollars building. We built it for around a hundred and fifty thousand. In its first year it did over three million in revenue. Same ambition. A fraction of the cost, a

fraction of the time. I will come back to how, because the how is the whole point.

A mobile vet spotted a real gap in her industry and did the hard part well. She understood the problem, she had the design thinking right. Then a designer tried to build it and could not. A developer took over and went in circles. Two years. Around thirty thousand dollars. Stuck at eighty percent the entire time, never once at a point where anyone could name what was actually broken. We kept her thinking as the foundation, rebuilt the thing from the ground up, and had it live and solid in weeks for almost nothing. Here is the part that stays with me. A small change used to take her a month. Now she sends twenty-seven major changes and they are actioned, tested, and released overnight.

I automated a chunk of my own consulting business, packaged it into a product, ran ads to it, and got crickets. Cool technology, genuinely unique, tiny market, nobody driving it including me. It stalled despite being good. I am telling you about a thing of mine that did not work in the same breath as the ones that did, because I would rather you trust the pattern than the highlight reel.

And there was a deal I badly wanted, a coach with deep knowledge and interesting AI, where the honest answer turned out to be no. The technology could not get accurate enough in exactly the place it needed to be accurate. So I said no. Did not take the money for a two-year mess. Paused it. I will tell you that whole story later, because the fact that I killed it is the most important thing I can show you about how I work.

Range from a days-long rescue to a hundred-and-fifty-thousand-dollar enterprise build. Different industries, different budgets, different starting points. Same underlying move every time.

I am giving you the failures in the same breath as the wins on purpose, and I want you to notice that. Most people selling you a build show you the trophies and quietly file the losses. I am handing you the loss where I ran ads to good technology and heard crickets, and the loss where I walked away from a deal I wanted, because those two tell you more about how I work than the three-million-dollar number does. The wins prove I can build. The losses prove I will tell you the truth about what to build, and when not to. You want both from the person who takes over your product. A builder who has never killed anything has either not built much or is not telling you the whole story.

What you are one product away from

There is a phrase I keep coming back to, because I have watched it come true for the people in those stories.

You are one product away.

One product away from your expertise earning while you sleep instead of only when you are in the room. One product away from serving the hundredth customer as easily as the first. One product away from an asset you own, that has a value on its own, that someone could invest in or buy, instead of a job that resets every Monday. One product away from the version of your work, and your life, that is sitting on the other side of getting this one thing built properly.

But, and this is the part that matters more than any other line on this page, only if it is built right. A product built wrong is worse than no product. It is the two-year mess. It is thirty thousand dollars and eighty percent and always almost done. So the whole game is the difference between the product that

ships and the product that traps you, and this book is going to spend most of its pages on exactly that difference.

The big idea

Here is the thing all of those stories are secretly about, and it is the one idea I want you to carry out of this book above every other. I have a name for it, because naming a thing is how you keep hold of it. I call it the ten percent rule.

Almost all of any product already exists. The part that is genuinely new, the part that has never been built before and is uniquely yours, is about ten percent of the whole thing. The other ninety is proven machinery the world already built, and you should refuse to pay to build it again. So the entire game is not learning to build. It is choosing the right ten percent, spending everything you have on that, and being disciplined enough to refuse the ninety. Get that one call right and a build that should cost millions and take years costs a fraction and ships in weeks. Get it wrong and you pour your money into rebuilding the ninety by hand, which is exactly what the industry is quietly set up to sell you.

That is why building something that works is a process problem, not a technology problem. The technology is the commodity. The scarce, valuable thing is the judgement to see the ten percent, and that judgement is yours, not the developer's, because it comes from knowing your industry cold. Keep things simple, use proven tools instead of inventing new ones, and plan for the fact that you will be wrong about something so you can move instead of starting over. The developers who leave a product stuck at eighty percent are usually not bad developers. They just never had that discipline, and they were building all one hundred percent every time.

That is the worst place to be. Not a clean failure you can learn from. A thing that is always almost done.

The whole book is going to take the ten percent rule apart and show you the underside. How the incentives in the industry got tuned against you. Why a product, of all the things an expert could build, is the one worth building. And what the fix looks like when someone runs it properly. I am not going to hand you a rigid blueprint you follow to the letter and get the same result every time, because that book would be lying to you. I am going to hand you the way I think, tested across a lot of builds, so you can tell the difference between a product being built well and a product being built to keep you paying.

Watch for the moment you start to trust me

One more thing before we move.

Somewhere in the next few chapters, you are going to notice yourself starting to trust me. It might be a number. It might be the vet and the twenty-seven changes. It might be the fact that I keep telling you which of my own bets failed. Whatever it is, notice it. Notice which page it happened on.

Here is the part almost nobody in my position would admit. That moment is built on purpose. I am arranging proof and honesty in an order designed to earn your trust, and I am telling you that while it is happening.

And it is going to work anyway. It will work because it is true. Every number in this book is real. Every failure I describe is one I had. I am handing you the blueprint of the horse before you step inside it, and the horse still works,

because a real product and a real book both do the same thing. They do what they say and say what they do.

So watch for the moment. When it comes, ask yourself one question. If this person is right about why I am stuck, what would it be worth to stop being stuck?

What this book does, in order

Since I am showing you the machine, here is the shape of it so you can see each piece coming.

Next I tell you how I got here, because you should know whether the person telling you to keep it simple has earned the right to say it. A top data scientist who loved the hard, useless maths, who learned the hard way that being fast and good is punished in a service business, and who walked toward products on purpose.

After that, why a product beats every other thing an expert could build. Consulting, content, courses, and the one asset that takes your judgement and makes it work without you in the room.

Then the real reason you are stuck, taken apart properly. The incentive that complicates. The seventy percent that gets reinvented for no reason. The ninety-nine-percent trap. And the discipline that separates a build that ships from a build that traps you.

Then the method itself, generously, the way I actually work, so you could run it yourself if you wanted to. And near the end, the part where some of you will want help running it, said plainly, in the place it belongs.

By the last page you will know how this is done. You will know what your product should look like. And you will know the exact gap between where you are now and where you

could be, so you can decide what to do about it, alone or with me. That is the trade. I show you the whole machine, and you decide what to do with it.

Before I earn that trust, though, you should know who is asking for it. So let me tell you how a top data scientist who loved the hard, useless maths ended up here, telling you to keep it simple.

2. The Accidental Builder

I never set out to build products for other people.

Before any of this, before I had partnered with a single expert or shipped a single thing that was not mine, I was a data scientist. That was the whole shape of my working life, and for a long time I could not have told you what it would look like if it stopped being that.

I was good at it. Not modestly good. I trained in the hard maths, the interesting problems, the kind of work where you can spend a month chasing an idea that might not pay off and the chase itself is the reward. I loved it. I still do. Give me a genuinely difficult modelling problem and a quiet afternoon and I am happy in a way that is hard to explain to people who do not do this work.

I need you to know that up front, because everything I am going to tell you for the rest of this book is going to sound like I am against complexity. I am not. I am a person who finds complexity beautiful telling you it will ruin your product. That contradiction is the whole point. I know exactly how seductive the hard version is, because it seduces me.

I should tell you plainly how my own head works, because it is the engine under all of this and it cuts both ways. The way I am wired is restless. I generate ideas constantly, I can run at something incredibly hard, and I can hold a whole system in my head at once, every moving part and how it connects. That wiring is a hundred percent why I am good at this work. It is also, left unmanaged, a liability. The same engine that lets me see the whole map can send me tearing off in the wrong

direction, or straight into building something far more elaborate than the problem ever needed. On my own I over-complicate. I reach for the technically perfect version when the plain one would have done. Simplicity, it turns out, is genuinely harder than complexity, and it is not my natural setting. I will come back to how I fixed that, because the fix is the reason I work the way I do now, but hold the shape of it for a moment. The thing that makes me good is the same thing that needs a harness.

The slow burn nobody warns you about

Before I get to any of that, I want to tell you what the years before it actually cost, because the myth is a lie and the lie does damage.

The myth is the founder who quits the job, sleeps on the office floor, and emerges eighteen months later with a finished thing. That is not what it looks like for almost anyone I know, and it is not what it looked like for me. What it actually looks like is a graveyard of half-built things squeezed into nights and weekends around a job, a mortgage, and a family. You get a two week burst of real momentum, the thing starts to take shape, you can nearly taste it, and then a kid gets sick, or work goes sideways, and the whole lot goes out the window. Weeks pass. You come back to code you no longer remember writing. Huge effort, and nothing moves.

That is the real cost, and it is not dramatic, which is exactly why nobody warns you about it. It is an idea sitting in your head for five years. Ten, in some cases. Going nowhere. Not because you are lazy and not because the idea is bad, but because you never had a clear enough run at it, and every time you got close, life closed the window. I know that feeling from the inside, and I suspect it is the reason you picked this book

up. If it is, I want you to know I am not writing to you from some other planet where things got built easily. I am writing from the same graveyard.

What the interesting work really pays

Here is the first thing the work taught me, and it took years.

The interesting problems are rare. Maybe one job in twenty is the kind of deep, novel, genuinely hard problem I got into this for. The other nineteen are reporting. Cleaning data. Wiring one system to another. Building the same dashboard a slightly different way for a different team. That is not a complaint about the field. That is the field. The exciting one-in-twenty sits on top of nineteen jobs of plumbing, and the plumbing is where the money and the scale are.

So if you want to build something bigger than yourself, you cannot build it on the one-in-twenty. You have to get good at the nineteen. You have to make the repeatable work fast, reliable, and cheap enough to do at volume. That was the first real turn in my career. I stopped chasing the beautiful problem and started building the machine that does the boring problems well.

I built a delivery team to do it. A real one, offshore, engineers who were genuinely excellent, and we delivered reporting and development work across major mining companies. Not a slide about a team. A team. People I hired, trained, backed, and paid, running real builds for real enterprises with real deadlines. I mention that because a lot of people who talk about offshore delivery have either never done it or tried it once and got burned. I ran it for years. I know what it takes to make it work and I know exactly where it falls apart, and both of those turn up later in this book.

Here is the thing most people get wrong about that team. They assume offshore means cheaper and worse, a trade you make when the budget is tight. That is a broad assumption about a talent market of hundreds of millions of people, and broad assumptions about markets that large are not a strategy. There are engineers in that team who would outbuild most of what you would find paying triple the rate locally. What made them good was never the geography. It was the context, the ownership, and the structure I gave them, and those are the same three things that make anyone good anywhere. I will come back to that, because it is one of the levers that makes a fast, affordable build possible in the first place. For now, just hold the fact that I ran a real team, at real quality, for years, before I ever told anyone else how to build.

The bug that pays the bills

And then the work taught me the second thing, which is the one that changed everything.

In a service business, efficiency is a bug.

Let me show you what I mean, because it sounds like a paradox and it is not. It is just uncomfortable.

We got good. Good enough that a client would come to us with a problem they thought needed a three-month program with fifty workshops, and we would look at it and see that the thing they needed was a five-minute fix. So we would do the five-minute fix. Happy client. Genuinely happy. Problem solved, fast, for almost nothing.

No money.

Because in a service business you are paid for hours, and the five-minute fix bills five minutes. The three-month program bills three months. The agency that quotes the fifty

workshops for a problem that needs one conversation earns more than the one that solves it on the call. Not because they are crooks. Because that is how the work is priced. The incentive sits underneath everything, and it points in one direction. Make it bigger. Make it slower. Make it more custom than it needs to be. The better you get at solving problems quickly, the less you earn per problem, and the harder you have to run just to stand still.

I want to be careful here, because this is the part that people mishear. I am not telling you every consultant is running a con. Most of them are not. Most of them are good people trying to do good work and get home to their kids. The problem is not the people. The problem is the machine they are standing inside. When the price is the hours, the machine rewards the hours, and even honest people bend slowly toward the thing that pays. I know because I felt it bend me. There were days I caught myself dreading how fast we had solved something, because fast meant thin, and thin meant a hard month.

That is a bad feeling to sit in for long. You are being punished, in the only currency the business speaks, for being excellent at the actual job.

There is a darker version of that feeling, and I am going to be honest about it because it is the thing I most needed to walk away from. It is not just that speed pays badly. It is what the model quietly encourages you to want. The ugly incentive underneath a service relationship is to keep the client feeling stuck. To keep them believing the thing is too hard, too specialised, too dangerous to touch without you, so that they keep paying you to stand between them and it. I never sat down and decided to do that. Nobody does. But I could feel the pull of it, the small voice that notices a client is dependent and reads that as job security rather than as a problem to

solve. How long can I keep you paying by keeping you believing only I can save you. That is the question the model puts in your mouth, whether you want it there or not.

Here is the thing I came to believe, and it is the hinge my whole career turned on. People only stop being taken advantage of when they learn it was never that hard. Not never hard for them to do alone, but never the impenetrable black art they were sold. The moment a client understands the shape of what you do, the mystique collapses, and with it collapses the model that was profiting from their confusion. I decided I did not want to be the person on the wrong side of that. I did not want a business whose quiet engine was other people's helplessness. That is a large part of why this book exists, and why it gives away the method instead of guarding it. The thing I am selling only works if it makes you less dependent, not more.

Let me give you the sharpest version of it I ever lived. A team I ran took a process that was delivering two or three fixes a year at high cost and rebuilt it to deliver seventy fixes at a fifth of the cost. Output up thirty times. Cost down eighty percent. A genuinely good result by any honest measure. And the response from the business side was to ask why the last fix took three hours when it should have taken one. Not thank you. Not this is extraordinary. Why was one of the seventy slightly slower than expected. I have received an email, in a moment like that, that said there are significant issues and I need a developer in front of me because that is the only way we will fix this. It was a ten-second fix. What it needed was a phone call. Instead it arrived as a formal escalation demanding a body in a chair. That is what the wrong incentive does to a relationship. It teaches both sides to measure the thing that is easy to see, hours and presence and the

appearance of effort, instead of the thing that is hard to see and actually matters, which is whether the work is any good.

I do not tell you that to complain about a client. The client was not a bad person. The client was standing in a machine that had taught them, over years, that speed and presence are how you tell if technical work is healthy. They are not. But they are visible, and the invisible thing is uncomfortable to lead, so people reach for the visible proxy. That is the machine, not the man, and it is exactly the machine I decided to step out of.

Why I walked toward products

So I did the maths on my own life, which is the same maths this book is going to ask you to do.

If value and hours are welded together, you are capped. There are only so many hours, and the better you are, the fewer of them each problem needs, which means the ceiling drops as your skill rises. That is a career that fights you. I did not want to spend twenty years running harder up a slope that gets steeper the fitter you get.

Scaling a service business is brutal for exactly this reason, and I do not think people say it plainly enough. To grow, you need more hours, which means more people, which means more management, more quality control, more clients pushing back on price, more of your time spent running the machine instead of doing the work you were good at. Every unit of growth costs roughly what the last one cost. There is no point where it suddenly gets easier, because the thing you sell is time, and time does not compound. You just add more of it, and more of the overhead that comes with it. The most

efficient consultancy in the world is still a treadmill. A faster treadmill, sure. Still a treadmill.

I watched good firms survive that reality the only way the model allows. By over-complicating. The three-month program for the five-minute fix is not always cynical. Sometimes it is survival. The firm has learned, correctly, that solving the problem quickly is a way to go broke, so it dresses the quick solution in enough process to justify the invoice. The client pays for the theatre because they cannot tell the theatre from the work. And around and around it goes, everyone slightly resentful, nobody quite able to name why, because the incentive doing the damage is invisible and everyone is just trying to make a living inside it.

A product breaks the weld. A product is priced on the value it delivers, not the time it took to build. Build a good one and it earns while you sleep, serves the hundredth customer as easily as the first, and stops being a thing you do and starts being a thing you own. The fast, good version is not punished. It is the entire point. Speed and quality make the asset more valuable instead of less. For the first time the incentive pointed the same way as my instinct to keep things simple and ship.

And here is the part that made it feel right rather than just smart. In a product built with a domain expert, nobody in the middle gets screwed. The expert brings the knowledge the market cannot fake. I bring the build. The thing that gets made is genuinely good because being genuinely good is what makes it sell, which is the opposite of the service model where the mediocre-but-slow version quietly pays better. When the incentive and the honesty point the same way, you can stop managing the tension between doing right by the client and keeping the lights on. There is no tension. Doing right by them is the business.

So I made the turn on purpose. I filtered the consulting down to almost nothing, kept only the occasional high-profit piece that walks in the door, and pointed the rest of my time at partnering with experts to build real products. Onwards Analytics, the business, is the delivery muscle behind that. Years of running real teams on real enterprise builds, now aimed at people with an idea and no way to build it.

I will admit the less strategic truth of it too. What I actually built is a hobby I have productised. I am almost compulsively fascinated by other people's ideas and industries, by the mechanics of how a business or a trade actually works under the surface. Sitting with someone who knows a world I have never seen and watching the shape of a product emerge from it is, genuinely, the thing I cannot stop doing. So I built a business whose whole point is that I get to do it all day. That is not a noble motive and I am not going to dress it up as one. It just happens to line up with what people need, which is the best kind of luck to have.

The harness

Now I can close the loop I opened earlier, about the engine that cuts both ways.

Left entirely to myself, the wiring that makes me good would also run me off a cliff. I would chase the interesting idea past the point of usefulness. I would build the elegant, technically perfect version of something nobody asked for. I would over-complicate, because complexity comes naturally to me and simplicity does not. That is not a confession of weakness. It is just an accurate map of the machine, and you cannot point a machine in the right direction until you know how it actually behaves.

So the resolution, and this is the real point, is that the engine needs a harness. The harness is two things. It is systems, the ways of working that keep the build pointed at the simple, shippable version instead of the beautiful sprawling one. And it is people, specifically a partner who owns the simplicity when I cannot. When someone whose instinct is to strip a thing back is sitting across from me, they pull me down to the right level before I have built three months of cleverness nobody needed. That is not a nice-to-have in how I work. It is load-bearing. The reason I believe so hard in the partnership model in this book is not that I read it in a book of my own. It is that I have watched it save me from myself. The engine is the edge. The harness is what turns the edge into something that ships.

The honesty engine

There is one more piece, and it is the one I care most about, so I am going to be plain about it.

The reason I can tell you when something will not work is that I am not paid by the hour to keep it going.

Think about what that does to the advice you get. When your builder earns more the longer it takes, every problem becomes an opportunity for more billing. When your builder owns a piece of the outcome and only wins if the thing actually works, the incentive flips. Now the honest answer and the profitable answer are the same answer. Now telling you a feature is not worth two years is not me talking myself out of money. It is me protecting the thing we both own.

That is the honesty engine, and it is not a personality trait. I would love to tell you I am just a more honest person than the agency that got you stuck. I am not. I am a person

standing in a machine that rewards honesty instead of one that punishes it. Put me back in the hourly model and I would feel the same slow bend I felt before. The model makes the man here, not the other way around, and I would rather you trust the model than trust me.

So when I tell you, later, about the deal I killed even though I badly wanted it, do not read it as me being noble. Read it as the incentives working. That is the whole design. Build in such a way that the truth is also the profitable thing to say, and you will find yourself telling the truth a lot more easily.

Which brings us to the real question. Of all the things an expert could build to break free of the hourly trap, a course, a book, a bigger consultancy, why a software product? Why is that the one worth the trouble?

3. Why a Product Beats Everything Else for an Expert

You have knowledge that took years to build and that most people cannot fake.

The question this chapter answers is what to do with it. Because there are several honest options, and most experts pick one of the weaker ones by default, not on purpose. They fall into consulting because someone asked them to. They start posting because everyone says to build an audience. They build a course because a course felt like the productised thing. All of those can work. None of them do what a software product does.

I am going to walk through each one, plainly, and show you exactly where it caps out. Not to talk them down. To show you the ceiling, so you can see the thing that has no ceiling in the same shape.

What you are trying to escape

Start with the trap, because every option is really a way of answering it.

Your expertise, right now, is trapped inside you. It only exists when you are in the room. When you stop, it stops. When you sleep, it earns nothing. When two clients want you at once, one waits. The value is real and it is yours, and it is welded to your presence and your hours. That weld is the ceiling. Everything good about your career and everything

limiting about it comes from the same fact. It runs on you being there.

So the real question behind why a product is this. Which asset takes your judgement, the thing only you have, and separates it from your presence, the thing you cannot scale? Rank the options on that single test and the answer stops being a matter of taste.

Consulting scales you down, not up

Consulting is the obvious move and the one most experts are already in. It pays well. It respects the expertise. It feels like the natural home for someone who knows a lot.

It is also the tightest cage of the lot, and I say that as someone who ran a consulting business for years. Consulting is your judgement, sold by the hour, delivered in person. Every good thing about it, the rate, the respect, the direct relationship, comes with the weld still on. You are still trading time for money, just at a higher price for the time. And as we saw in the last chapter, the model punishes you for being efficient. Get faster and you earn less per job. Get better and the good problems need fewer of your hours. The ceiling does not just exist. It gets lower as you get better, which is a strange and cruel shape for a career to have.

Consulting is a fine way to earn. It is a poor way to build something that outlives your calendar.

Content builds an audience, not an asset

So people say build an audience. Post. Show up. Become known.

There is real value in that and I am not going to pretend otherwise. Attention is useful. But content is a treadmill with the belt set to Monday. You post, it works, and then it is Monday again and the audience wants the next thing. The asset you are building is attention, and attention has to be re-earned constantly or it drains. Stop posting for three months and watch what happens. The best content people are not resting on an asset. They are sprinting on a belt that never stops, and the day they stop, the numbers start falling.

Content can be a brilliant front door. It is a bad house to live in. It feeds the machine. It is not the machine.

A course sells the knowing, not the doing

A course feels like the productised answer, and it is closer. You take what you know, record it once, and sell it many times. The weld between value and your presence finally loosens. That is real progress and I do not want to wave it away.

But look closely at what a course sells. It sells the knowing. It hands the buyer your understanding and then leaves them to do the hard part, which is the doing, alone. Most people who buy a course do not finish it, and most who finish it do not act on it, and that is not a failure of the course. It is the nature of the thing. Knowledge handed over is still work waiting to be done. A course scales your teaching. It does not scale your judgement into the moment the buyer actually needs it.

There is a version of this that is genuinely good and I have seen it change lives. It is just a rung below the last option, and I want you to see why.

A product scales your judgement into the moment of need

Here is the thing a software product does that none of the others can touch.

It takes your judgement, the actual decision-making, the thing you do in your head that clients pay for, and it bakes that judgement into something that runs on its own. Not your teaching. Not your presence. The judgement itself, working, at the exact moment the user needs it, without you in the room.

Think about what the expert really sells. Not facts. Facts are free now. What you sell is the pattern in your head that turns a messy situation into the right next move. The junior sees a spreadsheet. You see, in a glance, the three things that matter and the twelve that do not. That glance is the asset. A course describes the glance. A product performs the glance, for every user, every time, while you sleep.

That is the whole difference, and it is enormous. Consulting rents your judgement by the hour. Content advertises it. A course explains it. A product embodies it. The vet I mentioned did not sell a course on how she thinks about her patients. She built a product that carries how she thinks into every appointment, so the value shows up whether or not she is on the call. Her judgement now works while she sleeps. That is what took a thing that used to bill by the visit and turned it into an asset that runs on its own.

The asset test, run properly

Let me put the four side by side on the one test that matters, and be honest about each.

Consulting. Does it separate your judgement from your presence? No. It sells them as one thing.

Content. Does it separate them? Partly, but the asset is attention, and attention drains the moment you stop feeding it.

A course. Does it separate them? Yes, mostly, but it separates the knowing, and leaves the doing with the buyer.

A product. Does it separate them? Fully. It takes the judgement, embeds it, and runs it at the point of need without you. And unlike the others, it becomes a thing you own rather than a thing you do. It has a value on its own. Someone can buy it, invest in it, or partner on it, because it exists independently of your Tuesday.

That last point is the quiet one, so let me say it straight. Every other option is a job with a better rate. A product is an asset with a value. You can walk away from a product and it keeps working. You cannot walk away from your consulting for a month without your income walking away with you.

Here is a way to feel the difference in your gut. Picture selling each of these. Can you sell your consulting practice? Not really. Strip you out of it and most of what a buyer wanted walks out the door with you, because the practice was you. Can you sell your audience? Barely, and the buyer knows it drains the moment you stop posting. Can you sell a course? A little, but they are buying a recording of you teaching, and its value fades as the material ages and you are no longer behind it. Can you sell a product? Yes. Cleanly. Because the product is not you. It is your judgement, extracted and running on its own, and a thing that runs on its own is a thing that has a price. That is what an asset means. It is the difference between owning a business and being self-employed with good margins.

This is also why the way I work with experts is a partnership in the product rather than a bill for my hours. If the thing we build together is a real asset, then a share of that asset, growing, is worth more to both of us than a fee for building it. I would rather own a slice of something that works than the whole invoice for something that does not. But that is a conversation for much later in the book, and I only raise it here so you can see that the asset logic runs all the way through. A product is the one thing an expert can build that a second person can rationally want a stake in. Nobody wants a stake in your Tuesday.

The honest catch

Now the part I will not skip, because I told you I would not hide anything.

A product is harder to build than any of the others. A course you can record in a fortnight. A product takes real building, and if it is built wrong it becomes the eighty-percent trap I keep coming back to, which is worse than never starting. The upside is the biggest of any option, and so is the way it can go wrong. That is exactly why the how matters so much, and why most of this book is about the how rather than the why.

But do not confuse hard-to-build with not-worth-building. The reason a product is worth the trouble is the same reason it is trouble. It is the only one of these that takes the best thing about you and makes it work without you. Everything else keeps you in the room. This is the one that lets you leave, and finds the room still working when you get back.

There is one more thing I want to be honest about, because it changes who this book is for. The product needs

you more than any of the other options do, not less. That sounds backwards. Let me explain it, because it is the most important sentence in the chapter.

A course can be generic and still sell a bit. A product cannot. A product only works if it carries real judgement, and the only place that judgement exists is inside the expert who owns the pain. I learned this the expensive way. I built a genuinely clever product out of my own work, ran ads to it, and heard nothing, because clever technology with nobody who owns the pain driving it is just a nicely built answer to a question nobody was asking loudly enough. The technology was not the problem. The missing driver was the problem. Which means the expert is not the customer of this process. The expert is the hero of it. The whole thing runs on you and dies without you, and if you were hoping to hand over an idea and walk away, this is the wrong book. The good news is that if you have watched a problem for years and cannot stop thinking about it, you are exactly the fuel this needs.

So if the product is the right asset, and you are still stuck, the problem is not the asset and it is not you. The problem is in how it gets built. That is the thing I want to take apart next, because it is the real reason you are stuck, and it is almost never what you think it is.

4. The Real Reason You're Stuck

You have the idea. You know it works, because you have watched the problem up close for years and nobody has solved it properly. You might even have started. Hired a developer. Paid for a design. Watched a prototype get to eighty percent and then just, stop.

And somewhere in there a quiet story started running in the back of your head. That maybe you are not technical enough. That you should have learned to code. That you picked the wrong developer, the wrong agency, the wrong stack. That people like you, who know an industry but not software, are not meant to build things.

I want to take that story off the table on the first page of this chapter, because it is wrong, it is expensive, and it is the reason most good ideas die in a folder.

You are not stuck because you are not capable. You are stuck because the standard way of building software is tuned for the wrong thing.

That is the whole book in one sentence. Everything else is detail.

Most of the people you would hire to build your product make more money the longer it takes and the more complicated it gets. That is not a conspiracy. It is how the work is priced. An agency that quotes a three-month program with fifty workshops earns more than one that does the actual thing you need in a week. I ran a consulting business for years and watched this from the inside. We got good enough that

we would do the five-minute fix a client actually needed instead of the drawn-out program. Happy clients. No money. In a service business, efficiency is a bug.

So the machine you are plugging your idea into is tuned to make things bigger, slower, and more custom than they need to be. And you, on the outside, reading a list of features you do not understand, have no way to tell necessary complexity from the kind that is just running up the meter.

The trap runs in a technical person's head too

Here is a specific version of it. I trained as a data scientist. The hard maths, the interesting problems. So I can tell you exactly how a technical person thinks, because I am one.

Give a data scientist the choice between a ninety-nine percent solution that takes two years and might not work, and a ninety-five percent solution that ships in two days, and they will pick the ninety-nine every time. Not because they are lazy. Because they are trained to. Scores, not business. Almost nobody stops to ask whether that last four percent is worth two years, or whether the ninety-five was already enough to change your life.

It usually was. People wildly underestimate the ninety-five.

I have watched this exact trap sink a company. The founder was the ex-chief-operating-officer of a business that had gone under, and when I looked at why, the shape was familiar. A data-science problem, approached by people optimising for the score instead of the outcome. Ninety-nine percent custom, two years in the making, still not shipped, while the ninety-five percent version that would have won

customers sat unbuilt because it felt too simple to be worth building. They wound themselves up in complexity until the product was late, expensive, and beaten by something rougher and faster. That is not a story about bad engineers. Those were good engineers. It is a story about aiming at the wrong number.

I am a data scientist telling you the data-scientist trap is real, and that I chose the opposite path on purpose. That is not me being humble. It is me telling you where the bodies are buried, because I helped dig some of the graves.

It is a process problem, not a technology problem

The truth underneath all of this is unglamorous.

Building something that works is not really a technology problem. It is a process problem. Keeping things simple, using proven tools instead of inventing new ones, and planning for the fact that you will be wrong about something. The developers who leave a product stuck at eighty percent are usually not bad developers. They just never had that discipline. They built everything custom, sharpened the wrong four percent, and small things piled up until nobody could point at what was actually broken. That is the worst place to be. Not a clean failure. A thing that is always almost done.

I want to sit on the process point, because it is the load-bearing idea and it is the one that gets skipped.

When your build went wrong, you probably reached for a technology explanation. Wrong framework. Wrong developer. Should have used the other platform. Almost every time, that is not it. The technology was fine. The tools that build ninety-

nine percent of products are boring, proven, and freely available. What went wrong was the process wrapped around them. Nobody kept it simple. Nobody used the proven thing when they could have. Nobody built it expecting to be wrong. And so the small things piled up, one reasonable decision at a time, until you had a system nobody could hold in their head and a product that was always almost done.

That reframe matters for you personally, so let me say it directly. Your developers were probably not idiots and you were probably not fooled. You all just ran a process built to complicate, and it complicated. The shame you have been carrying, the quiet story about not being technical enough, is aimed at the wrong target. It was never a smarts problem. It was a process problem, and process is fixable in a way that intelligence is not.

Most of a product is not even unique

Here is a fact that should make you feel better and probably makes some developers uncomfortable.

Around thirty percent of your product is genuinely unique. The rest is proven tools, stitched together well. Logins, payments, storing data, sending an email, showing a list, letting someone upload a file. Every product does these. They have been solved a thousand times, and the solutions are sitting there, reliable and cheap, waiting to be used. The unique part, the actual expertise, the thing only you bring, is the thirty percent that sits on top.

This is why a good build is fast and affordable, and why a bad one sinks you. The fast builder spends almost all their effort on your thirty percent and reaches for the proven, boring solution for everything else. The builder who got you

stuck rebuilt the seventy percent from scratch, custom, because it felt more thorough, and drowned in maintaining a hand-built version of a thing that already existed and worked. Same product. One path is weeks. The other is two years and a mess.

So when someone quotes you a build, the real question is not how clever are they. It is how much of my seventy percent are they about to reinvent. The more they want to build custom, the more the meter runs and the more places there are for the small things to pile up.

I will add one caution here, because there is a place where the rules genuinely change, and if I do not name it you will be right to distrust the rest. Enterprise is a different sport. When you are selling into large organisations, the thing is stricter, more customised, and dragged through arbitrary hoops that have nothing to do with whether the product is good, security reviews, procurement, compliance tracks that take months. The users are often mandated to use what you build, which sounds like a gift and is actually a trap, because a mandated user who finds the thing confusing cannot walk away and tell you why. They just quietly resent it. So the interface has to be even simpler, not less, and the real work becomes compressing a twenty-year career into software clean enough that a mandated user adopts it without a fight, while you clear a queue of hoops that add cost and no value. That is why the enterprise build I mentioned cost a hundred and fifty thousand rather than almost nothing. The goal demanded it. The point is not that everything should be cheap. The point is that the cost should be driven by the goal, not by someone reinventing the seventy percent that never needed reinventing. Big budget for a big goal is fine. Big budget because the meter is running is the thing to catch.

The six-thousand-page book

Let me give you the analogy I use, because it lands harder than the argument.

Imagine a six-thousand-page book on a subject you care about. It would be more complete than any book you own. More correct. More content. It would contain everything.

It would also be worthless to you, because you would never read it.

A book's value is not its completeness. It is completeness traded against speed, clarity, and fit. The right two-hundred-page book that you finish, understand, and use beats the six-thousand-page one that sits on a shelf being technically superior. Value is not exactness on its own. Value is exactness times speed times clarity times fit, and the perfectionist optimises the first one while destroying the other three.

Software is no different. The ninety-nine-percent, everything-included, two-year product is the six-thousand-page book. It is more correct and less valuable. The ninety-five-percent version that ships in weeks, the one your users understand and adopt, wins, because it scores well on all four axes instead of maxing out one and starving the rest.

I will be honest that this analogy is a little self-serving, because you are holding the two-hundred-page version of it. This book could have been six thousand pages. It could have listed every tool, every edge case, every exception. It would have been more complete and you would have put it down. So I made it the ninety-five-percent version on purpose, the one that ships and gets used, because I would be a hypocrite to preach the short clear thing in a bloated long one. The book is the argument, demonstrating itself.

Plan for the fail

Now the last device, and it is the one that separates how I build from how the person who got you stuck built.

I have done this more than a hundred times. Different industries, different budgets, wins and kills both. That volume is not a brag, it is the actual point, because it changes what you can see. When you have watched that many builds, you stop believing there is one right way and you start seeing the pattern underneath all of them. Which decisions tend to be wrong. Where the small things usually pile up. What almost always needs to change after real users touch it.

The person who has done it once believes the opposite. They have the one way, the way that worked their one time, and they hold it like scripture. Here is the trap in that, and it is a real trap because it looks like wisdom. A single success is survivorship bias. That founder who built a hundred-million-dollar company on one particular approach might have found the best path. Or they might have hit the lotto and are now certain the winning numbers are a strategy. You cannot tell from one result. One success proves the approach did not fail that time. It does not prove it was right.

So I build the opposite way. I assume the plan is wrong somewhere. I do not know where yet, but somewhere, and the honest move is to build so that being wrong is cheap instead of catastrophic. Keep the stacks flexible. Ship the simple thing first and let real signal, not my confidence, tell me what to change. Pivot fast when the signal comes, because I built it to be pivoted.

That is why the vet can send twenty-seven changes and have them shipped overnight. Not because we are geniuses. Because the thing was built expecting to change. The builder who left her stuck built for a plan they were sure was right, so

every change fought the structure, and a small change took a month. Same vet, same idea. One build assumed it would be wrong and stayed cheap to fix. The other assumed it was right and turned every fix into surgery.

Plan for the fail is not pessimism. It is the only honest posture when you have watched enough builds to know you will be wrong about something. You just do not know what yet. So you make being wrong survivable, and then you are free to move fast, because moving fast no longer means betting the whole thing on your first guess.

There is a quieter reason this matters, and it is about who you should listen to. The person who has built one product and made it work is often the most confident voice in the room, and often the most dangerous to follow. They will tell you their one way with total certainty, because it worked for them, once. But you cannot separate skill from luck in a single result. The founder of a hundred-million-dollar company might have found the best path, or might have been in the right place at the right moment doing something that would fail nineteen times out of twenty, and now teaches the twentieth as gospel. Neither of you can tell which. That is not a knock on them. It is the maths of small samples. One data point cannot tell you whether the approach was right or merely not-fatal that time.

The only way out of that trap is volume. Watch a build go wrong a hundred different ways and the survivorship bias burns off, because you have seen the same confident approach work here and sink there, and you start to see which parts were skill and which were weather. That is the actual thing experience buys you. Not certainty about the one right way. The opposite. A calibrated sense of how often you will be wrong, and where, so you can build to absorb it. The confident one-timer builds a monument to their one plan. The

person who has done it a hundred times builds something they can change the moment they need to, because they already know change is coming.

So it was never you

Put it all together and the story about yourself falls apart, which is the point.

You were not stuck because you are not technical enough. You were stuck because you were handed a process tuned to complicate, run by people who reinvented the seventy percent that did not need reinventing, aiming at the ninety-nine percent that did not matter, building for a plan they were sure was right, until the small things piled up into a thing that was always almost done. None of that is a verdict on you. All of it is fixable, because all of it is process, and process can be changed.

The developers were not idiots. You were not fooled. The idea was not too hard. The machine was just pointed the wrong way, and nobody in the room could see it because they were standing inside it.

I can see it because I have stood outside more than a hundred builds. That is the whole of what I bring. Not more intelligence than your last developer. Pattern recognition your last developer could not have, because they had not watched it go wrong in enough different ways to know what wrong looks like early.

So the fix was never a better developer or a cleverer language. It was a process, and a way of thinking, held by someone who has watched this go wrong often enough to catch it early. That is the thing I actually sell, and it is the

thing you can learn to spot. What it looks like in practice is the rest of this book.

5. The Math, and What It Actually Takes

> *The number that scares you is almost never the real number. The real number is smaller, and it is spent differently.*

You have an idea and you have a fear, and the fear has a dollar figure attached to it.

You do not know what the figure is. That is the problem. Someone quoted you forty thousand and could not explain what it produced. Someone else quoted twelve and it did not feel right. A friend built something for their business and it cost them a hundred grand and two years and they still talk about it like a war. So the number in your head is not a number. It is a fog. And you cannot make a decision inside a fog, so you do the thing everyone does. You wait. You keep the idea in a folder and you tell yourself you will get to it when things are less busy.

I want to take the fog off the table in this chapter, the same way I took the story about not being technical off the table in the last one. Because the honest economics of building a product are not what the fog tells you they are. They are more knowable than that. They have a shape you can see. And once you can see the shape, the decision stops being a leap and starts being a choice.

So let me show you the actual math. Not the sales-deck math. The real thing, including the parts that are uncomfortable.

What building a product actually costs

Here is the first uncomfortable truth. There is no single price for building software, in the same way there is no single price for building a structure. A carport is not a house is not a hospital. Anyone who gives you a number before they understand what you are trying to do is not quoting the work. They are quoting what they think you will pay. I said that in the last chapter and I will keep saying it, because it is the single most expensive misunderstanding in this whole business.

What I can give you is a range, and the range is real because I have lived at both ends of it.

At one end, a rescue that takes days. Someone comes to me with a product that is stuck. It mostly works. The bones are there. What is wrong is a specific set of things that have piled up and tangled, and the person who built it can no longer point at what is broken. That is not a rebuild. That is a diagnosis and a handful of precise fixes, and it can cost less than a month of the retainer they were already paying the developer who left them stuck.

At the other end, a full enterprise product built from zero. A partner with a rare skillset, productising a career, selling into large organisations that have compliance requirements and security reviews and procurement processes that exist to slow everyone down on purpose. That is a real build with a real team over real time. One of those, in my own track record, came in at around a hundred and fifty thousand dollars and did over three million in its first year. I will come back to that one properly in a later chapter, because the number is not the interesting part of it.

Between those two ends sits almost everything. A first working version of a genuinely new product, enough to put in

front of real users and start learning, is usually a long way south of what you have been quoted. Not because someone is cutting corners. Because most of what you are afraid you have to pay for, you do not have to build at all.

That last sentence is the whole chapter. Let me explain why it is true.

The thirty percent that is yours, and the seventy percent that already exists

Here is the thing almost nobody tells a first-time founder, because the people quoting you have no reason to tell you.

Your product is not new the whole way down. It feels new to you, because the idea is yours and the pain is yours and you have never seen anyone solve it properly. But underneath the idea, most of what a product is made of is the same as what every other product is made of. Logins. User accounts. Storing data and getting it back. Payments. Sending an email when something happens. A screen that shows a list of things and lets you click into one of them. Permissions, so this person can see this and that person cannot.

None of that is your idea. All of it has been built ten million times. And you do not have to build any of it from scratch, because the tools to do it already exist, they are proven, and they are mostly cheap or free.

Roughly thirty percent of your product is genuinely unique. It is the specific thing your expertise makes possible, the part the market cannot fake, the reason someone would choose your product over a spreadsheet. The other seventy percent is plumbing that has already been invented. The job of a good build is to spend almost all of the real effort on your

thirty percent, and to stitch the seventy percent together from things that already work.

This is why the fog quotes you a house when you need a room. Someone building custom-everything is charging you to reinvent the login screen. A person who knows the landscape does not reinvent the login screen. They reach for the thing that already does logins, wire it in, and spend their good hours on the part only you could have thought of.

The ratio is why the work can be fast, and it is why it can be affordable, and the two are the same fact wearing different clothes. Fast and affordable are not a discount. They are what happens when you stop paying people to build what already exists.

The trap on the other side

Now the uncomfortable version, because I am not going to sell you cheap as though cheap were the point.

The mirror image of paying too much for plumbing is paying too little for the thing that actually matters. If your whole product is your thirty percent, and someone tries to do that thirty percent on the same budget you would spend on the plumbing, you get a product that technically runs and does not solve the problem. I have seen this more than I have seen the opposite. People hear “you do not have to build everything from scratch” and they hear “cheap,” and they take the number down and down until the only thing left to cut is the part that was the whole reason to build it.

Spend is not the enemy. Wasted spend is the enemy. The goal is not to build the cheapest possible thing. The goal is to match the spend to the outcome. If you are rescuing something that is nearly there, that is a small spend, and

paying more would be silly. If you are building a product to sell into enterprise, where the whole value is the depth of the thing, spending a hundred and fifty thousand is not extravagant. It is correct. The mistake in both directions is the same mistake. It is spending against a number in your head instead of spending against the outcome you actually need.

I have watched people burn eighty thousand dollars on the seventy percent and have nothing left for the thirty. I have watched people try to buy a nine-figure ambition on a lifestyle-business budget. Both of them got the math wrong in the same way. They decided what it should cost before they decided what it was for.

The six-thousand-page book

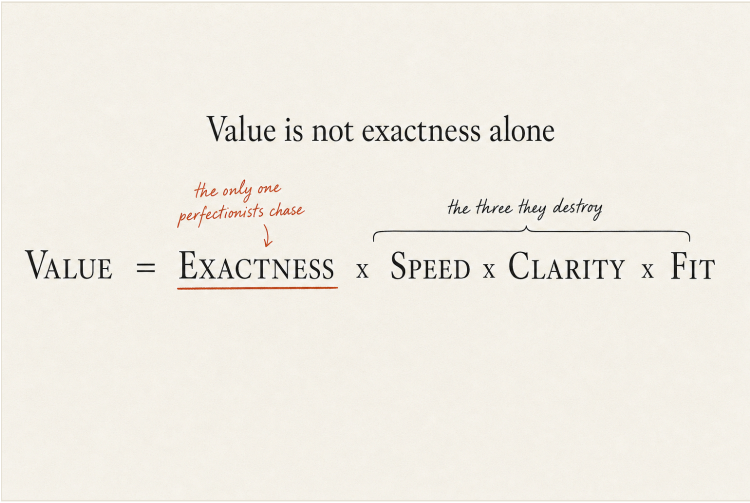
Let me give you the analogy I come back to more than any other, because it explains why more is not better and why the expensive version is often the worse one.

Imagine two books on the same subject. One is two hundred pages. The other is six thousand. The six-thousand-page book is more complete. It is more correct. It contains more true things. By every measure a technical person would reach for, it is the superior book.

Nobody reads it. Nobody can. It sits on the shelf being more correct than the book people actually read, and it changes no one's mind, because a book that does not get read has a value of zero no matter how much truth is packed into it.

The two-hundred-page book is less complete and it is worth more. Not despite being shorter. Because of it. Its value is not in how much it contains. Its value is exactness times speed times clarity times fit. Get to the point, get there fast,

make it clear, make it fit the person holding it. The six-thousand-page version wins on exactness and loses on the other three, and losing three out of four is losing.



Value is exactness times speed times clarity times fit. The six-thousand-page version wins one and loses three.

Software is the same, exactly the same, and this is the trap the perfectionist and the custom-everything crowd fall into every time. They optimise one axis. They chase the most complete, most correct, most feature-complete version, and they destroy speed and clarity and fit to get there. They build the six-thousand-page product. It is more correct than the thing that would have changed your life, and nobody uses it.

The version of your product that ships and gets used and starts making money is the two-hundred-page version. It is not the compromise. It is the target. And I will hold myself to the same standard here, because a book about shipping the

clear, short thing has no business being the six-thousand-page one.

The ninety-nine that never ships, and the ninety-five that changes your life

There is one more piece of math, and it is the most expensive one, and it is the one I am most guilty of myself because I was trained into it.

I trained as a data scientist. The hard maths, the interesting problems, the part of the field that feels like solving a puzzle nobody has solved. So I can tell you exactly how a technical person weighs a decision, because I am one, and I have to fight my own instinct on this every time.

Give a technical person a choice. On one side, a solution that gets to ninety-nine percent, takes two years, and might not work. On the other, a solution that gets to ninety-five percent and ships in two days. They pick the ninety-nine almost every time. Not because they are lazy. The opposite. Because they have been trained to chase the score, and ninety-nine is a better score than ninety-five, and almost nobody stops to ask whether that last four percent is worth two years, or whether the ninety-five was already more than enough to change your life.

It usually was. People wildly underestimate the ninety-five.

The ninety-five percent version ships this week. It goes in front of real users. It starts earning, or it starts teaching you why it is wrong, and either of those is worth more than a perfect thing that arrives in two years, if it arrives at all. The chase for the last few percent is where products go to die. Not

in a clean failure. In a slow, expensive, always-almost-done crawl toward a finish line that keeps moving.

The math of the ninety-five is the math of everything in this chapter. Spend on the thirty percent that is yours. Stitch the seventy that already exists. Ship the ninety-five that works instead of chasing the ninety-nine that does not. Match the spend to the outcome, not to your fear. Do that, and the number that scared you turns out to be a lot smaller and a lot more useful than the fog ever let it be.

So what does it actually take

The money is one half of the answer to "what does it take." Here is the other half, and it is the one that decides whether the money is well spent.

It takes a partner who does the maths I just walked you through, on your behalf, every time, without you having to check. Someone who knows which seventy percent is plumbing and reaches for the proven thing. Someone who can see your thirty percent well enough to spend the real effort there. Someone who ships the ninety-five and resists the ninety-nine, who builds the two-hundred-page version on purpose, and who tells you when the right answer is a days-long rescue rather than a build you do not need.

That is a specific kind of person, and there are not many of them, because the training pushes hard the other way. Everything in this book so far has been about the gap between the business side and the technical side. The person who can do this math for you is the person who has already closed that gap in themselves. They think in outcomes and they build in code, and they will not let you overspend on the plumbing or underspend on the point.

You are not going to become that person by reading a book. You do not need to. You need to be able to recognise one, and to know what the work looks like when it is being done right, so you can tell the real thing from the fog.

You do not have to build like that person. You have to be able to recognise one, hold the maths in your head, and refuse to be talked off it. Get that far and the number that scared you shrinks to something you can actually carry, whether you are starting from nothing or hauling yourself out of a two-year, thirty-grand mess. Both ends of that road come right with the same maths.

6. Stuck at Zero, Stuck at Eighty

> *There are only two places a good idea gets stuck. You are in one of them right now.*

Every person who has come to me with a product has arrived at one of two doors.

Behind the first door is the person who has not started. They have the idea. They have watched the problem up close for years and they know, with the kind of certainty you cannot argue someone out of, that nobody has solved it properly. What they do not have is a first step. They know it is hard. They do not know how to begin. So the idea sits.

Behind the second door is the person who started. Who hired a developer, paid for a design, watched a prototype get to eighty percent, and then just, stop. They are two years and thirty thousand dollars in. The thing is forever almost done. Every fix creates two new problems, and there has never been a single moment they could point at and say, there, that is what is wrong.

Same idea. Opposite ends of the same road. And the thing I want you to understand before we go any further is that both of these people are stuck for the same reason, and it is not the reason either of them thinks.

The four rooms

Let me put you back in the four rooms from the start of this book, because you belong in one of them and it helps to know which.

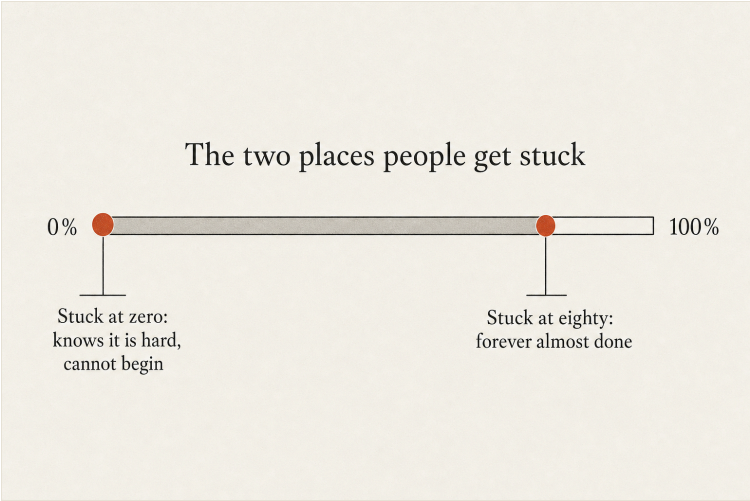
You are in a meeting. The developer is talking. Dependencies, technical debt, something. You stopped understanding thirty seconds in and you have been nodding for four minutes. You do not know if three more months is reasonable or if you are being managed. You walk out with no idea whether your product is on track.

You are staring at your bank account. Eighty thousand dollars. The agency says you are ninety percent done. They said that six weeks ago. Every time something gets fixed, two more things appear. You are starting to think ninety percent done is not a real number. It is the number that keeps you paying. So you approve the next invoice. Again.

You have an idea. A good one. You have validated it, people want it, you know exactly what problem it solves. You have no idea how to build it. You talked to two developers. One quoted forty thousand and could not explain what that would produce. One quoted twelve and something about it did not feel right. You are starting to wonder if the idea will still be relevant by the time you work out how to start.

You built the thing. It is good. You understand exactly how it works and how much better it is than what they had. They are not using it. Four meetings, four rooms full of nodding, then everyone went back to the spreadsheet they have used for six years.

Four founders. The person who cannot start is in the third room. The person stuck at eighty is in the first two. And the fourth room, the one where you built the right thing and nobody uses it, is the one everyone forgets is a form of stuck too. We will get to that one in the chapter on the flip. For now, hold the first three.



Two different kinds of stuck, one root cause. The path was wrong, not you.

Stuck at zero

Start with the person who has not begun, because it is the cleaner problem and the one people are most ashamed of for no reason.

If you are stuck at zero, the story running in the back of your head is that everyone else somehow knew where to start and you did not. That there was a memo about how to begin and you missed it. That people who build things are a different kind of person and you are not one of them.

There was no memo. Nobody knew where to start. The people who look like they knew just started badly and corrected, which from the outside looks like knowing. The reason you cannot see the first step is not that you are missing something everyone else has. It is that the first step is genuinely invisible from where you stand, because the first step is not building anything. It is working out what the

smallest real version of the thing actually is, and that is not a coding problem. It is a thinking problem, and you are already equipped for it.

Here is what actually happens when this goes right. You do not start by hiring a developer and describing your dream. You start by getting the idea out of the fog and into a shape. What is the one thing this has to do to be worth anything at all. Not the ten things you imagine it doing eventually. The one thing. The thirty percent that is genuinely yours, stripped of every feature you added because it seemed like it should be there.

Most people cannot do this alone, and they should not try, because the instinct when you finally get to start is to build everything you have been imagining for two years. That instinct is the thing that turns a person who is stuck at zero into a person who is stuck at eighty. The whole trick of starting well is starting small on purpose, and starting small feels wrong for the exact reason you have waited so long.

The person who has not started does not have a capability problem. They have a first-step problem. And the first step is not technical. It is deciding, with someone who has seen it a hundred times, what the ninety-five percent version is and building only that.

Stuck at eighty

Now the harder one. The person two years and thirty grand deep in a thing that is always almost done.

I want to describe this one carefully, because it is the one people carry the most shame about and the shame is the most misplaced. If you are here, you have probably told yourself some version of these stories. That you picked the wrong

developer. The wrong agency. The wrong stack. That you should have learned to code. That people like you, who know an industry but not software, are not meant to build things.

Take all of that off the table. Here is what actually happened.

You hired someone. Maybe a designer who could think about the product beautifully and then tried to build it and could not. Maybe a developer who took over and went in circles. They were probably not bad at their jobs. What they did not have was the one discipline this whole book is about. They built everything custom instead of reaching for what already exists. They chased the wrong four percent. They optimised the plumbing and neglected the point, or the reverse, and small things piled up on small things until the whole thing was a knot nobody could untangle, least of all you, standing outside it with no way to tell necessary complexity from the kind that is just running up the meter.

That is why there was never a moment you could name what was wrong. There was no single break. There were three hundred tiny ones, tangled together, and a tangle does not have a location. It does not present as a clean failure you could point at and fix. It presents as a thing that is always at eighty, always needing one more thing, never quite arriving. That is the worst place to be. Not because it failed. Because it did not fail cleanly enough to let you walk away.

And here is the part that should give you your evenings back. The tangle is not usually worth saving, and that is good news, not bad. The instinct is to protect the two years and the thirty thousand, to find someone who can fix what is there, because starting again feels like admitting the whole thing was wasted. It was not wasted. The thinking behind it, the design, the understanding of the problem you built up over those two years, that is the valuable part and it is intact. The

code is not the valuable part. The code is the plumbing, and plumbing can be rebuilt from proven parts far faster than it can be untangled.

The most dramatic version of this I have seen went from two years of going in circles to a rebuilt, solid, live product in a matter of weeks, for almost nothing, by keeping the good design thinking and throwing the tangle away. I will tell that one properly in the next chapter. The point here is the shape of it. Stuck at eighty is not fixed by finishing the last twenty percent. The last twenty percent is a lie. It is fixed by keeping what was actually good, which is your understanding of the problem, and rebuilding the rest the right way.

Why both doors lead to the same place

Here is the thing that connects the two.

The person at zero is stuck because they do not know how to start. The person at eighty is stuck because they started wrong. Both of them are missing the same thing, and it is not capability, and it is not money. It is a technical execution partner who does the maths from the last chapter on their behalf. Who knows the thirty percent from the seventy. Who ships the ninety-five and refuses the ninety-nine. Who is honest enough to say start smaller, or honest enough to say throw the tangle away and keep the thinking.

That is why it works from either door. The fix is the same fix. The person at zero gets a first step that is the right size. The person at eighty gets their good thinking rescued from the mess built around it. Different starting points, same partner, same discipline, same result. A thing that ships and gets used.

You do not have to choose the right door. You are already standing at one of them. The only question that matters is whether you keep standing there, or whether you find the person who can walk you through it.

I have said this works from either point. I have not yet shown you. The next chapter is nothing but showing you. Six real ones, across six industries, including the ones that did not work, because the ones that did not work are how you know I will tell you the truth about yours.

7. It Works Across Industries

> *Six real ones. Different industries, same pattern underneath. Including the two that did not work, because those are the ones that tell you I will be honest about yours.*

I could tell you this works. I would rather show you.

What follows is six real cases. A mobile vet. A university operator. An enterprise executive productising a twenty-year career. My own analytics business. My own Power BI tool. And a coaching product I badly wanted to build and could not. Different industries, different people, different sizes of money. The same pattern under all of them.

I have kept all of them deliberately vague on names, and that is on purpose, because the rule I hold myself to is simple. The lesson in each of these is the thing worth sharing, not the label on the company it happened inside. I will tell you a client worked at a serious altitude, or that a product did a certain kind of thing, but the moment I describe the specific work the name comes off, so that no piece of work, mine or anyone else's, is ever tied to a company that did not sign up to be a case study. You lose nothing that matters. A lesson does not need a logo to be true, and the ones that taught me the most are told here with the names left at the door on purpose.

Two of these are wins that look like miracles and are not. Two are principle cases, one of them mine, that show you the trap most people fall into. And two did not work, including one I killed myself. Read all six. The failures are load-bearing.

One word on where I am standing when I tell you these, since chapter two already told the full story. I started in analytics, at Onwards Analytics, and I learned the hard way the thing this whole chapter runs on: in a service business the better you get, the less you make, because the model sells hours and efficiency is a bug you have to hide. I left that behind for products and partnerships, priced on the asset you build rather than the hours you burn. Every case below is me applying that opposite instinct. Ship the simple thing that works. Spend on the part that is genuinely yours. Refuse the marginal chase. Tell the truth about the ones that will not fly.

Start with the first door.

The mobile vet: two years, thirty grand, forever eighty percent

Start with the one that looks like a miracle, because it is the cleanest picture of everything in the last two chapters.

Picture the actual work first. A mobile vet is someone who drives to your house and treats your animal in your kitchen, on your floor, with your cat under your bed instead of shaking in a cardboard box in a waiting room full of dogs. It is a better way to do a hard, stressful thing, and the woman who ran this one had seen exactly where the machinery around that work broke. How it gets booked. How the day gets routed. How the notes and the follow-ups and the reminders fall through the cracks between one house and the next. She had watched that problem up close for years, the way the reader of this book has watched theirs, and she had done the genuinely hard part well. She understood the problem. She had thought about the product properly, how it should feel, what it should do, the shape of the thing, the order the screens should come in, the

moment a stressed pet owner needs a hand held and the moment they just need to book and go. That is the thirty percent that is genuinely hard, the part the market cannot fake, and she had it, fully formed, held in her head for two years.

Then she tried to get it built. That is where it went wrong, and it went wrong in the most ordinary way there is.

The designer who had helped her think it through offered to build it. And you can see why she said yes. He already understood the vision, he was already in the room, he was cheaper than an agency and warmer than a stranger. The trouble is that designers are not developers, and this one could not build a real system, and what came out was what a certain kind of modern tool produces in the hands of someone who does not know what is underneath it. It ran. Sort of. Screens appeared. Buttons did things. From the outside it looked like software. Underneath it was a mess held together with tape, the kind you cannot see from the outside and cannot fix from the inside, code nobody had written and nobody fully understood, generated by asking a machine to make the error go away. A vibe-coded mess. It demoed. It did not hold.

So a second person took over. A developer this time, hired to finish it, a similar profile to the first, and this one went in circles. Fixed a thing, broke a thing. Added the feature she asked for, tangled two others that used to work. Closed one bug and opened two more three screens away. Round and round, month after month, the invoices arriving on time even as the product stood still.

And you have to picture what that does to a person, because the money is the smaller half of the cost. She was running a real business the whole time. Driving between houses, treating animals, holding the hands of owners on the

worst day of their pet's life, and doing all of it on the broken half of a system that was supposed to be finished a year ago. Every booking that should have taken ten seconds took a workaround she had built in her head. Every reminder that should have sent itself she chased by hand. She was paying for software and doing the software's job at the same time, and she could not stop paying, because stopping meant admitting the two years were gone, and the two years were never quite gone, because it was always almost done. That is the tax nobody quotes you. Not the thirty thousand dollars. The eighteen months of running your business with one hand while the other hand holds a thing that is perpetually about to be ready.

Two years. Around thirty thousand dollars. Forever eighty percent.

And here is the detail that matters most, the one I want you to sit with if you are the person from the last chapter standing at the second door. There was never a single point where anyone could name what was wrong. She would ask what the problem was and get an answer about a specific bug, and the bug would get fixed, and the thing would still be at eighty. She would ask again and get a different specific bug, and that one would get fixed too, and it would still be at eighty. Because the problem was not a bug. The problem was three hundred small ones, tangled together, with no location. A knot does not have a place you can point at. It just sits there being a knot, and you keep paying people to pull on individual threads while the whole thing stays tied, and every honest status update is true and useless at the same time. Almost done. Almost done. Almost done.

That is not a clean failure. A clean failure you can grieve and walk away from. This is worse. It is the thing that is

always about to be finished, so you never quite let yourself stop, and the meter never quite stops with you.

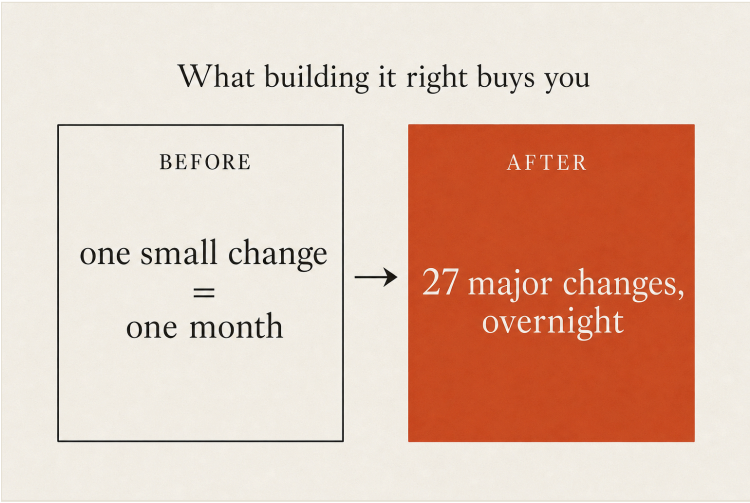
When it came to me, the instinct everyone expects, including hers, was to save the two years. Do not throw away thirty thousand dollars of work. Get someone good in, find the last twenty percent, finish it. That instinct is the trap, and it is worth naming slowly because it is so reasonable. The thirty thousand dollars of code was the plumbing, and the plumbing was tangled, and tangled plumbing is not an asset you finish. It is a cost you keep paying. The valuable part was never the code. The valuable part was her design thinking, her understanding of the problem, the shape of the product she had carried for two years. That was intact. That had never been the thing that was broken. So we kept the only part worth keeping, and we threw the code away without sentiment, and we rebuilt from the ground up on proven parts, the boring reliable parts that thousands of real products already run on.

I want to slow down on that decision, because it is the one that feels most wrong to the person paying, and it is the one that saves them. Throwing away thirty thousand dollars of code sounds like throwing away thirty thousand dollars. It is not. The code was never worth thirty thousand dollars. It cost thirty thousand dollars to produce, which is a different thing, and the two only match on a receipt, never on a balance sheet. What the code was actually worth, to anyone trying to finish or extend it, was less than nothing, because every hour spent understanding the tangle before you could safely touch it was an hour you would never get back, and the tangle got a vote every single time. Keeping it would have meant inheriting the same knot the last two people drowned in. The honest move was to see the sunk cost for what it was, sunk, and refuse to let it drag two more years down after the first two. We kept

the thinking. We binned the plumbing. And once we were building on proven parts instead of a machine-generated maze, the thing that had resisted two developers for two years came together fast, because it was never actually that hard a product. It had only ever been built the hard way.

Live and solid in weeks. For almost nothing, set against the two years behind it.

Here is the before and after in one line, and it is the line I come back to more than any other in this whole book. A small change used to take her a month. Now they send twenty-seven major changes and they are actioned, tested, and released overnight.



A month for one change, down to twenty-seven overnight. That is a system, not a faster developer.

That is not a faster developer working harder. Nobody works that much harder. It is a different category of fast altogether, the difference between a product built as a tangle

and a product built as a system. When the thing is built right, changing it is cheap and safe and fast, forever, because the changes sit on the surface and the system underneath does not move when you touch it. When it is built wrong, every change is a fresh negotiation with the knot, and the knot always wins in the end.

And that gap is not a one-time win. The overnight-instead-of-a-month thing is not the finish line. It is the new baseline she lives on now, forever. A competitor does something, she responds this week, not next quarter. A customer asks for a thing that would help, she tries it tonight and sees by morning whether it was worth keeping. The product stops being a thing she is afraid to touch and becomes a thing she steers. The tangle taxed her every month. The system pays her back every month. Same product, opposite sign.

The lesson she carries is the two teaching devices from the earlier chapters, wearing real clothes. It was a process problem, not a technology problem. And most of the product was proven plumbing she never needed to reinvent, only stitch together well. Her developers were not idiots. I want that said plainly, because if you are the vet in this story you have probably spent two years privately wondering if you hired badly, or worse, if you are the problem. You are not. They just never had the discipline to keep it simple and reach for what already existed instead of building everything custom. Almost nobody has that discipline until someone shows them, and by the time someone does, the meter has usually been running for two years.

If you are standing at that second door right now, stuck at eighty, paying someone to pull on threads, take one thing from the vet before we move on. The fact that nobody can name what is wrong is not a sign that it is nearly fixed. It is

the diagnosis. You are not twenty percent from done. You are looking at a tangle that has no last twenty percent, only more thread. The two years are already gone and the only question left is whether you spend a third the same way. The vet did not. She kept the one thing that was hers, the thinking, and let a stranger throw away the rest without flinching, and she got her product and her speed and her business back in weeks. The sunk cost felt like the thing to protect. It was the thing to release.

The university operator and the trap of the last one percent

The second case is a principle case, and it is the one that names the trap most explicitly, because the person in it lived through the trap before he ever came to me.

He was the chief operating officer of a university-facing consulting and technology business. A serious operator, running real numbers, selling into institutions that move slowly and buy carefully. The business failed. And the honest read on why it failed is that it fell into exactly the trap I am about to describe, which is what makes him the right person to tell it, because he did not read about this trap in a book. He watched it eat his company from the inside, and he lived to name it.

You should understand what that failure looks like before it happens, because it never looks like failure while it is underway. It looks like diligence. It looks like a team doing careful, serious, defensible work, hitting technical milestones, passing internal reviews, building something more capable every month. Nobody in a company like that is slacking. They are the opposite of slacking, which is the tragedy of it. They

are working hard on the wrong axis, and hard work on the wrong axis feels exactly like hard work on the right one from the inside, right up until the runway ends. By the time the truth arrives, it does not arrive as a lesson. It arrives as a payroll you cannot meet. He came out the other side of that knowing something most operators never get taught, because most operators only see this trap once and do not survive it clearly enough to name what killed them.

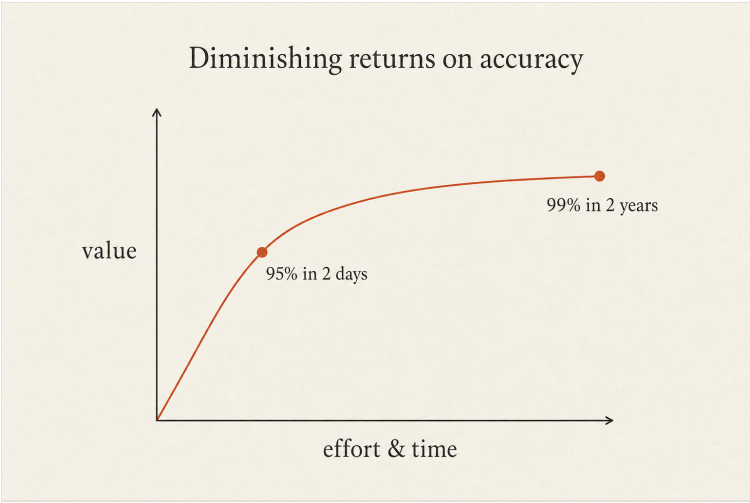
The work in front of us was a scoring product. A way of measuring something about how an institution was performing, some tangle of inputs about how a place was doing its job, and turning it into a number a busy person could act on. A score. And the moment you say the word scoring to a certain kind of technical person, you have started a race that has no finish line, and most of the runners do not know there is no finish line until they are two years in and out of money.

I trained as a data scientist, so let me tell you exactly how a data scientist thinks about a scoring problem, because it is the whole lesson and I have lived inside that head. Give a data scientist a scoring model and they will chase accuracy. It is the most natural thing in the world to them. Ninety percent is not good enough when ninety-five is clearly possible. Ninety-five is not good enough when ninety-nine is sitting right there on the table, taunting you. So they will spend two years building a custom model that gets to ninety-nine percent accurate, deeply optimised, academically defensible, the kind of work that would survive a room full of peers picking at it. Genuinely impressive as a piece of engineering.

And it will be commercially useless.

Because here is what the score-chaser never stops to ask, not once, in the whole two years. The version that gets to ninety-five percent could have been built in two days. Two

days, not two years. And the last four percent, the four percent that cost the two years, does not change a single decision anyone makes with the number. Nobody acts differently on a ninety-nine than they would have on a ninety-five. The dean does not run the institution differently. The board does not vote differently. The person holding the score cares about one thing, which is whether they can trust it enough to move, this quarter, cheaply enough that it was worth having at all. The fourth decimal place is a private pleasure of the person who built it. It is invisible to the person who paid for it.



The last four percent costs the two years, and changes no decision anyone makes with the number.

Let me put a shape on those two numbers so the gap stops sounding abstract. The two-day version is not a rough draft you would be embarrassed by. It is a sensible model built out of the parts that already exist, the standard, proven, well-

understood parts that ninety-five percent of the answer lives in, because ninety-five percent of most problems has already been solved by somebody and is sitting there waiting to be used. You wire those parts together, you sanity-check the output against reality, and you have a number a busy person can trust and act on by the end of the week. The two-year version chases the last five percent, the part that is genuinely novel, the part nobody has solved, and it chases it because that part is interesting and the rest is boring. But the reason nobody has solved that last five percent is usually that it is enormously hard and barely matters, in that order. So the team spends the ninety-five percent of its time and money on the five percent of the answer that changes nothing, and calls it rigour, and runs out of road. The maths of where the effort goes is the exact inverse of the maths of where the value is. That inversion is the whole disease.

Data scientists think in scores. Business runs on decisions. The gap between those two is where two years and a company disappear, and it disappears so quietly that everyone inside is proud of the work right up to the moment the money runs out.

This is the trap the whole industry winds itself into, over and over, in every vertical. Chase the marginal accuracy. Build custom where proven would have done. Optimise the one axis a technical mind can measure and destroy the three it cannot, speed and clarity and fit, until the product is late and expensive and too complicated to use, more correct than the thing that would have worked and used by no one. It is the six-thousand-page book, built out of maths. More content, more rigour, more pages, less value, because value was never volume. Value was the thing that got there in time to matter.

I can call this trap out with a straight face for exactly one reason. I am a data scientist. I love the ninety-nine percent

problem. It is the interesting problem, the puzzle, the part of the field that feels like solving something nobody has solved before, the part that got me into this in the first place. And I have had to train myself, over more than a hundred projects, to walk past it. To ship the ninety-five that changes someone's life this week instead of chasing the ninety-nine that might arrive in two years and might not work at all. That is not a natural instinct for me. It is a learned one, learned against the grain, and I learned it by watching people who never learned it lose everything to the last four percent while telling themselves they were nearly there.

The university operator's product is honestly early. I am not going to dress it up with a revenue number it has not earned, because that would make me the thing this whole book is against. Its job here is not to be a trophy. Its job is to name the trap, from the mouth of a man whose company died in it, verified by a man who was trained to fall for it and chose, project after project, not to. People wildly underestimate the ninety-five. He knows it now. It cost him a company to learn.

And there is a reason I want this case standing second in the chapter, right after the miracle rescue, rather than tucked away. The mobile vet is the happy version, the one you want to be, the stuck build brought back to life. This one is the warning underneath it. The vet got out because someone made the ruthless call to bin the tangle. The university business did not get that call in time, and it is not because the people in it were less capable. It is because the trap wears the costume of good work. Nobody in a failing over-engineered company feels like they are failing. They feel diligent. They feel like the responsible adults refusing to ship something half-baked, holding the line on quality while lesser teams cut corners. That feeling is the trap talking. The responsible thing was almost never the ninety-nine. It was the ninety-five,

shipped, earning, teaching you what to build next from real customers instead of from your own imagination. He would tell you that himself now, which is exactly why his product, early as it is, earns its place in a chapter otherwise full of numbers.

The enterprise product: over three million in year one, built for a hundred and fifty grand

The third case is the biggest, and it is the one I have to tell you the most carefully, because the person at the centre of it does not want it named, and that is the right call, and I am going to honour it completely. So here is what I can say, and I have said it to enough people now to know it lands without a name attached.

A twenty-year enterprise expert. Someone with a rare and specific skillset built over two decades inside large organisations, the kind of knowledge you cannot hire and cannot fake and cannot get from a course, because it only comes from twenty years of being in the room where the hard version of the problem plays out. He decided to take that career and turn it into software. Not a lifestyle business. Not a small tool to sell on the side. A serious product with a nine-figure ambition, sold into big enterprises, the kind of thing that either becomes an industry standard or does not exist.

Enterprise is a different sport. I need you to understand that before the numbers, because the numbers do not make sense without it, and I do not want you reading this and thinking your idea will do the same thing on the same budget, because it might not, and honesty is the whole point of this chapter.

Everything I have told you so far about shipping simple and fast still holds here. It just holds inside a much harder set of walls. The product is sold centrally, to an organisation, and the people who actually use it every day did not choose it. They were told to use it. That one fact changes everything about how you have to design it, because a mandated user who finds the thing confusing does not go and find a better tool the way a consumer would. They cannot. So they resent it instead, quietly, at their desk, and the resentment spreads, and the rollout curdles from the inside while the numbers on the contract still look fine.

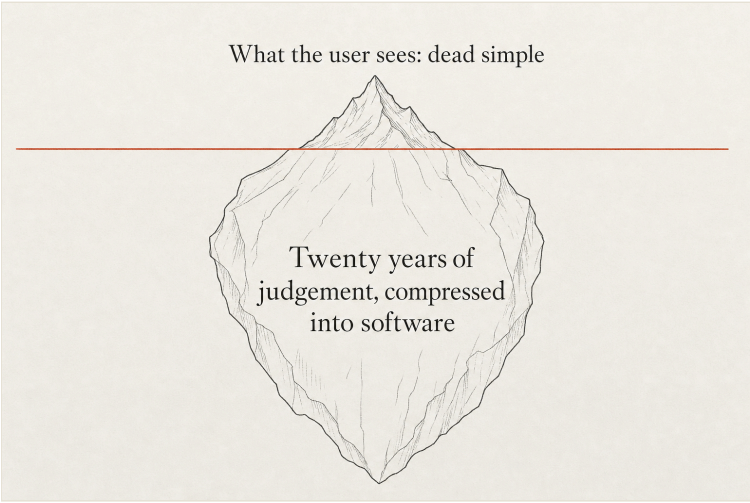
Picture the mandated user, because designing for them is a different job than designing for a customer. A customer wanted your product. They arrived with goodwill, ready to be pleased, willing to forgive a rough edge because they chose to be there. The mandated user arrived with none of that. Someone above them signed a contract and now there is a new system they have to use to do the job they already knew how to do, and their first feeling toward it is suspicion, maybe irritation, definitely no patience. If the first screen confuses them, they do not lean in and figure it out. They decide the thing is rubbish, and they are right to, from where they sit, and they tell the person at the next desk it is rubbish, and now you have two people deciding it before they have really tried. That is how a rollout dies, not in a boardroom but in a hundred small resentments at a hundred desks. So the interface has to be simpler than a consumer product, not more complex, even though the thing underneath is doing something far heavier, because the user has no goodwill to spend and no exit to threaten. Every gram of confusion you leave in gets paid back with interest by someone who did not want to be there.

On top of that sit the enterprise hoops, and there are many, and most of them exist to slow you down on purpose. Security reviews. Compliance tracks. Procurement processes with their own committees and their own quarters. Scope that shifts under your feet because the buyer is not a person, it is a committee, and committees change their minds and their sponsors and their priorities halfway through.

And underneath all of that, the real work, the actual thirty percent, was the hardest kind I have ever had to help build. It was compressing twenty years of one person's rare expertise into clean, simple software that a stranger could use correctly without ever being told how. That is not a feature you add. That is the entire product, and it is brutally hard, and it was the one thing that only that specific partner could ever have brought to the table. I could build the software. I could clear the hoops. I could not have supplied the twenty years. Nobody could, except him.

It helps to be concrete about what that compression looks like, because it is easy to nod at the phrase and miss the work. The expert carries a judgement in his head that he makes in a second, without noticing he is making it, because he has made it ten thousand times. He looks at a situation and knows what matters, what to ignore, what order to do things in, where the risk is hiding. To a stranger, that same situation is a wall of choices with no signposts. The job of the product is to take the judgement that lives silently in his head and bake it into the software so completely that the stranger makes the right call without ever realising a call was being made. The screen has to already know what he would have said. That is not writing down what he knows. Writing it down produces a manual, and nobody reads the manual. It is engineering his instinct into the shape of the thing, so the easy path through the software is also the correct one. Get that right and the product

feels obvious, almost too simple to be worth what it costs, which is exactly the tell that it was built well. All the difficulty went underground where the user never has to meet it. That underground work was his career, translated. It was the only irreplaceable thing in the whole build, and it was the only thing I could not have done without him.

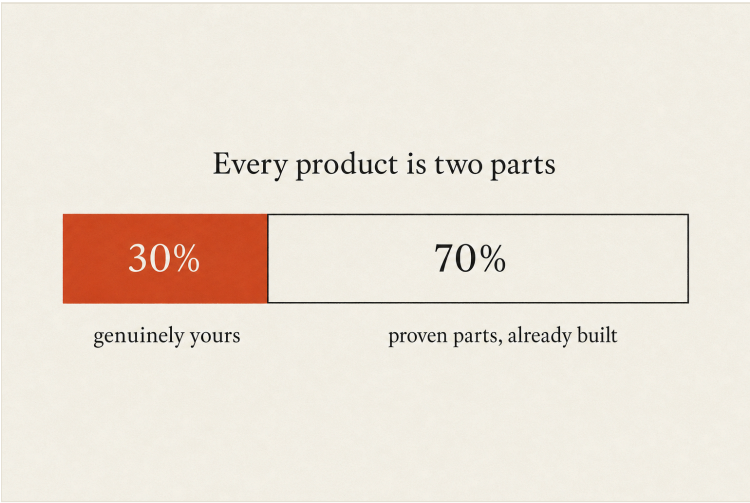


Get it right and the product feels almost too simple to be worth what it costs. That is the tell that it was built well.

We built it from zero. Full team. Multiple enterprise sales closed while the thing was still being shaped, which tells you the market was real before the product was finished. Consulting support running alongside the build so the early buyers were held properly. The compliance track running in parallel the whole way, with the security certifications that large buyers demand coming into view on schedule rather than as a panic at the end.

It did over three million dollars in its first year. It was built for around a hundred and fifty thousand.

Sit with the gap between those two numbers, because it is the whole argument of this book compressed into a ratio. The conventional way to build that product, the custom-everything, full-agency, three-to-four-year way, would have cost two to three million dollars and a team to match, and it would have arrived late enough that the market might have moved on. We did it for a hundred and fifty thousand and it earned twenty times that in twelve months. Not by cutting corners. By spending the money on the thirty percent that was the actual product, the compression of that career into software, and stitching the seventy percent of plumbing together from proven parts instead of paying to reinvent things that already exist and already work.



Spend on the thirty percent only you can bring. Stitch the seventy together from parts that already exist.

And notice the number I did not shrink. A hundred and fifty thousand dollars is real money. I did not do this on a shoestring and I would not have wanted to. This is the point from the maths chapter about matching spend to the outcome, and it is the point people miss when they hear the rest of the book and think the message is spend as little as possible. It is not. A nine-figure enterprise ambition warrants a hundred-and-fifty-thousand-dollar build. That is not extravagance. That is correct. The founder who tries to buy that outcome on a ten-thousand-dollar budget fails just as certainly as the one who spends eighty thousand on a login screen. The discipline was never spending little. The discipline is spending the right amount, on the right thirty percent, and refusing to waste a single dollar on the seventy that already exists.

That is what running real budget looks like when the goal demands it. The weeks-long vet rescue and the hundred-and-fifty-thousand-dollar enterprise build are the same person applying the same maths at two very different scales. Spend matched to the goal. Nothing wasted at either end. If your thing is a mobile vet, we will not spend enterprise money on it. If your thing is a nine-figure enterprise product, I will not pretend it can be built for the price of a website. The number follows the goal, and the goal is yours to bring.

And underspending kills just as many good ideas as overspending, which is the point the maths chapter and the fifty-thirty-ten chapter make in full: the miser who starves the thirty percent loses just as certainly as the perfectionist who gilds the seventy. Ten thousand dollars aimed at the compression of a twenty-year career would have been the most expensive saving that founder ever made.

My own tool: the flip, and why cool tech is worth nothing on its own

The fourth case is mine, and it is a failure, and I am telling it on purpose because it taught me the single most important thing about who this whole model serves. It taught it to me by embarrassing me, which is the way I tend to learn things properly.

Onwards runs support for a lot of Power BI work. Reviewing files, tracking what changed between one version and the next, keeping documentation current as reports evolve, all the unglamorous, repetitive plumbing of a serious analytics operation. It is the kind of work that eats hours and never shows up in a highlight reel. So we built a tool to automate it. And internally, it was genuinely good. That is not me being kind to myself. It did something real, it saved real time on real work, and the technology under it was clever and, honestly, fairly unique. We used it. It earned its keep the day we shipped it to ourselves.

So we did the obvious thing, the thing anyone in my position does. We looked at a tool that was plainly good and thought, other people have this problem too, let us sell it. We productised it, put a price on it, built a proper landing page, and ran ads to sell it to the world.

Crickets.

Not a slow start. Not a rough launch we could iterate our way out of over a few months. Something much closer to silence. You put a thing into the market you know is good, that you have used yourself, that saved you real hours, and you sit there watching the numbers, waiting for the world to agree with you. And the world just does not answer. The ads ran. The clicks came, some of them. And almost nobody stayed, and almost nobody paid, and the ones who did needed

so much hand-holding to get value that it was clear this was never going to run on its own. That is a strange, quiet kind of failure, because there is no single thing to point at and fix. It is not broken. It works. People just do not want it enough to move toward it, and no amount of shipping features closes that gap, because the gap was never in the product.

And the reflex, when that happens, is to assume the tech was not good enough, or the ads were not sharp enough, or the price was wrong. I checked all of those. I am reasonably good at all of those. None of them was the answer. The tech was good, and being good was not enough, and that gap is the whole lesson.

Here is why it went quiet, and I have had a long time now to be honest with myself about it, which is harder than being honest about somebody else. The market was smaller than it looked from the inside, and the thing was hands-on to make work for any given customer, so it did not sell itself the way a clean self-serve product does. Those things were true. But underneath them sat the real reason, and it took me longest to say out loud. There was no dedicated partner who owned the pain and burned to solve it. There was no Power BI-obsessed person whose entire working life this tool was, driving it into their world, evangelising it in the places those people gather, shaping it week after week around a pain they felt in their own bones. And crucially, there was no drive from me to be that person, because it was not my pain. I built it because it was a clever solution to an annoyance. I did not build it because I could not sleep until it existed. Nobody involved could not sleep until it existed. So it existed, and then it sat there.

That is the flip, and it is the most important sentence in this chapter. No one cares how cool your tech is if it does not make a clear pain point better for someone who feels that pain in their bones. The tech is not the hero of the story. The

tech is never the hero. The expert who owns the pain is the hero, and without that person driving it, the cleverest tool in the world sits in a folder making no sound at all.

Read that against everything else in this book, because it inverts the exact thing you have been afraid of. You have been carrying a quiet story that you are the weak link in this. That you are the non-technical one, the person who does not understand the code, the one who has to be carried by whoever builds the thing, the passenger. That tool is the proof that the opposite is true. I had the tech. I had the team. I had the skill to build anything I could specify. And it went nowhere, because the one thing it did not have was you. The domain expert who owns the pain and drives it into the world. You are not the passenger in this. You are the engine. The build is the part I can hire and hire well. The obsession is the part that cannot be hired or faked or manufactured, and it is the part you already have, sitting in you right now, which is the whole reason you are reading this book.

Think about what a driver actually does that no builder can. A driver knows, without a meeting, which of a hundred possible features is the one that matters this month, because they feel the pain that feature answers. A driver knows where the buyers gather, because they are one of the buyers, and they can walk into that room and be believed in a way a vendor never can. A driver hears a customer complaint and knows in a second whether it is a real problem or a one-off, because they have lived the problem for years. A driver keeps going after the first launch is quiet, because it is their thing and they cannot let it sit, where a builder shrugs and moves to the next contract. Every one of those is a decision the product needs made correctly, over and over, for years, and every one of them comes from owning the pain, not from knowing the code. I had all the code and none of that, and the product died

of it. You have all of that, and you have been told it does not count. It is the only thing that counts. The code was always for sale. The driver never was.

Which brings me to the reason I can offer to build your product and take a minority stake in it, and why that is not a trick hidden in the fine print. A hundred percent of nothing is worth a lot less than a minority stake in something that actually makes money. I own a hundred percent of that tool. It makes almost nothing, because it never had its driver. I would trade all of it, gladly, for a minority share of a real business with a real expert burning to run it, every single time, because a minority of something is worth infinitely more than all of nothing. That is not generosity dressed up as a deal. It is just the maths, and it happens to make the partnership honest, because it only pays me if your thing actually works. My upside is your success. There is no version of this where I win and you do not. That one tool of mine is the reason I know that in my body, not just on a slide.

Sit inside that trade for a second, because it is the part people are most suspicious of, and the suspicion is fair, and the answer holds up anyway. The question people ask, half under their breath, is why would a builder who can build anything hand the majority of the thing to the person who cannot build it at all. It sounds too good, and things that sound too good usually are. But run the numbers the way I have had to run them since that tool went quiet. I can build a hundred products a year and own all of every one of them, and if none of them has a driver who owns the pain, I own a hundred percent of a hundred silent folders. That is what I actually own right now with the one I built for myself, scaled up. Or I can take a minority of a small number of things that have their driver, the person who cannot sleep until it exists, and watch those things actually move in the market, and a

minority of a thing that moves is worth more than the whole of a thing that sits. The scarce ingredient was never the build. I have proven to myself, expensively, that I have plenty of build. The scarce ingredient is the driver, which is you, and the person holding the scarce ingredient should hold the majority. That is not charity. That is just pricing the two things correctly. It only looks generous if you still believe the tech is the valuable half, and that silent tool of mine is the receipt that proves it is not.

The one I killed

The last case is the one I am proudest of telling, because it is the one that did not work and I stopped it myself, and I want you to see it before you ever consider handing me your idea.

It was a coaching product, and it was exactly the kind of deal I badly wanted. A motivated coach with deep, specific domain knowledge, the sort of person this whole model is built for. Interesting technology, the sort I love most, pose analysis driven by AI. The idea was clean and genuinely exciting, and I want you to feel how good it sounded, because that is the point. Take a coach's hard-won expertise and a swimmer's video, run the footage through pose estimation, automatically analyse the athlete's technique frame by frame, and turn a slow, manual, expensive coaching process into something fast and scalable. A great coach could suddenly help ten times the swimmers. I wanted it to work. I did not hold back. I leaned all the way in.

And I want to be honest about the pull of it, because the pull is the thing that gets good people into bad builds. Everything about it was the kind of thing that makes you want to say yes. The coach was exactly the driver my own tool never had, the person who owned the pain and could not stop

thinking about it, who lived in the world the product was for. The technology was the flavour I am most tempted by, novel and elegant and the sort of problem that makes a data scientist's hands itch. The story told beautifully in a sentence. Slow, expensive coaching turned fast and scalable, one great coach reaching ten times the athletes. If you had pitched it to me as a stranger, I would have leaned in exactly as hard as I did. That is the danger. The deals most worth being disciplined about are the ones you want most, because wanting it is precisely what makes you skip the hard question until it is too expensive to ask.

It did not work, and here is exactly why, in detail, because the why is where the honesty lives and vague failures teach nobody anything.

Two things killed it, and they did not just add up, they compounded, each one making the other worse. The first is that sport is hyper-specific. Elite technique is not made of big, obvious movements that a camera catches easily. It is made of tiny ones. A few degrees of rotation in the shoulder. A fractional change in the timing of the catch. The difference between a good entry and a great one measured in centimetres and milliseconds, invisible to anyone but the coach who has spent a career learning to see it. The entire value of the coaching lives in changes that small. Which meant the tool had to be accurate enough to see changes that small, reliably, every time, or it was not a lesser version of the product. It was worthless, because a swimming tool that cannot see the thing swimming is about is not a swimming tool.

The second thing is worse, and it is the specific detail that made me stop rather than push. The most important moments in the stroke happen on the fringes. The exact instant the hand breaks the surface of the water. The catch, at

the front of the stroke, half in and half out. The parts of the movement that are half-submerged, blurred by spray, obscured by the swimmer's own body, moving fast, at precisely the angles and in precisely the conditions a camera handles worst. And those fringe moments, where the footage is least reliable, are exactly where the technique problems live and exactly where the accuracy had to be highest. The tool needed superhuman precision in the one place precision was impossible. Not merely hard. Impossible, with what the footage could physically give us. The place we most needed to see was underwater, blurred, and gone in a hundredth of a second.

Notice how that inverts the good news of every other case in this chapter. Everywhere else, the lesson is that ninety-five percent is enough, that people wildly underestimate the ninety-five, that the last few percent is a vanity chase you should walk past. This is the one place that flips. This one needed the last few percent, because here the last few percent was the entire product. The whole value was in seeing the thing nobody else could see, at the fringe, underwater, in a hundredth of a second, and a version that got the easy ninety-five percent of the stroke right and missed the hard five was not a useful ninety-five. It was a confident lie about the only part that mattered. This is the discipline in both directions. You have to know when ninety-five is plenty, which is almost always, and you have to know the rare time when it is not, because the value has stacked itself entirely into the part you cannot reach. Reading that difference correctly is most of the job. Getting it wrong in either direction burns years.

So we could not get close enough to real. Not ninety-five percent close, which would have been fine, which would have shipped. Not even close enough to be useful at all. And when I understood that, when I could see it was a wall and not a hill,

I did the thing I am asking you to trust me to do with your product. I paused it. I did not push through and ship an eighty-percent tool that gave coaches confident, wrong feedback about the exact moments that matter most, which is worse than no tool, because a wrong tool that sounds sure of itself does damage a missing tool never could. A coach who trusts a bad reading changes a stroke that was fine and breaks it. The athlete gets slower and nobody knows why. I did not take the coach's money for a mess and call it a first version and promise the accuracy would come later. I said, not yet, this one does not work with what we have, and I stopped.

That is the capstone, and it is the reason all five cases before it should carry weight. I will volunteer a failure I was invested in and wanted badly and had already leaned into, and I will tell you exactly why it failed, right down to the moment the hand breaks the surface of the water, because a partner who only ever tells you it will work is not a partner. He is a salesperson with a nicer vocabulary. The person you actually want building your product is the one who will tell you the truth, including the truth that is not yet, while it is still cheap to hear. If it had worked, I would be telling you it worked. It did not, so I am telling you it did not, and that is the same instinct pointed straight at your idea. If yours is one of those, an accuracy wall dressed up as a hill, you will hear it from me early, before you have sunk two years and thirty grand into it. If yours is a mobile vet, you will hear that too, and we will go and build it.

What the six of them tell you

Line them up. The vet, rescued in weeks by keeping the thinking and binning the tangle: a process problem, not a tech one. The university operator, the trap named from inside it,

because business runs on decisions, not scores. The enterprise product, over three million in year one for a hundred and fifty thousand, because the money went to the thirty percent that mattered. My own tool, the flip: cool tech with no driver is worth nothing, which means you are the part that cannot be replaced. The one I killed, the honest kill, because a partner worth having tells you before it costs you anything. And under all five, the analytics business that taught me the whole thing: fast and right is punished when you sell hours and rewarded when you build assets.

Six industries, one pattern. Ship the simple thing that works, spend on the part that is genuinely yours, refuse the marginal chase, and be honest about the ones that will not fly. It is not a trick that happens to work in one lucky vertical. The pattern is structural, not industrial. It does not care what your industry is. It only cares whether the thirty percent that is genuinely yours is real, and whether you are the one who cannot sleep until it exists.

You have seen the maths, from both doors, at both ends of the money, and you have seen me kill a product I loved rather than sell you a dream. What is left is the method. The actual how, the way of building that produces all of this on purpose instead of by luck. That is the rest of the book, and it starts now.

8. The Method: Build for the Pivot

> *You will be wrong about something. The only question is whether you built so you can move when you find out.*

I want to start this part of the book with a confession, because the method that follows only makes sense if you understand the thing it is built to survive.

Every plan I have ever made for a build has been wrong somewhere. Every one. Not catastrophically wrong, usually. But wrong in a spot I could not have predicted at the start, in a way that would have quietly wrecked the project if I had committed everything to the original picture. A feature the user did not really want. A tool that looked perfect in the demo and fell apart in real use. A model provider that shipped, three weeks in, the exact capability I had spent a month building by hand.

That last one is real. In one of my own products I spent months building a back-and-forth reasoning loop for an AI feature. Wired it up, tested it, got it working. And then the company whose model I was using released the same capability natively, out of the box, and it was a hundred times better than mine. Months of careful work, made pointless overnight by a thing I had no way of seeing coming.

Here is the part that matters. It did not hurt. Because I had built it so that piece could be pulled out and replaced without touching anything around it. I swapped their version in, deleted mine, and the product got better in an afternoon. If

I had welded my reasoning loop into the guts of the thing, that same event would have been a rebuild. Same surprise. Completely different cost. The difference was not that I saw it coming. I did not. The difference was that I had assumed something like it would.

That is the whole method in one story. You are going to be wrong. Build so that being wrong is cheap.

Why a fixed blueprint is the trap, not the answer

Most books about building software want to sell you a blueprint. Do these steps in this order and you get this result. It is a comforting promise and I understand why people buy it. A blueprint feels like safety. It feels like someone has removed the risk for you.

The problem is that a blueprint assumes the plan is right. It assumes the thing you decided on day one is the thing you should still be building on day forty. And in my experience that is almost never true, because you do not know what you are building until you have built a piece of it and watched a real person use it. The map is drawn before you have seen the territory. Then the territory turns out to be different, and the map, which felt like your protection, becomes the thing dragging you toward a cliff.

I have watched this happen from the inside more times than I can count. A team commits to a detailed plan. The plan turns out to have a wrong assumption near the middle of it. And instead of changing the plan, the team defends it, because so much has already been built on top of the assumption that pulling it out feels like admitting the whole thing was a waste. So they build around the wrong thing. They

add scope to compensate for it. And the product gets heavier and slower and further from what it needed to be, all in the name of honouring a plan that was written before anyone knew anything.

The fixed blueprint does not remove risk. It concentrates it. It takes all the uncertainty in a project and bets it on the accuracy of a document written at the moment of maximum ignorance, which is the start.

The one-timer and the survivor

There is a specific kind of advice you get about building products, and it usually comes from someone who has done it once. They built a thing, it worked, and now they know the way. They are certain about it. This exact stack, this exact process, this exact order. It worked for them, so it is the truth.

I want to be careful here because I am not saying that person is stupid or dishonest. They are neither. They are describing, accurately, the one path they walked. The problem is that a single success tells you almost nothing about whether the path was good. It tells you the path was survivable that one time.

Think about it the way you would think about any other sample size of one. A company that grew to a hundred million dollars did a thousand things on the way. Some of those things helped. Some of them were actively harmful and the company succeeded despite them, carried by one or two things that worked and a fair amount of luck. But the founder, looking back, cannot tell which was which. So they teach all of it as gospel. The good calls and the lucky escapes get bundled together and sold as the method. That is survivorship bias, and it is most of what passes for building advice.

I have built and rebuilt hundreds of things. Funded startups, enterprise applications, internal tools, small rescues, products of my own that worked and products of my own that did not. That number is not a brag. It is the only thing that lets me tell the difference between a rule that holds and a rule that happened to hold once. When you have watched the same decision play out across hundreds of different contexts, you stop seeing the specific stacks and processes and start seeing the pattern underneath them. And the pattern is not a particular way of building. The pattern is a particular way of being wrong well.

The one-timer is set in stone because they have one data point and it looks like certainty. The person who has done it hundreds of times is loose, flexible, quick to change direction, because they have seen the plan get overturned enough times to stop trusting any single plan. It looks like the experienced person knows less. They are more willing to say I do not know yet, let us find out. That is not less knowledge. That is the actual shape of the knowledge.

What building for the pivot means

So if the method is not a blueprint, what is it. Let me make it concrete, because build so you can move is easy to say and vague enough to be useless if I leave it there.

Building for the pivot means three things in practice. Getting the idea right, scoping the simple version, and then building it well. The chapters that follow take each of them in turn.

It means getting the idea right before you build anything, which sounds obvious and is the step people skip hardest. Getting the idea right does not mean writing a longer

specification. It means finding the single thing the product has to do, the one job that if it fails nothing else matters, and being ruthless about separating that from everything that merely came along for the ride. Most builds go wrong here, at the very start, because the expert has a rich and complicated picture in their head and tries to build all of it at once instead of finding the core of it first. That is the first.

It means scoping the simple version, the ninety-five percent that ships in days rather than the ninety-nine percent that ships in years, and being honest about which lane you are in while you do it. There is a version of almost every product that is small, sharp, and shippable now, and a version that is complete, careful, and two years away. The simple version is not the compromise. In most cases it is the better product, and I will show you why. That is the second.

And it means building it by stitching proven tools together for the parts that are not special, and spending your real effort only on the part that is, so that the thing comes together fast and stays flexible enough to change when you learn you were wrong. Most of any product is not unique. Roughly a third of it is genuinely yours and the rest is plumbing that every product needs and that has already been solved by tools you can buy off the shelf. The teams that stay stuck at eighty percent are almost always the teams that tried to build the plumbing themselves. That is the third, and the chapters on the method itself show you exactly how it is done.

Get the idea right. Scope the simple version. Build it on proven tools with the effort saved for the part that matters. Three phases, and running underneath all three, the same assumption. You will be wrong about something. Keep the stack flexible so that when you find out, you move instead of rebuild.

Flexible frameworks, simple things, fast pivots

Let me hold those three ideas up next to each other, because they sound like they might contradict, and the fact that they do not is the heart of it.

The first idea is that you build flexible frameworks rather than fixed features. I covered a version of this earlier in the book, in the operations tool that got rebuilt in a day once we understood it needed to change every few weeks. Instead of building ten survey questions as ten separate things, each with its own logic and its own maintenance cost, you build one system where the questions sit on top as a surface layer and the underlying logic is written once. Adding a question becomes a small change on the surface instead of a new build. The framework is designed to absorb change rather than resist it.

The second idea is that you ship the simple thing. You do the least you can get away with, precisely defined, and you get it in front of a real person as fast as possible, because the real person is the only reliable source of truth about whether you got the idea right.

The third idea is that you pivot fast on real signal. When the real person tells you, through their behaviour rather than their words, that you were wrong about something, you change direction quickly, without ego, without defending the plan.

Here is why those three do not fight each other. The flexible framework is what makes the fast pivot cheap. The simple thing is what gets you to the real signal quickly. And the fast pivot is what turns the signal into a better product before you have spent everything. They are one loop. Build something small and flexible, put it in front of someone real,

watch where it breaks, change the surface layer, put it back. The reason this beats the blueprint is not that it is more disciplined. It is that it treats being wrong as information to be gathered early and cheaply, rather than a failure to be avoided and then denied.

The perfectionist hates this loop because it means shipping something they know is incomplete. But incomplete and in front of a real user beats complete and still in your head, every time, because the real user knows things about your product that you cannot know until they touch it. You are not lowering your standard. You are moving the moment of truth forward, to when it is still cheap to act on.

The cost of building for correctness instead

I want to show you the other path, because you have probably been on it, and naming it is how you stop.

The other path is building for correctness. It looks responsible. You plan thoroughly. You build each piece properly and completely before moving on. You do not ship until it is right. Every individual decision along the way is defensible, which is exactly what makes the path so hard to escape, because at no single point are you doing anything visibly wrong.

And then, somehow, you are at eighty percent and you cannot get off it. Not because of one big failure. Because of a hundred small things that piled up, each one built properly in isolation, none of them tested against a real user until it was too late to change cheaply. The custom pieces you built instead of buying are now load-bearing and cannot be swapped. The assumptions from the original plan are welded

into the middle of the system. Every fix creates two new problems because everything is connected to everything and nothing was built to move.

That is the worst place a product can be. Not a clean failure you can learn from and walk away from. A thing that is always almost done. Forever ninety percent. The developer was not bad. They built for correctness when they should have built for the pivot, and correctness, in a world where you are guaranteed to be wrong about something, is the more expensive mistake.

I have been called in to rescue products stuck in exactly this place. The mobile vet from the last chapter is the clearest one. Two years and thirty thousand dollars, forever eighty percent, and nobody able to name what was wrong. That is the signature of building for correctness without building for change. The detail that matters for this chapter is the after, not the rescue. Before, a small change took a month. Now they send twenty-seven major changes and every one is actioned, tested, and released overnight. That gap is the whole argument for the pivot.

That is what building for the pivot buys you. Not a product that never needs to change. A product that changes so cheaply that changing it stops being the thing you are afraid of.

The three phases that follow are how you build one. We start where every build has to start and where the most damage gets done, before a single line of code exists. We start with getting the idea right.

9. Get the Idea Right

> *The most expensive mistakes on any build are made before anyone writes a line of code. They are made in the gap between what is in your head and what a developer can build.*

You know your industry. That is not in question. You have watched a problem up close for years, you have felt it in a way nobody outside the field ever could, and somewhere in your head there is a picture of the thing that would fix it. The picture is rich. It has edges and exceptions and a hundred small details that matter because you have seen what happens when they are ignored.

And that richness, which is your greatest asset, is also the first thing that sinks the build.

Because the picture in your head is not a plan. It is years of judgement, compressed into a feeling of obviousness, most of which you cannot even see anymore because it has become instinct. When you go to explain it to a developer, you hand over the parts you can articulate and quietly assume the rest. The developer builds what you handed over. And the thing that comes back is wrong in ways you struggle to name, because the parts that made it right were the parts you never said out loud.

This is the work of getting that picture out of your head and turning it into something buildable, without losing the judgement that made it valuable and without drowning the whole thing in detail that does not matter. It is the hardest

part and the one people skip. Get it right and everything downstream is straightforward. Get it wrong and no amount of good engineering will save you, because the team will build the wrong thing beautifully.

Your expertise is compression, and that is the problem

Let me describe what a domain expert really is, because it explains why this phase is so hard.

An expert is someone who has compressed a huge amount of experience into fast judgement. You look at a situation and you know what to do, and if someone asks you why, you can give a reason, but the reason is a reconstruction after the fact. The real decision happened underneath your conscious awareness, using pattern recognition built over years. That is what makes you good. It is also what makes you almost impossible to build software from, because software cannot run on a feeling of obviousness. It needs the rules spelled out, and half your rules are invisible to you.

I saw the purest version of this in an enterprise product I helped build. My partner had roughly twenty years in a rare and demanding field, and the entire point of the product was to take his judgement and put it into software so that people who did not have twenty years could get to a fraction of his answer in minutes. That sounds like a documentation exercise. It is not. It is a fight, over months, to pull decisions out of a person who has forgotten he is even making them. Every screen was a negotiation over a judgement he made in half a second and had never once had to explain, because for twenty years the explanation lived in his head and did not need to leave it.

The lesson from that build applies to every build, at every size. The value you are trying to capture is exactly the part you cannot easily articulate, which means getting the idea right is not about writing down what you know. It is about surfacing what you know but cannot see. And you cannot do that alone, because you cannot see it. You need someone across the table asking the question that makes the invisible thing visible.

Name the one thing it must do

Here is the single most useful exercise in this phase, and it is deceptively hard.

Name the one thing the product must do. Not the five things. Not the list of features. The one job that, if it fails, nothing else matters, and that, if it works, the product is worth having even if everything else is rough.

Every product has one. Underneath all the features and the nice-to-haves and the things you added because a competitor had them, there is a single core job that is the actual reason anyone would use the thing. The mobile vet product had one. Get a booking from a pet owner to a vet with the least friction possible. Everything else, the profiles, the history, the notifications, was in service of that one job, and if that one job did not work smoothly, none of the rest of it would ever matter, because nobody would get far enough to use it.

Naming the one thing does two jobs at once. It gives the whole team a way to judge every future decision, because now every proposed feature can be measured against a simple question. Does this serve the one thing, or does it just come along with it. And it protects you from the most common

failure of this phase, which is building the whole rich picture at once instead of the core of it first.

The test for whether you have found the one thing is whether you can say it in a sentence that a stranger could understand without a diagram. If it takes a paragraph, you have not found it yet. You have found a bundle of things and called it one thing. Keep cutting.

The mistake that sinks everything: building the ninety-nine percent version

Now I want to walk through the specific mistakes that get made in this phase, because getting the idea right is mostly about not making them. The first and most expensive is building for the version of the idea that is complete and correct instead of the version that is small and shippable.

This is the ninety-nine-versus-ninety-five trap from the maths chapter, wearing new clothes. There it was about what a build costs. Here it is about what you decide to build in the first place, which is where the trap actually gets set. The technical instinct picks the ninety-nine percent version that takes two years over the ninety-five that ships in days, every time, because ninety-nine is the better score, and almost nobody stops to ask the only question that matters. Was the ninety-five already enough. Was it already going to change everything for the person who needed it.

It almost always was. People wildly underestimate the ninety-five. The version of your product that is simple and shippable and slightly imperfect is, in the overwhelming majority of cases, more than enough to be worth having, to get in front of real users, to start earning, to start teaching you what you got wrong. The ninety-nine percent version is a

fantasy you pay two years and a fortune for, and by the time you get there, if you get there, the market has moved and the thing you should have been building is different anyway.

I watched an ex-university-executive fall into exactly this. Smart person, real domain knowledge, a genuine data problem to solve. But the instinct was to build the complete, correct, custom version, the one that handled every case and optimised every score, and the product wound itself into so much complexity chasing that last few percent that it became expensive, late, and worse than the simple version would have been. The pattern is not rare. It is the default failure mode of capable, careful people, and getting the idea right means catching yourself doing it before you have spent the two years.

The second mistake: over-scoping out of generosity

The second mistake is subtler and it comes from a good place, which is what makes it dangerous.

You are trying to help. So when you hand over the idea, you hand over everything. Every option you can think of, every edge case, every framework and example, all of it, because leaving something out feels like doing a bad job. You are being generous with context. And the developer, receiving a rich and detailed brief, does the respectful thing and builds all of it, because that is what you do with a thorough brief. You honour the effort.

I told the full version of this story earlier in the book, the operations lead who documented everything to make his technical team's life easier, and got back a heavy, all-encompassing tool that was almost the opposite of the light thing he wanted. The generosity created the over-scope. He

was not describing what to build. He was giving context. But context handed over as a specification gets built as a specification.

The fix is to separate two things that feel like the same thing. There is what you know about the problem, which should be huge and detailed and freely shared. And there is what you are asking to be built, which should be small and sharp. The failure is letting the first become the second. When you brief a build, the context can be a novel. The scope has to be a sentence. Getting the idea right means being generous with the first and brutal with the second, and knowing which one you are doing at any moment.

The way through it is a conversation, not a document. The operations lead's real requirement came out in thirty minutes of the right questions, not from the pile of documentation he had prepared. Someone asking what really helps you here, and what does success look like in a way you have not written down yet. That question, asked at the right moment, does more than any specification, because it goes at the thing you know but did not say.

The third mistake: mixing the lanes

The third mistake is one I will spend a full chapter on later, but it starts here, in the idea phase, which is why I raise it now.

Products live in one of two lanes, and most people who get stuck are trying to live in both at once. One lane is broad and templated. You serve a lot of people, roughly, with a standard product that does not bend much to any individual. The other lane is specialist and careful. You serve a narrow set of people,

precisely, with a product that handles their specific reality in detail. Both are good lanes. Both work.

What does not work is the middle. The product that tries to be broad enough to serve everyone and precise enough to serve each one exactly. That product carries the cost of the specialist lane, all that careful custom handling, and the price expectations of the broad lane, and it collapses under the weight of trying to be both. It is too heavy to be cheap and too generic to be precise. It is the most expensive kind of stuck, because every decision pulls in two directions and the team is forever building for a customer who does not exist, the one who wants the depth of the specialist product at the price of the templated one.

The reason this belongs in the idea phase is that the lane is a decision about what the idea is, not about how you build it. If you have not chosen your lane, your one thing is unstable, because it means something different in each lane. Choose the lane while you are getting the idea right, or you will discover, expensively, halfway through the build, that you never decided what you were making.

What getting the idea right feels like when it is done

Let me tell you what the finish line of this phase looks like, so you know when you have crossed it.

It is not a specification document. It is a specific feeling. It is the moment the picture stops being rough and starts being settled, when you can describe the problem and the one thing the product does to solve it plainly enough that someone who was not in the room could understand it without asking a clarifying question. I described this earlier as the difference

between probably clear and genuinely clear, and I have learned not to trust probably clear, because probably clear is where the expensive assumption is hiding.

You get there by iterating, not by planning. You put a rough picture of the problem on the table, knowing it is partly wrong. You ask why it is wrong. You put a better picture down. You keep going, moving around the map, running backwards from the outcome to the start, until the shadowed parts fill in and you can see the whole thing. It is uncomfortable if you have a bias toward getting started, which I do. Every instinct says start building, you will figure it out as you go. And you will figure some of it out. But the things you figure out by building the wrong thing cost weeks. The things you figure out by asking one more question cost thirty seconds. The more you stop, question, and repivot before you build anything, the better the outcome. That is not a personality preference. It is the maths of the thing.

When the picture is settled, when you have named the one thing, separated your context from your scope, and chosen your lane, you are done. You have an idea that is right, which is to say an idea that is small enough to build, clear enough to hand over, and honest about what it is.

But right is not the same as scoped. You know what the product must do. You have not yet decided what the shippable version of it looks like, the ninety-five percent you can put in front of a real person in days. That is the next phase, and it is where the idea becomes a plan you can build. We scope the simple version.

10. Scope the Simple Version

> *There is a version of your product that ships in days and a version that ships in years. The days version is almost always the better product. This chapter is about how to find it.*

You have the idea right. You know the one thing it must do, you have separated what you know from what you are asking to be built, and you have chosen your lane. Now comes the phase where most of the value is either captured or thrown away, and it turns on a single decision that sounds trivial and is not.

What, exactly, is the smallest version of this that is worth shipping.

Not the smallest version you can imagine. The smallest version that a real person would use and get real value from. There is a floor, below which the thing does not do its one job and nobody wants it, and there is a ceiling, above which you are adding weight that does not earn its cost. The whole game of this phase is finding the point just above the floor and refusing, hard, to build toward the ceiling. That point is what I call the ninety-five percent that ships. It is the version that does the one thing well and leaves almost everything else out, and it is the version that will teach you more in its first week in front of a user than a year of planning ever could.

The ninety-five that ships beats the ninety-nine that does not

I keep coming back to these two numbers because the gap between them is where fortunes and years disappear.

The ninety-nine percent version is complete. It handles every case, covers every exception, polishes every edge. It is also, in almost every build I have seen, a trap, because the last few percent of completeness costs an amount of time and money that is wildly out of proportion to the value it adds. Technical work is exponential. The difference between a simple version of something and a complete version is not a bit more work. It can be the difference between a week and six months, because each additional case you handle interacts with all the others, and the complexity compounds rather than adds.

So you have a choice. You can spend six months getting to ninety-nine, arriving late, expensive, and with a product built entirely on assumptions you have never tested against a real user. Or you can spend a week getting to ninety-five, shipping now, cheap, and learning within days which of your assumptions were wrong so you can fix them while it is still cheap to fix them.

Said that starkly, it sounds obvious. In the room it does not feel obvious at all, because the missing four percent is visible to you and it itches. You know the edge case is not handled. You know there is a scenario the product does not cover. And the pull to just handle it, just cover it, just get it right before anyone sees it, is enormous, especially if you are careful and conscientious, which the best domain experts are.

Here is what I have learned to hold onto in that moment. The user does not experience the four percent you are missing. They experience the ninety-five percent you shipped.

And the ninety-five percent, if you scoped it around the one thing, is the part that changes their life. The four percent is the part that changes your score. Ship the ninety-five. Learn what is missing from a real person instead of from your own anxiety, and you will nearly always find that the four percent you were agonising over was not the four percent that mattered.

Cutting to the core is subtraction, not addition

Scoping the simple version is a subtraction exercise, and almost nobody treats it that way. The instinct is to build up a list of what to include. The discipline is to start from the whole rich idea and cut everything that is not the core, one thing at a time, until removing the next thing would break the one job.

Here is how the conversation goes when it goes well. You put a feature on the table and you ask, honestly, does the one thing still work if this is not here. If the answer is yes, it comes out of the first version. Not out of the product forever. Out of the first version. It goes on a list of things to add once the core is live and earning and teaching you. Most features survive that question badly. You discover that the thing you thought was essential is really a comfort, something you wanted so the product would feel complete, and complete is not the goal of this phase. Alive is the goal.

I have run this cut down to ninety percent scope reduction in a single conversation. The operations tool I mentioned earlier went from a six-month build to a one-day build because once the real core was clear, ninety percent of what had been scoped turned out to be weight, not substance. That

is not unusual. When you apply the does the one thing still work without this test with discipline, most builds are carrying two thirds more than they need for a first version. The reason they carry it is that nobody made them cut, because cutting feels like losing and adding feels like progress. In this phase, cutting is the progress. Every thing you remove is a thing that cannot break, cannot delay you, cannot get in the way of the real user reaching the core.

There is an emotional trap here worth naming. Cutting a feature you love feels like conceding the product is less than you dreamed. It is not. The dream is intact. You are sequencing it. The cut version ships first, earns first, and teaches you which of the dream's other parts are real and which were only ever in your head. You will add back the ones that turn out to matter, and you will be glad you never built the ones that did not.

Pick one lane, and never both

I raised the lanes in the last chapter, as part of getting the idea right. Now, in scoping, it becomes a hard operating decision that shapes every cut you make, so I want to give it its full weight.

There are two lanes. Broad and templated, or specialist and careful. Pick one.

Broad and templated means you serve a lot of people with a standard product. It does not bend much to any individual. Its strength is reach and simplicity and low cost to run, because everyone gets the same thing, and the same thing is cheap to build once and sell many times. Its scoping discipline is ruthless standardisation. You resist every request to handle a special case, because a special case is the enemy of the

templated lane. The moment you start bending the product to one customer, you have left the lane, and the economics that made the lane work fall apart.

Specialist and careful means you serve a narrow set of people with a product that handles their specific reality in depth. Its strength is precision and value and defensibility, because you handle the hard cases the broad products cannot, and the people who need those cases handled will pay for it and stay. Its scoping discipline is depth over reach. You say no to whole categories of user so you can go deep for the ones you chose. You accept a smaller market in exchange for being genuinely, specifically right for it.

Both lanes ship simple versions. But they cut differently. The broad product cuts anything that is not standard. The specialist product cuts anything that is not deep for its narrow user. What kills people is trying to scope a simple version without having chosen, because then every cut is ambiguous. Is this feature core, or is it a special case. Depends on the lane, and if you have not picked one, you cannot answer, so you keep it, and you keep the next one, and the simple version bloats back into the thing you were trying to escape.

The enterprise product I worked on was firmly in the specialist lane, and it made the scoping clear. Because end users were mandated to use it, sold in centrally rather than chosen individually, the product did not need broad appeal at all. It needed to be dead simple for the specific mandated user and deep enough to capture a twenty-year career. Every scoping call flowed from that. We cut anything aimed at winning over a user who had a choice, because that user did not exist here. A broad product would have made the opposite cuts. Neither is right in the abstract. Each is right for its lane. The failure is having no lane and therefore no basis to cut.

The thirty percent that is yours and the seventy percent that is not

Now the second big decision of this phase, and the one that determines whether the build comes together fast or grinds to a halt at eighty percent. You have to divide the whole product into two piles. The part that is genuinely unique to you, and the part that is not.

Here is the thing that took me years and hundreds of builds to fully trust. In most software, roughly thirty percent is genuinely unique. The rest, call it seventy percent, is the same bones every product has. Accounts and logins. Storing data and getting it back. Taking payments. Sending emails. Hosting the thing so it stays up. Keeping a record of what happened. None of that is special to your product. Every product needs it, it has all been solved, and solved well, by companies whose entire business is solving that one piece for everyone, and you can assemble it rather than build it.

The thirty percent is the part that is genuinely yours. It is your one thing, the specific capability that is the reason your product exists and nobody else's does the same job. For the enterprise product, the thirty percent was the logic that captured the expert's judgement. For the vet, it was the specific booking workflow that removed the friction between owner and vet. Everything around that core, in both cases, was the standard seventy percent that every product needs.

Scoping the simple version means being honest about which pile each thing goes in, because the two piles get treated in completely opposite ways, and getting the treatment backwards is the most common way builds die. I will cover the how of the build in the chapters that follow. But the scoping decision, the sorting of the piles, happens here, and it happens with a simple question asked of every part of

the product. Is this the reason we exist, or is this something every product has. If it is the reason you exist, it goes in the thirty and it gets your real effort. If every product has it, it goes in the seventy and you will assemble it from proven tools rather than build it, no matter how much your instinct says you could do it better yourself.

That instinct, by the way, is the enemy here. Technical people especially want to build the seventy percent themselves, because they can, and because a payment system or an authentication system they built feels more theirs than one they wired in. That feeling is exactly the trap. Every hour spent hand-building a piece of the seventy percent is an hour not spent on the thirty percent that is the only part anyone will pay for, and it is an hour spent creating a custom piece that you now have to maintain forever, that nobody else is improving, and that cannot be swapped out when it turns out to be wrong. The seventy percent is not where you compete. Treating it like it is is how you end up stuck.

What a scoped simple version looks like on paper

Let me pull this together into what you should have at the end of this phase, so it is not abstract.

You should have a description of the smallest version worth shipping, defined by the one thing it must do, with everything that does not serve that one thing cut into a later list rather than built now. You should know your lane, broad or specialist, and every cut should be consistent with it. And you should have every part of that scoped version sorted into two piles, the thirty percent that is genuinely yours and gets

your real effort, and the seventy percent that every product needs and will be assembled from proven tools.

That is a plan you can build. It is small enough to ship in a timeframe measured in days and weeks rather than quarters and years. It is honest about what it is. And it is structured so that the effort goes where the value is and the plumbing gets bought rather than built.

Notice what this plan is not. It is not complete. It does not handle every case. It is, deliberately, the ninety-five percent, scoped to teach you fast what the missing five percent really is, from real users rather than your own worry. It is built for the pivot, because everything in the seventy percent is a swappable proven tool rather than a welded custom part, which means when you learn you were wrong, and you will, you move instead of rebuild.

You have the scope. Now comes the part everyone thinks is the whole job and is the smallest part of it if the first two phases were done right. You build it. And building it well is mostly a matter of doing what the pile-sorting told you to do, stitching the proven seventy percent together fast and spending your real effort only on the thirty percent that is yours. That is what the rest of the method covers, and it is where the simple, flexible, shippable product finally comes together.

11. Software Is Not What You Think It Is

> *You can skip this chapter. But if software has ever felt like a language everyone else speaks and you do not, ten minutes here will change how you read the rest.*

I am going to stop the book for a moment and do the thing every book like this assumes it has already done. I am going to tell you what software actually is.

Because here is the problem with everything up to this point, and everything after it. I keep using words. Backend. Stack. Framework. Shipping. I use them the way everyone in this world uses them, as if you were born knowing what they mean, and it leaves you nodding along to a conversation you are only half inside. So this chapter is the software one oh one nobody bothers to offer you, with my hopefully unbiased opinion on what matters and what is just noise.

If you already know a backend from a frontend and why nobody in tech ever agrees on anything, skip ahead to the method. You will lose nothing. But if software has always felt like a black box that other people hold the key to, give me these ten minutes. Everything after this reads differently once the box is open.

The first thing, and it is the biggest

Software is not a special, exact science that only a certain kind of mind is allowed to touch.

It feels that way. It is built to feel that way. It arrives wrapped in words that keep you standing outside, and the people inside are not always in a hurry to let you in, because the mystery is worth money. But underneath the words, building software is just a complex project. That is all it is. It is no different in shape from building a house, standing up a new factory line, or launching a new arm of a business. It has parts that fit together. Some of those parts are load-bearing and some are decoration. It runs late when it is planned badly and comes in clean when it is planned well.

And like every complex project, there are a million ways to do it, and every single person who does it for a living is quietly certain their way is the right one. This is the thing to understand before you hire anyone or believe anything. Most of what you will be told as fact is taste, dressed up as fact. You have to use this one. Nobody builds it that way anymore. That is the wrong database. Some of that is true. Most of it is preference wearing the costume of expertise. Once you can hear the difference, you stop being at the mercy of whoever spoke last and loudest.

What the thing is made of

Let me open the box, plainly.

Code is just instructions. A long, precise list of steps written for a machine that does exactly what it is told and nothing it is not. That is the whole of it. It is not magic and it is not maths. Just instructions, and instructions can be written well or badly, clearly or in a tangle, the same as any set of instructions you have ever been handed.

Every product has two halves. The frontend is what the user sees and touches. The screens, the buttons, the parts

with a colour. The backend is everything behind the glass, where the information lives, the rules run, the money moves. If the product were a restaurant, the frontend is the dining room and the backend is the kitchen, the books, and the loading dock. Most of what makes a product solid or fragile sits in the back, where the customer never looks, which is exactly why you have to.

The stack is just the collection of tools and materials someone chose to build the thing with. Brands of brick, in effect. The database is where the information is kept so it can be found again. None of these words is as clever as it sounds.

The language argument, and why it barely matters

You will get pulled into arguments about which programming language to build in, and they will sound consequential. They are mostly not.

A programming language is a tool, like a brand of power tool. A great builder makes a great thing with almost any of them. A poor one makes a mess with the best on the market. The differences people argue about, this one is faster, this one is more modern, this one is what the serious people use, almost never decide whether your product lives or dies. I have watched good products win on a boring, unfashionable language and cool products die on the newest thing going.

So here is my whole opinion on the language question, and it is worth roughly what the question deserves. Choose the boring, proven one that a lot of people already know. Not the fastest on paper. Not the newest. The one with a deep bench, so you are never held hostage by the single person on earth who understands your particular corner. Boring and

popular beats clever and rare almost every time, because your real risk was never speed. It was being trapped.

So where do the advantages actually come from

If the language barely matters, and the tools are mostly interchangeable, then where does the real edge come from? Not the technology. It is everything around the technology: how the work is structured, who is chosen to do it, how tightly the scope is held, whether the thing was planned to be moved or planned to be perfect and therefore doomed. Those are the levers, and they are the whole rest of the book. This chapter was the vocabulary, and the one belief that frees you from being managed by jargon. What comes next is where the advantage lives.

A few more words you will hear, quickly

Keep this last part as a pocket dictionary rather than a chapter. These are the phrases that will get thrown at you in a build meeting, with what they actually mean and where the meter tends to hide inside them.

“We’ll build you an MVP.” A minimum viable product. The smallest version that does the one thing it is for, and nothing else. Not a rough draft, not a throwaway. The spine you build the rest on. If someone uses it to mean a cheap half-thing we replace later, walk.

“We just need to integrate with their API.” An API is the socket one piece of software offers so another can plug in. Every integration is a thing you did not have to build yourself. Good news, almost always. When someone wants to

build the thing an API already gives you, that is the meter running.

“There’s some technical debt to pay down.” The mess left behind when things were built fast or built wrong. A little is normal. Too much shows up as small changes taking longer and longer, and every new feature breaking two old ones. That is the interest, and someone took the debt on without telling you.

“We’ll just refactor it.” Rewriting code to be tidier without changing what it does for the user. Sometimes genuinely needed. Sometimes a developer redoing their own work on your clock. Ask what you, or your customer, will be able to do afterwards that you cannot do now. If the honest answer is nothing, you are paying for someone’s tidiness.

“That’s an edge case.” A situation that happens rarely. Worth knowing about, rarely worth holding the launch for. Most can wait until a real customer hits one.

“It’s basically done.” The most expensive sentence in software. Done and almost done can be a week apart or a year apart, and the person saying it often cannot tell which. Ask what specifically is left, as a list, with time next to each item. If they cannot make the list, it is not done.

12. The Fifty, the Thirty, and the Ten

> *Engineering stopped being the hard part a while ago. Choosing the right ten percent is the hard part now. Almost nobody has caught up to that.*

Earlier in this book I told you something that probably felt generous. I said that about thirty percent of your product is genuinely yours, and the other seventy percent is proven plumbing you should never reinvent. That thirty makes it feel new. The seventy is logins and payments and storing data and sending an email when something happens, and you do not have to build any of it from scratch because it already exists and it already works.

That was true. It is still true. But it was the beginner's version of the truth, and I gave it to you on purpose, because you needed to stop being afraid of the seventy before you could hear the real number.

So here is the real number. On any new product I build, the part that is actually new is not thirty percent. It is closer to ten.

That number has a name, because I lean on it constantly and by the end of this book you will too. The ten percent rule. Almost all of any product already exists; the genuinely new part, the part that deserves your money and your judgement, is about a tenth of it. I named it on the first page and told you it was the one idea to carry out of here. This chapter is where I make good on the number.

Let me show you how that happens, because it is the single most important idea in everything I do, and once you see it you cannot unsee it. It is also the reason I can build a serious product for a fraction of the normal cost and time, and I want to be honest about that reason, because it is not the reason people assume. It is not that I am cleverer than the person quoting you two hundred grand. It is not heroics. It is that I am only ever building a small slice of the thing from scratch, and they are building the whole thing every time.

Three layers, and only one of them is new

Every product I have ever built is three layers stacked on top of each other. Once you can name the three layers, you can look at any quote, any build, any founder's grand plan, and see straight through it to where the money is actually going.

Think of a car. Nobody who builds cars builds a car from a pile of ore. There is a chassis, the platform underneath that every model in the range shares. There is an engine, the bit that does the actual work of moving you, and the same engine gets dropped into several different cars. And there is the body and the trim and the badge and the price, the part that makes this one a family wagon and that one a sports coupe, even though they are the same machine underneath.

Software is exactly this. Chassis, engine, wrap. Let me take them one at a time, because the proportions are the whole point.

The chassis, about half of it, and none of it is yours

The chassis is the foundation every product sits on. Hosting, so the thing is online and stays online. The database, where all the information lives. Payments, so people can pay you. Sign-in, so people have accounts and the right person sees the right thing. Sending email. Storing files. The security, the audit trail, the boring, invisible, absolutely-load-bearing machinery that every product on earth needs and no customer ever thanks you for.

This is roughly half of a product. Half. And here is the thing that should change how you think about every quote you ever get. It is the same half on almost every product in the world. Your idea and a completely different idea, in a completely different industry, sit on a chassis that is about ninety percent identical. The login screen on a mining compliance tool and the login screen on a wedding-planning app are doing the same job in the same way.

None of it is your competitive advantage. Nobody has ever chosen a product because it had a really good audit trail. And yet everybody needs one.

So I built mine once. Years ago. And I have never built it again. Every new product I start begins with the chassis already there, already paid for, already proven in production, already carrying real customers and real money without falling over. When a first-time founder gets quoted for a build, a huge slab of that quote is someone charging them to build this half from scratch, custom, as though the login screen had never been invented. That is not a build. That is a meter running.

The engine, thirty to forty percent, and this is where your expertise lives

The engine is the clever part. It is the reusable capability that does the specific hard job at the heart of the product. Parsing a messy file and pulling the real structure out of it. Analysing a video and understanding what is happening in it. Scoring something against a standard. Taking a pile of inputs and generating a finished document a human would have spent a day writing.

This is the part that is genuinely hard to build. This is where the intellectual property is. When I told you earlier that thirty percent of your product is yours, this is the thirty I meant. The engine is the reason someone chooses your product over a spreadsheet, and it is the part the market cannot fake, because it took real skill and real time to make it work.

Now here is the piece you have not seen yet. The piece that takes the beginner's version of the truth and turns it into the real one.

An engine, once it is built, does not power one product. It powers about five.

The engine that parses a messy file does not care what industry the file came from. Build it once, properly, and you can wrap it as a tool for accountants, and as a tool for lawyers, and as a tool for procurement teams, and as a tool for researchers, and each of those looks like a completely different product to the person using it, and underneath they are the same engine wearing four different coats. The engine that analyses a video is the same engine whether the video is a vet examining an animal or a coach reviewing a training session or an inspector walking a site.

This is the distinction that matters, so let me say it plainly. An engine that only ever powers one product is not an engine. It is just a product. The moment I build a capability, I am already asking what else it can drive, because the second,

third, and fourth uses of that engine are close to free, and the first one is where all the cost was.

So on the very first product an engine appears in, yes, you are paying to build it. And I want to be square with you about what that means, because if this is your first product the engine is not free, it is the expensive part. The ten percent number is what a build becomes for me, on my tenth or fiftieth product, when the engine I need is already sitting on the shelf. It is not what your first one will feel like. On your first one, the honest split is closer to the thirty I gave you earlier: a chassis you can buy, a wrap that is days of work, and an engine that is real money and real weeks and the bulk of your budget, because it has never been built before and someone has to build it once.

The ten percent is not a lie. It is a promise about what happens next. Build the engine once, properly, and it stops being a cost and becomes an asset, and the second product it powers really does drop to that ten. The same is true if you partner with someone who already owns the engine your idea needs, which is a large part of why partnering can collapse the number so fast: you are renting a shelf someone else already paid to stock. But nobody should read this chapter, look at their first build, and expect to pay for a tenth of a product. You are paying to build the thing that makes every product after it cheap. That is a good trade. It is just not a free one, and I would rather you walked in knowing which number is yours.

The wrap, about ten percent, and it is the only new thing

Which leaves the wrap.

The wrap is who this product is for. What the output looks like when it lands in front of them. What it is called. What it costs. Where you find the people who need it. The words on the page, the colour of the button, the shape of the report that drops into their inbox.

That is it. That is the genuinely new part of any given product, sitting on top of a chassis I already built and an engine I probably already built. And it is not months of work. It is days to weeks. The wrap is the ten percent.

Sit with that, because it rearranges everything. When you have an idea and you feel the weight of it, the whole thing, the whole daunting machine you would have to build, most of what you are feeling is the chassis and the engine. And most of that, on a good build, already exists. The part that is actually, uniquely yours, the part that has never been built before, is a thin and glorious ten percent on the very top. That is the part worth obsessing over. That is the part worth your money and your judgement and your late nights. The other ninety is a solved problem you should refuse to pay to solve again.

Sameness is the feature

Here is a thing that sounds wrong until you see why it is right. My stack is deliberately identical across every product. Same chassis, same tools, same patterns, same way of doing sign-in, same way of handling payments, same everything, on purpose, every single time.

Most people assume variety is a sign of skill. That a good builder reaches for the perfect bespoke tool for each new job. The opposite is true, and the reason is the most valuable idea in this chapter after the three layers themselves.

When the stack is the same everywhere, every hard-won lesson becomes a permanent asset. The first time I hit a nasty trap in payments, I lost days to it. I solved it once. And because every product I build sits on the same chassis, that solution is now baked in everywhere, forever. I will never lose those days again, on any product, for the rest of my life. The trap that took a week the first time takes zero minutes the second time, and the third, and the hundredth.

Now imagine the opposite. Imagine a builder who reaches for a different tool on every project, a fresh custom foundation each time, because it feels more tailored and more impressive. Every trap they hit is a one-off. They pay for it, they solve it, and then they walk away from the solution and go and hit a brand new set of traps on the next job. They are always starting over. They are always at day one of the same fight. That is why their builds are slow, and expensive, and fragile, and it is not because they are working less hard than me. They are working far harder than me. They are just rebuilding the ninety percent every single time, and calling it craft.

Sameness is not laziness. Sameness is compounding. Every product I ship makes the next one cheaper and faster and more solid, because they all share the same spine, and the spine only gets stronger. That is the machine. That is the actual secret, and it is boring, and it works.

This is why the numbers are what they are

Go back to two examples I have mentioned in this book and the numbers will finally make sense.

There is an enterprise platform I built that cost around a hundred and fifty thousand dollars and did over three million

in its first year. On paper, the normal price for a build like that is two to three million. People assume the gap is corners cut, cheaper labour, some compromise you cannot see. It is not. The gap is that a two-to-three-million build is paying to construct the chassis and the engine and the wrap all from scratch, custom, as one giant bespoke monolith. My version paid almost nothing for the chassis, because it already existed. It paid for the engine, because that build genuinely needed a new one, and the engine was the hard, valuable heart of it. And it paid for the wrap. The money went where the value was. Nowhere else.

Then there is a mobile product I built for a vet. That one had been built before, badly, the slow expensive way. I rebuilt it in weeks. Not because I am fast in some superhuman sense. Because the chassis was already there and the engine mostly was too, so the rebuild was almost entirely wrap, and wrap is a weeks-long job. What had been a mountain the first time was a hill the second time, because ninety percent of the mountain was already standing.

Neither of those is a story about cutting corners. Both are stories about refusing to pay twice for the same ninety percent.

The trap on the other side, again

I am not going to sell you the ten percent as though small always meant cheap, because that is the fastest way to ruin a good product, and I have watched it happen more than I would like.

If the wrap is the only genuinely new part, then the wrap and its engine are the whole product, and starving them is fatal. I have seen founders hear “only ten percent is new” and

translate it straight into “so it should be nearly free,” and take the budget down and down until the only thing left to cut is the part that was the entire reason to build the thing. A hobby budget aimed at the ten percent that is your actual product fails just as certainly as a bloated budget wasted rebuilding the ninety. Both get the math wrong. One overpays for what already exists. The other underpays for the only thing that does not.

The point was never cheap. The point is that the spend goes to the ten percent that is the product, and almost nothing goes to the ninety that is already solved. Spend hard on the part that is yours. Refuse to spend a cent on the part the whole world already built. That is the discipline, in both directions.

The one question that cuts through all of it

So here is what to do with this, and it is a single question.

The next time someone quotes you a build, before you flinch at the number, before you go and hide the idea back in the folder, ask them this. How much of this already exists, and how much are you building from scratch?

Then watch what happens. A good builder will light up, because it is their favourite question, and they will tell you exactly which parts are chassis they already have, which parts are engine they can reuse or need to build fresh, and which thin slice is the genuinely new wrap that your money should go to. They think this way already. You are speaking their language.

A builder who wants to custom-build the chassis will get uncomfortable, because you have just pointed at the meter

and asked why it is running. There may be a real answer. Sometimes there is a genuine reason the foundation has to be bespoke. But most of the time the honest answer is that they build everything from scratch because that is how they have always done it, and the discomfort on their face is the most useful information you will get in the whole conversation.

You do not need to become the person who can build the ten percent. You have a whole book here that is not going to turn you into a data scientist and does not need to. You need to be able to see the three layers, name them out loud, and ask the one question that tells you whether the person in front of you is charging you for the ten that is yours or the ninety that already exists.

Engineering is no longer the bottleneck. It really is not. The chassis is a solved problem, the engines are increasingly things you can reuse and reach for, and the raw ability to build has never been more available. What is scarce is the judgement to choose the right ten percent and to build that ten percent well. That is the entire game now, and almost everyone is still playing the old one, rebuilding the ninety by hand and wondering why it costs so much.

So here is the ten percent rule in a single line, the one to keep. You are only ever building ten percent of it. The whole game is choosing the right ten percent, spending on that and nothing else, and building it well. Everything after this chapter is how you do exactly that.

Which brings us to the real work. Choosing the ten percent is one thing. Running the build of it well, so the money and the weeks actually turn into the product you pictured, is another. That is what the rest of this book is for. The chapters that follow are the levers, the specific ways you run the build of that ten percent so it lands. The what is settled now. Next comes the how.

13. Context and Communication

> *Everyone in the room is trying hard to work together. That is exactly the problem.*

I want to start with a story, because it shows the thing better than any principle I could write down.

An operations lead needed a tool built. He was a good leader. Pragmatic, supportive, genuinely invested in making it work. And because he was a good leader, he did what good leaders do when they hand work to a technical team. He tried to make their lives easy. He documented everything. He laid out all the options. He mapped the manual process in detail. He included frameworks, examples, and ideas for how the technical solution might work. He handed it all over and felt confident he had given the team everything they needed.

The technical team received it and thought exactly the same thing. They had been given a full brief. Their job was to build what was in front of them, as completely and as solidly as possible, because that is how you respect the effort someone put into preparing a brief. So they built it. All of it. Every option, every framework, every idea the operations lead had included to be helpful.

Here is the problem. The operations lead was not describing what he wanted built. He was trying to make the technical team's job easier by giving them context. What he actually wanted, what he never said out loud because it felt too obvious to say, was a light tool he could update every few

weeks as site processes changed. Something simple enough that his team would trust it straight away. Something he could launch once and never have to launch again. He wanted speed, flexibility, and simplicity. He did not want everything he had written built exactly as written.

The technical team wanted the opposite. They wanted specificity. Solidity. Stability. Something built properly that would not need to change. So they built something thorough, technically sound, and almost completely wrong for what the operations lead actually needed.

The conversation that would have caught this took thirty minutes. Not a workshop. Not a discovery phase. Thirty minutes of someone asking the right question. What actually helps you here, and what does success look like in a way you have not written down yet. Once that question was asked, the operations lead answered it straight away. He was a deeply pragmatic person. When someone laid out the tradeoffs honestly, we can remove this dynamic logic, it cuts most of the complexity, does that work for you, he said yes without hesitation. Of course. That is what he wanted all along.

The scope dropped by ninety percent. The solution that had been six months in progress was rebuilt in a day.

Sit with that for a moment, because it is the whole chapter in one number. Six months of real work, from a capable team who were doing their honest best, all pointed at the wrong target. Nobody was lazy. Nobody was incompetent. The gap was not in the code. It was in the thirty minutes that never happened.

Technical work is exponential in its complexity. Adding a small amount of scope does not add a small amount of work. It can double or triple the complexity of the whole system. The difference between a simple version of something and a thorough version is often the difference between one week

and six months. Which means doing less, but doing the right less, precisely defined and well understood, is almost always the most powerful decision available on a technical project. And the reason this is hard is not that it takes skill to remove scope. It is that you have to know which scope to remove, and you can only know that if you have closed the gap between what someone said and what they meant. That gap is where the six months goes.

Tech has to live in systems, not features

There is a second context failure that is just as common and just as expensive, and it lives one level deeper than the first.

Most people have heard the term technical debt. They understand it in a vague sense. It is the accumulated cost of shortcuts taken earlier that make everything harder later. The best way I have found to describe it is this. You pick a tunnel to dig. You dig it for three months. You then find out it is going in the wrong direction. You have to climb out, start again, and dig a new tunnel, except now you are three months behind, the original tunnel is in the way, and everything you learned digging the wrong one is only partly useful for digging the right one.

Technical debt is not laziness. It is almost always the result of building features without understanding the system those features need to live inside. Go back to the operations lead. Once the real requirement was clear, the pattern underneath it became obvious. The system needed to change often. Instead of building each component explicitly, ten survey questions, each with its own data flow, its own reporting logic, its own maintenance overhead, you build a dynamic system where the underlying logic is written once and the specific questions sit on top of it as a surface layer.

Want to add a question? You add it to the surface. The system does not change. Only the top layer does.

This is the difference between building features and building systems. Features are specific answers to specific questions. Systems are flexible frameworks that can generate many different answers from the same underlying logic. A system-oriented approach inverts the cost curve. The initial build takes longer, because you are designing for flexibility. But the cost of change stays flat, or drops as the system matures, because changes are surface level only. Quick, safe, and easy.

I want to be honest about how hard this actually is to get right, because the person who leaves a product stuck at eighty percent usually got this exact decision wrong, and they got it wrong for understandable reasons. Building features is faster today. It feels like progress. Every feature you build explicitly ships something you can point at. Building a system is slower at the start and produces nothing you can demo for a while. Under deadline pressure, with a client asking why nothing is visible yet, most developers reach for features. Then the tenth change request arrives, and the eleventh, and each one takes longer than the last, and nobody can point at what is actually broken. That is technical debt arriving on schedule. It is not a failure of intelligence. It is a failure to make an unglamorous call early, when making it was cheap and invisible.

The AI version of this makes it concrete. The instinct when building an AI-powered product is to fine-tune a model for the specific use case. This feels thorough. The problem is that fine-tuned models are tunnels. Instead, the system-oriented approach separates the layers. The model is a general capability. The prompts are the surface layer that directs that capability at your specific use case. When something needs to change, you change the layer that needs to change, usually the

prompt, without touching anything else. Knowing to do that, before you have dug three months into the wrong tunnel, is not obvious. It is pattern recognition. It comes from having dug the wrong tunnel enough times to smell it coming.

What you are actually trying to do

There are two circles. The first is the technical circle. What can be built, how it works, what the constraints are. The second is the user or business circle. What problem needs solving, how the work actually gets done today, what good looks like from the perspective of the person who has to live with the outcome. Your job is to find the middle of the Venn diagram.

The natural human instinct, and this applies to technical people more than almost anyone, is to go down rabbit holes. A detail surfaces, it seems important, and suddenly everyone is two hours deep into a conversation about an edge case that may not even matter. The discipline of context mapping is the discipline of not doing that. Of staying at the level of the problem rather than the solution for longer than feels comfortable.

The way I approach it is iterative rather than linear. You ideate. Put a picture of the problem on the table, even if it is rough and partly wrong. Then you question why it is wrong. Then you ideate again with the new information. You keep going until the picture feels complete. Not perfect. Complete. The test is whether you can describe the problem clearly enough that someone who was not in the room could understand it without needing to ask a clarifying question.

I find myself asking what am I missing here more than almost any other question. Not once. Repeatedly, at different

points in the conversation, in different words, coming at the same territory from different angles. It is less like a checklist and more like navigating a map where parts of it are still in shadow. You keep moving to different sections, running backwards from the outcome to the starting point, until the shadow disappears and you can see the whole picture.

The counterintuitive part, and this catches almost everyone who has a bias toward getting started quickly, which I do, is that the more you stop, question, and repivot before you build anything, the better the outcome. The right question closes the context gap. Not a full interrogation. One specific question, asked at the right moment, that surfaces the assumption that was about to become an expensive mistake. Knowing which question that is, out of the hundred you could ask, is the skill. It looks effortless from the outside. It is not.

What this takes in practice

I have a bad habit of wanting to get started. I find thinking by doing more natural than thinking by thinking, and the pull toward building something, even something rough and provisional, is strong enough that I have to actively push against it.

What I do in practice is keep asking the same question in different words until I feel genuinely clear rather than probably clear. There is a specific feeling when context is actually complete. A kind of settledness where you stop having the nagging sense that something important has not surfaced yet. I have learned not to trust probably clear, because probably clear is usually where the expensive assumption is hiding.

The other thing I do is treat the first pass at almost anything as a throwaway. Not because I expect it to be wrong, but because building a light version and seeing where it breaks is often the fastest way to surface the context gaps that conversation alone did not find.

I also genuinely believe that process owners, the non-technical people who live inside the problem every day, are often better at this kind of context mapping than technical people, once you give them the right frame. Technical people tend toward efficiency and specificity. Process owners understand the problem in a way that is richer and more textured, because they have been living with it. If you can get a process owner to stop trying to describe a solution and just describe their experience of the problem, what is hard, what is slow, what breaks, the context you get is usually far more useful than anything a requirements document produces.

One of the best developers I have worked with is impressive for reasons that have almost nothing to do with technical skill. What makes him genuinely rare is this. He can take an ask, get underneath it to the real context, change direction when the first approach is wrong, put ideas forward and discard them without ego, and keep reorienting until the picture is clear. None of those are technical skills. All of them are context skills. And it is one hundred percent why the work he produces is so consistently right. I mention him because people looking to hire a developer look for the wrong thing. They look for the person who knows the most tools. The person you actually want is the one who can find the middle of that Venn diagram, and that person is much harder to spot and much rarer to find.

A worked example, the support automation project

The goal sounded straightforward. Automate as much of the work in our support contracts as possible. From an email with an issue to a resolution, make that process better and faster. That is broad. So we did not start by designing a system. We started by poking at the problem.

First pass. We thought managing emails would be where the biggest gain was. We set up an automation flow, some email forwards, looked at how tickets flowed into our tracking tool. It worked technically. But when we looked at it honestly, we had not really solved anything. We had automated the easy part and left the hard part, triaging and diagnosing the actual issue, completely untouched. Light demo, no real infrastructure, no wasted time. But we knew we had not found the problem yet.

Second pass. Could we automate the triage? We ran through real examples and found some tools that could export diagnostic information. We manually copied the output into ChatGPT to see if it could identify the issue. It could. But the process was heavy, slow, and completely unscalable. Not a solution. But it proved the triage was solvable in principle, which was useful information.

Third pass. Could we get that diagnostic information without the manual export step? We reviewed the relevant APIs, ran some one-off tests, and found we could pull most of what we needed programmatically. Now we had two pieces, automated information gathering and AI-assisted triage, that worked independently.

At this point we stopped and reassessed. We could see the value clearly, but we could also see the risk. Full automated triage felt like too much exposure too soon. What could we do

that cut the workload significantly without taking on that risk? The answer came from thinking about what our team actually spent their time on. Not the diagnosis itself, but everything around it. File versioning. Storage management. Change logs. Documentation. If we could automate those and give the team better diagnostic information to work from, we would remove ninety percent of the effort without touching the part that needed human judgment.

So that is what we built. End to end on a demo first. A drag and drop workflow tool, because it forces you to stay light. A few weeks of running back and forth, finding the gaps, fixing them, testing again. And then an end-to-end system that handles ninety percent of complex support work.

The path from there to a production-ready system is not a fundamental change. It is more security, more error handling, more edge case coverage. The same thing you do when you take a presentation you built for a colleague and prepare it to send to an external audience. Same core content, more care around the edges. Development is no different. But notice how much judgment sat inside that story. Four separate decisions to stop and reassess. One decision to deliberately not build the riskiest, most impressive part. That restraint is the skill, and it is the opposite of what most builds do, which is chase the hardest version because it is the most interesting.

What it looks like when it goes wrong

I will be honest. I fail at this regularly. Not in the dramatic ways. I do not miss the big obvious gaps. I miss the ones that seemed too obvious to check.

We had a predictive modelling project recently. The ask was specific. It was in their budget, in their business plan,

they had a demo of a competitor's version. So we scoped it, they approved it, and we started.

Somewhere in the middle of it I realised they had no idea what predictive modelling actually was. Not a partial understanding. No understanding. Nobody had ever asked if they knew what it meant. Including me.

The ask was so specific, the signals so convincing, that I did not stop to ask the most basic question. Do you actually understand what this produces and what you would do with it? That question takes thirty seconds. Not asking it cost weeks. Thirty seconds.

The other version of this is subtler. Sometimes I will be in a conversation asking the same question ten different ways. At some point I have learned to just say it. I have either not had enough coffee or I am missing something, but I genuinely do not know what you are describing. Can we start from scratch?

I share these because I want to be clear about something. I know this material well. I have watched it work hundreds of times. And I still get it wrong, because getting it right is not about knowing the framework. It is about having the discipline to run it every single time, even at ten hours into the day, even when you are certain you already understand. The gap does not care how experienced you are. It opens the moment you assume it is closed.

What I am still figuring out

My biggest challenge with context and communication is my own ego and impatience. I slip into exactly the mindset I am telling you to avoid. Just trust me, why do I need to explain this, you know I can do the work. I get tired. I get bored of

certain projects. I have a bad night with the kids and my care factor drops. I know better so I will just do it my way. And then, surprisingly, I get it wrong. Miss something. Build the wrong thing. It is almost like ignoring the context step produces bad outcomes. Who knew.

The honest reason is time. I do not want to spend the extra hour in meetings. I want to spend that hour doing the work. I am ten hours into the day, the kids want to play, I promised a project would be done. So I skip the conversation and go straight to execution. It is not an excuse. It is just what happens. What I try to do when I catch it, usually once the day is done and I have had time to step back, is fire off a quick message. Not a formal apology, not a full debrief. Just an acknowledgment that I pushed ahead without the conversation I should have had, and an offer to talk it through the next day. It is not perfect. It is better than pretending it did not happen.

The other thing I genuinely have not solved is calibration. There is no clean answer to how much context work is enough. I can be cruising along, light touch, feeling like the client is completely on the same page, and then get a question that makes it obvious they are not. Equally I can feel like I am over-explaining, dumbing things down, wasting their time. The approach I have landed on is to say it directly. In this role I am biased toward the technical, call it out if you do not understand something, tell me if you want me to slow down, and more importantly tell me if you want me to hurry up. This is about getting the work done, not making me feel good. That honesty tends to work better than any amount of calibration I could do on my own.

The hardest version of this is when you deliver something genuinely good, built for a fraction of what a competitor would charge, in a fraction of the time, and the response is

why was it so expensive, why did it take so long, why does it not have this feature. The instinct is to delete everything, tell them to go spend the money elsewhere, and come back in a year. I have felt that instinct more times than I would like to admit.

What I try to remember in those moments is that the frustration on their side is not personal. What they could afford placed real pressure on them. They stuck their neck out internally to make it happen. And they probably did not fully understand some of the gaps until they could see the finished thing. It is easy to negatively frame every difficult interaction. But people are mostly good and mostly just trying to do their best so they can get home to their kids, see their friends, keep their job. Sometimes remembering that is the only context work that actually matters.

I lay all of this out, the wins and the ways I still get it wrong, so you can see the shape of the thing clearly. This is the first lever, and it is the one where the gap first opens. Get it right and you have removed most of the reason products stall. Get it wrong and no amount of talent downstream will save you, because the whole team will be running hard at the wrong target. But context on its own is not enough. Once everyone actually understands what you are building, the next question is how the work gets broken down so it ships instead of drifting. That is where we go next.

Before you move on

- Think about a project that went sideways. At what point did the context gap actually open, and when did you first notice it?

- What is a brief you have recently given or received that felt complete at the time but turned out not to be? What was missing?
- What is the equivalent of the systems versus features question in your current work, the thing that keeps getting built around instead of solved?

14. Work Structure

> *Every time you step in to fix something yourself, you make the next problem more likely.*

There is a version of this chapter that gives you a list of project management best practices. Sprint cadences, ticket formats, definition of done, velocity tracking. That version exists already, in a hundred books and twice as many consulting decks, and if that is what you need you can find it easily enough. This is not that version.

What I want to talk about is the harder thing underneath all of those practices. The thing that decides whether any of them actually work. Because the reason effective work structure is so difficult on a technical build is not that the frameworks are complicated. It is that your role inside those frameworks is genuinely hard to get right, and when it goes wrong, it goes wrong in ways you almost never see. This holds whether the people building your product are a partner like Onwards Labs, a contractor, or the one or two people you bring on yourself. The structure is the same. The scale is smaller than an agency, and that is a mercy, not a handicap.

That last part is worth pausing on. Most ways a project can fail announce themselves. A deadline gets missed. A budget runs out. Someone quits. Bad work structure is quieter than that. It degrades the output slowly, from the inside, while every surface signal still looks fine. The meetings happen. The updates get written. The invoices get paid. And the quality drifts down a little each week until one day you

look up and the product is stuck, and you cannot point at the moment it went wrong, because there was no moment. There was a hundred small ones.

Your job is not to manage the work. It is to lead the system.

When you bring in the people who will build your product you are, by definition, bringing in people who know more than you about their specific domain. That is the point. Whether it is one contractor, a build partner, or a couple of hires, the moment you start directing how they do that work, the specific technical decisions, the implementation approach, the architecture choices, you are not leading anymore. You are getting in the way of the capability you paid to access.

Your role has three parts that are distinctly yours, that nobody else in the team can do as well as you. The first is context. You are the person who understands why the work matters, what it needs to achieve, and how it connects to everything else in the business. The second is application. You are the one who takes technical information and turns it into business decisions. When a developer tells you that cutting the load time in half would take a month of engineering work, the business part of that decision, whether a month of engineering time is the right investment right now, is yours. The third is the non-tangible decisions. Which features get built in which order. Whether the team should slow down and address technical debt or push through to a deadline. These are context problems, not technical ones.

This is the heart of being the non-technical person directing technical work. You are not the best engineer in the room. You are not supposed to be. You are the person who

owns context, application, and direction, and who is disciplined enough not to reach for the keyboard.

I want to name how counterintuitive this is, because it is the part people get wrong even after they have nodded along to it. Every instinct you built as an expert in your own field tells you to lean in when something matters. Roll up your sleeves. Get into the detail. Own it. In almost every other part of your business, that instinct is correct, and it is probably part of why you got this far. In technical work it is the exact thing that breaks the system. The discipline the job asks for is the opposite of the discipline that got you here, which is why so few people manage it, and why the ones who do are worth so much.

Micromanagement in technical teams is particularly destructive

In most work environments, micromanagement is inefficient and demoralising. In technical teams it is that and something more specific. It actively destroys the conditions required for good technical work to happen.

Good technical work runs on clear ownership, genuine contribution, and a real sense that the people doing the work have a say in how it is done. When a developer owns a system, really owns it, with the autonomy to make real decisions about how it is built, they think about it differently than when they are executing someone else's specification. They catch problems earlier. They make better trade-offs. They care about the outcome in a way that produces quality that cannot be mandated from the outside.

Micromanagement removes ownership without you realising it is happening. Every time you weigh in on a

technical decision that should sit with the person building the thing, you are sending a signal that their judgment is subject to your override. Over time that signal accumulates into someone who stops exercising judgment independently, because they have learned, through experience, that it will be second-guessed anyway.

And here is the trap. The damage is invisible while it happens and only visible once it is done. You do not feel the ownership draining out of the person. You feel like you are being a diligent leader, staying close to the work, catching things. What you are actually doing is teaching a capable person to wait for you. By the time you notice, the builder you had has quietly become someone who executes tickets and flags nothing, and rebuilding the earlier version is far harder than it was to preserve it.

The honest difficulty

Getting the balance right between involvement and delegation is genuinely hard on technical work, and it is harder than it is in other domains. The reason is that the work is less visible. In a sales pipeline you can see the numbers. In an operation you can see the throughput. In technical work, the most important part, the thinking, the design decisions, the architectural choices, is largely invisible until it surfaces as either a working system or a problem.

That invisibility makes it difficult to know when to step in and when to stay out. And when it is not clear, the default for most people is to step in. To review, to check, to ask for updates, because the alternative feels like flying blind. I raise this not because I have a clean solution for it but because naming it honestly is more useful than pretending the frameworks make it simple. They do not. Getting this right

takes active effort and deliberate attention. The question I have to keep asking myself is am I involved in this because my involvement is genuinely necessary, or because it makes me feel less exposed. Most of the time the honest answer is the second one, and learning to sit with that discomfort instead of resolving it by interfering is most of the job.

Start with the person, not the process

The first thing I do when setting up how a project runs has nothing to do with task tracking or sprint cadence. It is a conversation about how we are going to work together as people. Whether that is a build partner, a contractor, or someone I have just brought on, I want to know their background. What they are trying to get out of this engagement beyond the obvious. What else is going on in their life. I also make the rules of how we work explicit from the start. I will never send something urgent without first asking if that is okay. We do not accept rudeness from clients.

The communication stack I use. Scheduled recurring meetings, weekly or fortnightly, for brainstorming, deep dives, and personal check-ins. Screen capture videos for anything that needs context but not a live conversation. A five to ten minute video explaining a problem is almost always faster than a meeting, and it produces a transcript that can go straight into a GPT for further brainstorming. A chat tool for coordination. A phone call for genuinely urgent matters, which we try to make rare.

On goals, I work from long to short. Start with the big goal. What does success look like at the end of this phase? Then the near-term goal. What needs to be true in the next two weeks? Then the stretch goal. What does better than expected look like? The person doing the work generates the

tasks underneath those goals themselves. Not because I do not have opinions, but because the act of generating the tasks is how I verify that they have the same picture of the goal that I do. This whole process should take five minutes. It is a quick alignment check, not a planning session.

Read that last part again, because it is doing more work than it looks like. I am not having them write the tasks to save myself the effort. I am having them write the tasks because it is the cheapest, fastest test in existence for whether the context landed. If the tasks they generate match the picture in my head, we are aligned and I can step back. If they do not, I have caught the gap in five minutes instead of five weeks. It ties directly back to the last chapter. Work structure done well is just context, verified out loud, before anyone starts building. It works exactly the same whether the person on the other end of that conversation is your build partner or the first person you hired.

What this takes in practice

The honest version of my work structure practice is that I front-load the relationship and the clarity, and then try to stay out of the way. The first conversation with anyone new is not about the work. It is about them. What they care about, what they want to get out of this, what I can do that is not just paying them.

I also make my own role explicit early. I tell them directly. You are better than me at your specific thing, I hired you because of that, and I want to hear it when you think I am wrong. My job is to give you context and direction, not to tell you how to do your work. If I start doing that, call me on it.

The thing I actively monitor is whether the people building the thing still feel genuine ownership of their work, or whether they have quietly slipped into execution mode, waiting for direction rather than generating it. The signal is usually in the quality of the questions they ask. When I notice the shift I usually find that something upstream has changed, a deadline got tighter, I got more involved than I should have, and I need to step back and reestablish the conditions that made the ownership possible in the first place. Noticing that shift early, from the texture of someone's questions, is not a skill you can write into a process document. It comes from having watched enough builds drift to feel it starting.

A worked example, the client that nearly broke the team

Early in one of my businesses I had a client who was under significant pressure from their own organisation to deliver results quickly. That pressure came down the chain and landed on my team, in the form of late night messages, unreasonable deadlines set without consultation, and a communication style toward my offshore team members that I can only describe as contemptuous.

My team said nothing to me about it directly. They absorbed it. They worked harder and stayed later and said yes to things they should have pushed back on. I found out not because anyone complained but because I noticed the quality of their work starting to drop and the energy in our team communication starting to feel flat in a way it had not been before.

When I dug into it the picture was clear. The client had bypassed the working norms we had set at the start of the

project. The escalation path, which should have gone through me, had been ignored. I ended the engagement. Not immediately and not without a conversation, but within two weeks of understanding what had been happening. I was explicit with my team about why. Not to make a point, but because they needed to see that the rules we had set at the start were real. The team response was immediate and visible. The energy came back. The quality came back.

The lesson buried in that story is not be nice to the people who work for you. It is that the working norms you set at the start are load-bearing, and the only thing that makes them real is whether you enforce them when it costs you money to do so. That is true whether you are protecting a team, a single contractor, or the partner building your product. Ending a paying engagement to protect a norm is expensive in the moment and cheap over the life of the relationship. Getting that trade right, again and again, under real financial pressure, is the actual work.

What it looks like when it goes wrong

I do not have many catastrophic work structure failures to share. What I have is a thousand small ones that add up to the same thing.

Not being across team dynamics until something surfaces that has been brewing for weeks. Not double-checking context on a handoff. Getting looped out of a client conversation because someone handled it directly and did not think to flag it, not out of bad intent, just because the escalation path was not clear.

That last one is the most common. I do not want to be the bottleneck. My team does not want to bother me. So both of

us make a completely reasonable individual decision and collectively create a mess. Every time.

The fix I keep coming back to is cadence over intervention. Not checking in when something feels wrong, checking in on a rhythm so things do not get far enough wrong to feel that way. Short, regular, low stakes. Not glamorous. Just the thing that works. And I want to be clear that it is not glamorous on purpose. The reason this is hard is not that the fix is clever. It is that the fix is boring, and boring things are the first to get dropped when you are ten hours into a day and a fire is burning somewhere else. Holding a dull rhythm in place, week after week, when nothing is visibly wrong, is a discipline. It is the opposite of the heroic intervention that feels like leadership and usually is not.

What I am still figuring out

The chapter on work structure talks about staying out of the way, and there is one specific place where I take that too literally: seniority. I go quiet in front of people who are clearly skilled, exactly when I should be pushing hardest, because the doubt that creeps in when you are about to challenge an expert is genuinely hard to override. It is not a general reluctance to have hard conversations. It is a very particular blind spot that switches on the more impressive the person across the table is.

A few years ago I was working with a very senior data lead on a critical system. Extremely well paid, widely respected, the kind of person who had accumulated enough seniority that almost nobody questioned them directly. He had fundamental flaws in his process that were creating significant problems for every other team that touched the system. Ninety-nine percent of the issue was ego. He knew

better than everyone, went extremely technical to overpower other teams in any room, and was simply wrong about several things that were costing the business a significant amount of money.

And I let it sit for too long. Because the doubt crept in. Do I really know this better? What if I am missing something? Surely someone this senior has a reason for this approach that I am not seeing. That doubt is not irrational. It is actually the right instinct when applied correctly. The problem is that I used it as a reason to wait rather than a reason to investigate. By the time the situation was undeniable, the cost was already significant and the relationship had deteriorated in ways that made the conversation harder than it needed to be.

I see a smaller version of this constantly with developers, designers, anyone who is better than me in their specific domain. The balance between giving a high-level ask and letting them add their professional judgment, versus spelling out exactly what I want and getting precisely that but not what I actually needed, I find that line genuinely difficult to draw. Too vague and I get something technically impressive that misses the point. Too specific and I get exactly what I asked for, which is also not quite right, because I am not the expert and my specification had gaps I did not know about.

The thing I keep coming back to is this. Be clear about where you sit and where the other person sits on the technical to business scale. Say it out loud at the start. Then drop the ego on both sides and back yourself when the evidence supports it. When I do not back myself and it costs the project, I own that it is on me, not on the person I deferred to. My system, my call, my responsibility. That accountability is the thing that makes me do it differently next time rather than look for someone else to blame.

I put all of this on the table, the parts I have solved and the parts I plainly have not, because I want you to see the truth of it. There is no version of this where you install a process and the judgment takes care of itself. The frameworks are the easy ten percent. The hard ninety is holding the line between too involved and too absent, week after week, project after project, with real money and real relationships on the line and no clean signal telling you which way to lean. It is learnable. It is not quick, and it is not a checklist. Which brings us to the next question, and in some ways the most important one. None of this structure matters if the wrong people are inside it. So how do you judge whether someone is any good, whether that is a build partner you are choosing or a person you are bringing on, when you are not technical enough to check their work yourself? That is where we go next.

Before you move on

- Where in your current work do you step in when you probably should not? What is that costing the person you are stepping in for?
- What are the decisions your team is currently waiting on you for that they could make themselves with a clearer brief?
- If you had to describe your role in one sentence to a new team member, not your title, your actual function, what would you say?

15. The Right People

> *The best hiring signal you will ever get has nothing to do with the CV.*

I want to be upfront about something before this chapter starts. My approach to choosing people works well for me and I am happy to share it in full, but it is not a system I would prescribe universally. Roughly one in ten does not work out. Most of what follows is drawn from years of hiring, because that is where I learned it, but the judgment underneath it is the same judgment you use to pick a build partner or a single contractor. You are reading someone in a short conversation and backing a call you cannot fully justify. Whether the paperwork says employee or vendor changes very little about that. I mention the failure rate not to undermine what follows, but because honesty about the miss rate of any approach to choosing people is more useful than presenting it as a solved problem. It is not. Anyone who tells you they have hiring figured out, that they have a process that reliably screens out the misses, is either lying or has not run it enough times to hit the misses yet. The skill is not eliminating the failure rate. It is keeping the cost of each failure small enough that the model still works.

The CV is pass or fail. Nothing more.

Most hiring processes treat the CV as the primary filter. I use the CV differently. It is a binary check. Do they have semi-

relevant experience in the general area of what I need? Yes or no. If yes, they can probably do the job. If no, we are done. That is the entire function of the CV in my process.

The reason is simple. Technical skills in most domains change faster than any CV can reflect. The specific tools, languages, platforms, and frameworks that matter today are different from the ones that mattered two years ago. A CV that looks perfectly matched to today's requirements is a snapshot of what someone has already done, not a signal of what they are capable of learning or how fast they will move when the problem is genuinely interesting to them. The CV is a history document. You are hiring for now.

This is exactly where non-technical experts get stuck, and it is worth naming plainly. When you cannot evaluate the work yourself, the CV, or the portfolio, or the case study deck a build partner sends you, feels like the one solid thing you can hold on to. It is written down. It has dates and companies and skills you can match against what you think you need. So you lean on it, hard, because it is the only part of the process that feels legible to you. And it is the least reliable signal in the whole thing. The trap is not that people value CVs too much in the abstract. It is that a CV is the thing a non-technical person can read, so it becomes the thing they judge on, precisely because everything that actually matters is invisible to them.

Effort and enthusiasm are the signal

The hires that have worked best for me share a specific characteristic that has nothing to do with their credentials. They are the ones who were still thinking about the problem after the conversation ended. The ones who sent a quick message saying they had a few more ideas. The ones who saw

a job ad go out and reached out directly, not with a polished cover letter but with actual thoughts about the problem the role was trying to solve.

The interview itself is short and informal. Fifteen minutes. I bring a real problem I am currently working on, not a hypothetical, not a case study, a real thing I am actually trying to figure out, and I talk about it with them. If the conversation is fun, if they lean in, if they ask good questions and start building on ideas in real time, they are the one.

Here is the part that makes this hard to copy, and I want to be honest about it rather than pretend it is a repeatable trick. Judging that conversation well depends entirely on you being able to tell the difference between someone who is genuinely engaging with the problem and someone who is performing engagement. Those look nearly identical for the first ten minutes. The difference shows up in whether they build on an idea in a direction you did not feed them, whether they push back on something you said, whether they ask a question that reveals they have already run three steps ahead of the conversation. Reading that difference is a skill, and it is one you sharpen by getting it wrong. Which is a polite way of saying you will misread it, and the one in ten is where you find out you did.

Minimise the downside on both sides

One thing I am deliberate about. I do my best to avoid requiring major commitments on either side before we have established that the fit is real. Locking into a long contract, or asking someone to clear their plate for you, and then discovering after two weeks that it is not the right match, is a much bigger ask than structuring the first stretch of work so it is small, contained, and real. If it does not work out, nobody is

worse off. The same logic applies when you are choosing a build partner. Start with a small, paid, real piece of work before you hand over the whole product.

This is not a nicety. It is the single most important decision in the whole process, and it is the one people skip because it is administrative rather than exciting. The structure of the engagement, not the interview, is what determines how much a wrong choice costs you. Get the structure right and a miss costs you two weeks. Get it wrong and a miss costs you six months, a legal process, and a relationship that curdles into something ugly. More on that shortly, because it is worse than it sounds.

Values alignment is the thing nobody writes in the job description

Beyond the enthusiasm and the personality fit, there is something less tangible that I have come to think of as the most important filter of all. Do they hold themselves to the same standard I do? Do they say something when they think something is wrong, or do they nod and quietly do what they were told? Do they care whether the outcome is actually good, or just whether their contribution to it is defensible?

High performing teams have this in common. Not identical personalities. Some of the best teams I have been part of have had real friction and genuine disagreement. But a shared standard for what good looks like, a shared commitment to saying the thing that needs to be said rather than the thing that is comfortable. This is the hardest thing to test for in a fifteen minute conversation, because everyone claims it. Everyone says they speak up. The only real test is watching what someone does when they disagree with you in

the room, in the moment, which is exactly why I put a real unsolved problem in front of them. A real problem gives them something honest to disagree with.

On specialists, and why you probably do not need one yet

Hyperspecialists excel at the final one percent of a problem. The optimisation, the edge case handling, the marginal gain that matters when everything else is already working exceptionally well. But you need to do the ninety-nine percent first. And the ninety-nine percent is almost never something that requires a specialist. It requires someone curious, capable, and genuinely invested in figuring it out.

Founders reach for the specialist too early because the specialist feels safer. Their expertise is legible in a way the generalist's is not. A twenty-year expert in a narrow field is easy to justify. But you are usually paying premium rates for someone to solve a problem you do not have yet, while the ninety-nine percent of the work that actually needs doing sits untouched. The person who can do that ninety-nine percent well, the curious generalist who gets underneath the problem, is harder to identify and far more valuable at the stage most people are actually at. Knowing which one you need, and resisting the pull toward the impressive-sounding specialist, is itself part of the skill.

What this takes in practice

Before I post a job ad or reach out to anyone, I try to understand what I am actually hiring for. Not the job title, the specific capability, what it looks like in practice, and where

the hard parts are. Fiverr is genuinely useful for this. When I am entering a domain I do not know well, I browse services the way you would browse a market. Not to find the cheapest option, but to understand what the capability landscape looks like. What skills are being offered, what tools keep appearing, what combinations seem popular, what the adjacent services are that tell you something about where the real complexity sits.

Once I know what I am looking for I run things in parallel rather than one after another. I ask people I trust if they know anyone. I post a LinkedIn ad. I test a specific capability on Fiverr with a small paid engagement. The ad is deliberately specific. I describe what I am building, where I am stuck, and what my goal for the engagement is. I put my email on it. The people who respond with genuine enthusiasm about the specific problem are the ones I talk to first. You do not need a network of leads to do a smaller version of this. One clear ad and one small paid test is enough to start.

I then offer a short, contained piece of paid work, structured so nobody has to over-commit up front. We work together for two weeks, part time, on something real. Not a test task, actual work toward an actual goal. Worst case they made some money and I learned something. My maximum exposure is two weeks of part-time pay and some time. If you are weighing up a build partner instead of a person, this is the same move. Give them one real slice of the product first and judge the result, not the pitch.

What I personally do

All I ever wanted when I was starting out was for someone to give me a shot. That is still what drives me when I am looking for people. I am trying to find the person who just needs a

shot. The one who is so genuinely excited about the problem that the enthusiasm is hard to fake.

The hiring method itself matters less than people think. Every framework, twelve-step interviews, personality assessments, structured scorecards, is ultimately just trying to protect against the downside. They all fail at roughly the same rate. The difference is not in the sophistication of the process. It is in whether you back your judgment when you feel it, and whether you protect against the downside structurally rather than procedurally. Back your judgment. Keep the exposure small. Move on cleanly when it is not right.

I want to sit on that for a second, because it is the quiet claim underneath this whole chapter and it is easy to skate past. All those elaborate hiring systems fail at about the same rate as a fifteen minute conversation and a two week trial. That is not an argument for being careless. It is an argument that the elaborate process is buying you a feeling of safety, not actual safety, and the actual safety comes from somewhere else entirely, from the structure of the engagement rather than the rigour of the screen. Most people invest in the wrong one because the wrong one feels more responsible.

A worked example, the iOS developer

My plan was to use a vibe coding tool and generate a mobile app that used some vision modelling work I was running for another client. I went on Fiverr to understand the capability landscape first. Lovable and Capacitor kept appearing. I had no idea what Capacitor was, but the fact that it appeared constantly told me there was a reason. It turned out to be the library that bridges a web app to native iOS functionality, and the iOS submission process at the end of it is genuinely

painful and full of rules you only find out by submitting and waiting for it to fail.

I posted a LinkedIn ad describing exactly what I was building, where I was stuck, and what my goal was. A guy reached out who was genuinely pumped about the idea. Passionate about vibe coding, had built things himself, exactly the energy I look for. We ran a two-week trial, part time. He was amazing in the first chats but he really did not understand the work. Was not progressing. Not responding. I had a direct conversation, set a clear goal, gave it the two weeks. He did not get close to it. We finished up at the end of the fortnight. I paid him in full, said thanks, and that was it. No legal complexity, no hard feelings. The structure made the exit clean.

The harder version happened earlier, when I was running more traditional salaried roles. Six month probation periods, genuine investment in coaching and correction, trying hard to make it work. The longer you invest in trying to correct something that is not working, the more it compounds. I had situations that escalated legally, offshore in particular, where the laws are different, your onshore lead can be exposed, and the person on the other side may have quit a stable job to take the role and is now in a genuinely desperate position. The structure of the engagement is the most important risk management decision you make in hiring. Not the interview process. The structure.

Put those two versions side by side, because that contrast is the entire lesson. Same instinct, same kind of miss, wildly different cost. The iOS developer cost me two weeks of part-time pay and a friendly goodbye. The salaried version cost months, legal exposure, and real human damage on both sides. Nothing about my judgment changed between them. The structure changed. That is the whole game.

What it looks like when it goes wrong

The honest version of my hiring failures is this. I almost always know in the first meeting.

The hires that did not work out? There was usually a moment in the first conversation where something did not quite land. Something I noticed and then immediately talked myself out of, because I wanted it to work.

The pattern when it goes wrong is consistent. They say the right things. They need the work. A few days in there is nothing. No output, just updates and explanations. The explanations are good. Genuinely good. Which makes the whole thing worse.

What I have changed is the structure. The two-week trial means backing someone costs me two weeks if it does not work, not six months. What I have not solved is the moment before, where I know something is off and proceed anyway, because I want to be the person who gives people a shot.

What I am still figuring out

Early in my career I found it extremely difficult to get anyone to give me a chance. All I wanted was someone to back me and let me learn. The experience of applying to everything, getting knocked back, being ignored on reach-outs, never quite making the cut because I did not have the exact speciality they were looking for, that was genuinely defeating. It left a mark.

I have a very strong drive to offer that chance to others. When I find someone in the exact same position, driven, passionate, capable, but needing someone to back them, backing them is one of the most rewarding things I do. One of the best developers I have ever worked with had never been a

developer. He had done some low-code work, was learning on his own, and sent me an email that was just so genuinely keen it was hard to ignore. I gave him a call. Zero ego, super smart, wanted this badly. I gave him the chance and he was extraordinary.

The thing I have not fully solved is that this instinct can be exploited by people who are very good at identifying it. The story that matches what I am looking for, the enthusiasm that mirrors the signal I am responding to, the presentation of someone who just needs a shot. Some people are genuinely that person and some people have learned that presenting as that person works on people like me. I am normally good at spotting the difference. Not always.

When I get it wrong it is really not nice. The pattern when it goes badly is consistent. The work does not materialise, the story expands to explain why, and when the engagement ends it shifts quickly from personal hardship, cannot afford rent, it will hurt the family, I have sacrificed so much for this, to anger, constant contact, legal threats, and the kind of sustained pressure that is designed to make you pay more to make it stop rather than because you owe anything.

I still back people. I will keep backing people. The one in ten failure rate is just part of the model, and the nine that work out, including that developer who had never written production code before I hired him, are worth it. What I have changed is the structure. The two-week trial is not just a hiring tool. It is the thing that means backing someone costs me two weeks of part-time pay if it does not work, rather than six months of salary, a legal process, and the kind of sustained stress that bleeds into everything else.

What I have not figured out is a reliable way to tell the difference before the trial. The signals I look for, genuine enthusiasm, specific questions, a willingness to push back,

can all be replicated by someone who has learned to replicate them. I do not have a clean answer to that. What I have is a structure that limits the damage when I am wrong and a track record that tells me the instinct is right more often than it is not.

I am telling you the parts I have not solved because I want you to understand what judging people actually involves, especially when you cannot judge the work itself. This is not a matter of learning a better set of interview questions. It is reading a person in fifteen minutes, backing a feeling you cannot fully justify, and being wrong one time in ten, then building the whole thing so that being wrong one time in ten does not sink you. That is a real skill, developed over a lot of hires and a lot of misses, and it does not compress into a checklist no matter how much anyone wants it to.

The credential is not the capability

I have more formal education than most people in my industry. A BSc in psychology. A Master's in marketing. An MBA. A year of medicine. A started PhD. My honest assessment is that the credential and the capability are almost entirely separate things.

What the degrees actually gave me. The experience of doing something hard and finishing it. Exposure to ways of thinking I would not have encountered otherwise. A broad base of context that occasionally surfaces in useful ways. That is real value. It is just not what most people think they are buying when they enrol.

I have a Master's in marketing. Someone who spent one focused month learning digital marketing would outperform me in that domain. Not because the degree was bad, because

the world moved and the degree did not. The shelf life of formal education in fast-moving fields is shorter than the time it takes to complete the course.

The access problem is the one nobody talks about honestly. Formal education requires money or debt, time out of the workforce, and a support structure that is simply not available to everyone. When I look at the people I have hired who did not go that route and outperformed people who did, the credential starts to look less like a signal of capability and more like a signal of access. Those are not the same thing.

The CS degree version of this is worth addressing specifically. CS degrees were never designed to produce job-ready developers. They were designed to give you a broad enough theoretical foundation that you could tackle problems you had not seen before. That groundwork is genuinely valuable. It is increasingly available outside a four-year program, at a fraction of the cost and time, with practical application built in rather than added on. AI accelerates this further. The barrier to building something real has dropped to the point where a genuinely curious person with six months of focused effort can produce work that would have required years of formal training five years ago.

What I look for when I hire has almost nothing to do with formal credentials. It is evidence of genuine interest pursued independently. Something built. Something figured out. Something hard that did not have to be done but was done anyway. That signal is rarer than a degree and worth more than any of them. And here is the honest catch for you as a non-technical founder. That signal is also much harder to read than a degree. A degree you can verify in a line. Evidence of self-driven capability you have to actually assess, which means you are back to needing judgment you may not have built yet. This is the whole difficulty of the chapter in one

sentence. The thing that actually predicts a good hire is the thing hardest to see from where you are standing.

Across these three levers, context, structure, and people, the same pattern keeps surfacing. The frameworks are simple to state and genuinely hard to run. The value was never in knowing that context matters, or that you should not micromanage, or that the CV is a weak signal. Everybody knows those things. The value is in the judgment it takes to apply them under real pressure, with real money on the line, at ten hours into the day, one time in ten getting it wrong and being structured so that it does not matter. That judgment is not a thing you download. It is a thing you accumulate, over hundreds of builds and hundreds of hires, until the patterns are so familiar you feel the mistake coming before it arrives. That is the honest picture of what doing this properly takes. What you do with that picture, whether you spend the years building that judgment yourself or bring in someone who already has, is the decision the rest of this book is here to help you make.

Before you move on

- Think about the best person you have ever worked with. What made them exceptional, and how much of that showed up on their CV?
- Think about a hire you made that did not work out. Looking back, what was the real signal you missed or ignored?
- What would a two-week trial engagement look like for the next person you need to bring in? What would you need to see from them to know it was working?

16. Location Arbitrage

> *The reason most people get offshore wrong has nothing to do with the people they hire.*

Let me be straight about this one before we get into it. Location is a lever, a powerful one when it works, but it is not a guaranteed value case and I would be doing you a disservice to present it as one. Some of the best people I have ever worked with are offshore. I have also had some genuinely bad experiences. Both things are true, and both are worth understanding before you decide how much weight to put on this lever.

Here is why this matters to you even if you never intend to hire anyone overseas yourself. The talent that builds your product does not need to be local, and understanding that changes what your options actually are. If you go the build-partner route, Onwards Labs already runs this model, the distributed team, the lead relationships, the legal layer, so what you are really doing is judging whether the people behind the badge are good, which is the last chapter. If you hire directly, you inherit the whole thing yourself. Either way, the reason to read this chapter is to see clearly what sits behind the words “offshore” and “distributed”, so nobody can sell you a bad version of it and so you do not rule out a good one on reputation alone.

I want to spend this chapter showing you what a distributed build really looks like from the inside. Not the pitch version. The version with the legal exposure, the failure

modes, the years it took me to get the one relationship right that makes the whole thing hold together. By the end you will understand the model well enough to see why it works when it works. You will also understand why so few people manage to run it themselves, which is exactly why most experts building one product are better off partnering with someone who already has, and why the people who point at a bad experience as proof that offshore does not work are almost always pointing at a process they got wrong, not a geography.

Offshore is not always cheaper by rate

The first thing to unlearn is that this is about rate. It is not, or not mainly.

What you are buying is access to a much larger talent pool, the ability to find the specific person you want rather than settling for whoever is available in your city, and a flexibility in how you structure the engagement that is close to impossible to replicate onshore. A developer who works ten focused hours a week for you, and earns well for those ten hours, will outperform a full-time local hire who is productive for maybe three of their eight daily hours. You are not buying cheaper time. You are buying a higher proportion of actual impact time.

The most underrated version of this is the senior specialist for whom your project is a second engagement. Someone already operating at a high level, who runs hard for you inside a contained, well-defined scope, because the work is interesting and the arrangement fits their life. That person is gold. They are also hard to find, hard to structure correctly, and easy to lose if you treat the relationship carelessly. Everything difficult about this lever lives in that sentence.

The legal and commercial reality is the hard part

Here is the piece almost everyone who writes about offshore hiring skips, because it is unglamorous and complicated and changes country to country. It is also the part that will hurt you if you get it wrong.

To engage someone offshore in a compliant way you need either a legal or payroll presence in their country, or you work with someone who already has that structure built. Most people starting out have neither. The people who do have it know they have leverage, and they price accordingly.

The liability piece is rarely discussed honestly, so I will. In most offshore arrangements the legal owner of the employment contract is your local lead, not you. If you stop paying, they typically have limited recourse against you but significant exposure to the people they contracted on your behalf. Three to six months of someone's salary is real money, and it is sitting on your lead's shoulders, not yours. Once you understand that, you understand why good leads are hard to find, why the ones who exist are cautious about who they work with, and why you cannot treat this like posting a job ad and picking the cheapest bid. You are asking someone to carry legal and financial risk in their own country on the strength of your word. That is not a transaction. It is a relationship, and it takes years to earn.

I am walking you through this on purpose. Not to scare you off it. To show you that the thing standing between a founder and a working distributed team is not the developers. It is this layer, the legal, cross-border, tax-and-liability layer that nobody puts in the brochure, and that takes real time and real scar tissue to get right.

The lead is everything

If there is one thing I would want you to take from this chapter, it is this. The lead is the single most important factor in whether the distributed model works. Not the rate. Not the platform you find people on. Not the contract structure. The lead. If you partner with someone to build your product, this is the person you are really relying on inside their organisation, and it is worth knowing they exist and what they do. If you ever run it yourself, finding this person is your whole job.

Finding the right lead has taken me years and several painful experiences to get right, and I want to be honest that "years" is not a figure of speech. What I am looking for is someone who acts as a genuine partner in that location. Someone who takes pride in running the team well, who is accountable for outcomes rather than activity, and who has enough standing in their local market to be useful when things go wrong. Because things will go wrong.

The model I have landed on is a senior lead who manages the legal and contractual side, can recommend people directly to cut down search time, and is first in line when any issue arises. I keep this person genuinely happy and engaged, not only financially, but in the relationship and the sense that they are a real partner. In return they refer their best people to me. They manage the local complexity I cannot manage from the outside. They stand in front of the problems before those problems reach me.

I am also transparent about the economics. On fixed-cost client work the general structure is around forty percent going to the team, covering their rate, their taxes, and any overhead the lead carries. On some projects a team member has earned far more, a hundred and fifty dollars an hour where the work

warranted it. Transparent ownership of the numbers, shared wins when they come. That transparency is not generosity for its own sake. It is what keeps a lead loyal when a competitor offers them a slightly better cut, and loyalty in this layer is worth more than a few points of margin.

You could read all of that and think, fine, I will find a great lead. Sit with how hard that sentence is. You are looking for a person, in a country you do not live in, who will carry legal risk for you, run a team to your standard without you in the room, tell you the truth when something is failing, and stay when someone offers them more. I have found people like that. It took a long time, more than one relationship that did not survive, and a way of treating people that I had to learn by getting it wrong first. This is the honest reason most experts building a single product should not try to assemble this from scratch. The whole point of a build partner is that this layer is already built and already trusted, so you get the output without spending years earning the relationship that produces it.

The real risks, and they are real

The bad experiences I have had, and the ones I have heard about from others, cluster around a few specific failure modes. Recognise them, because they are the reason offshore has the reputation it has.

The bait and switch, where you assess an impressive portfolio and the actual work is done by someone who clearly did not produce that portfolio. High turnover with no control over your resources, because a middle layer owns the relationship and shuffles people in and out. Agencies that squeeze margin at every point until the quality is the first

thing to go. And the one I find genuinely troubling, the use of unpaid or badly underpaid interns to do paid client work.

The direct hiring model through a trusted lead is the most reliable protection against all of these. Notice what that means, though. The protection is the exact thing that is hardest to build. The whole reason people fall into the bait-and-switch and the middle-layer churn is that they went looking for a shortcut around the relationship layer, and the shortcut is where all the failure lives.

The three locations, and what each one costs you

My current setup runs across three main locations, and I will tell you what is real about each, including the parts that are inconvenient.

The Philippines. Direct hire with very low tax overhead, seriously good people, and a lot of specialists to be found. Some bad practices exist in offshoring there, which cuts both ways. It means there is a real opportunity to recruit exceptional people by simply treating them properly, and it means you are operating in a market where being treated badly is normal enough that trust has to be earned before anyone runs hard for you.

My technical team is split between Serbia and Bangladesh. Serbia has an outstanding talent pool, but very high tax rates and strict regulation that requires your local lead to take on significant risk. Read that again. The talent is there, and the price of accessing it is that someone has to shoulder regulatory exposure to make it legal. Bangladesh has dedicated tax incentives for offshore technical development work and is more internationally friendly, which is part of

why the biggest build I have ever run offshore was centred there.

None of that is a menu you order from. It is a map of trade-offs, and picking the wrong point on it, or picking the right point without the lead to make it work, is how good intentions turn into a stalled project and a lead who quietly stops returning your messages.

What I do with my people

I look after my people and I make sure they know it. Not in a performative way. In a practical one. I pay above the minimum, always. I do not make promises I cannot keep. I am honest about commercial uncertainty. If a project might run out of runway in three months I say so, rather than letting people assume a continuity that is not guaranteed.

That last point sounds small and it is not. The easy thing, the thing that keeps everyone comfortable in the short term, is to let people believe the work is more secure than it is. It buys you a happier team this month and a betrayed one the day the runway ends. Doing it the honest way costs you a harder conversation now and buys you a lead and a team who believe you the next time you tell them something. That trade, uncomfortable now for credible later, is the whole job in this layer, and most people cannot make themselves pay the near-term cost.

The Bangladesh build, in full

The biggest build I have run offshore was centred on a lead in Bangladesh that my existing contact introduced me to. He

was, simply, exceptional. The kind of person you build a team around rather than a team you slot a person into.

The project was a full enterprise software product. At peak the team was three developers, two designers, a project manager, and two data scientists, all found through the lead's network, all direct hires through his local structure. We worked side by side as a single team. No local layer and offshore layer. Just a team, operating the same way regardless of where anyone happened to be sitting.

The total cost came in at roughly five to ten percent of what an equivalent onshore team would have charged for the same scope. And here is the part I want you to hold onto, because it is the whole lesson of this chapter compressed into one point. The majority of that difference was not rate. It was structure. No meetings that existed to make people feel included. No time spent on work that looked like productivity and was not. A team of people running genuinely hard for a contained number of hours, because the engagement was built to make that possible. The result was a production-ready enterprise product built by a team I would put against any onshore equivalent for that category of work.

You cannot buy that outcome. You assemble it, over years, out of a lead you trust, a structure you have gotten wrong before and corrected, and a way of treating people that makes them want to run hard for you rather than merely show up. The five-to-ten percent number is what people quote at each other. The thing that produced it is not for sale on a platform.

The real question is not offshore or local

Here is something that surprises people. Nobody actually asks me whether they should go offshore. What they ask is, how do

I get this built? The offshore conversation comes later, as a consequence of that question, not as the starting point.

We had a client recently who wanted their project finished. The conversation was about what they needed, what kind of person could do it, and what their options were on cost and timeline. Offshore came up because it was the fastest path to the right skill set, not because it was the cheapest path to any skill set. That distinction is the whole thing.

The framing most people bring in is wrong from the start. Offshore is not the budget option for people who cannot afford to do it properly. Take budget out of the equation entirely and, in most cases, you can find the right people faster, get the right model set up, and deliver faster using global markets than local ones, because you are drawing from a much larger pool.

If the question is high output work, quickly, with flexibility, offshore is often the better answer regardless of cost. If the question is finding an extremely senior specialist with deep domain expertise in a narrow field, say a finance systems architect with twenty years of specific institutional experience, then local is probably better. Not because offshore cannot produce that person, but because you need one specific individual and the signal-to-noise is harder to manage at distance.

The framework itself is not complicated to state. What is the role? What does great look like? Where does that person most likely exist? Then go find them, locally, globally, or both. The framework is simple. Executing it well, across borders, with the legal layer and the lead layer and the cultural layer all handled at once, is the part that takes a career to learn.

The stigma, and where it comes from

Offshore carries a stigma that is almost entirely the product of people doing it badly. The common picture is that offshore means paying someone twenty dollars a week and hiring a hundred of them. That is not the model I am describing, and it is not the model that works.

The reality is a broad spectrum. Australia is a lower-cost option compared to parts of the US market. Every location has different price points, different talent concentrations, different strengths. What you are doing when you go global is refusing to lock yourself into your local city. That is not a fundamentally different model from hiring someone from a lower-cost part of your own country. It is the same thing at scale, with more variables to get right.

People use cheap offshore work as proof that offshore does not work, and then complain about the quality. If you paid a bottom-of-the-market rate to a local agency you would get the same result. The quality of the output is almost entirely a function of the quality of the process. How clearly you define what you need, how carefully you evaluate who you hire, how much care you put into the relationship. That does not change when you cross a border. You just have somewhere more exotic to point the finger.

The question I always come back to is this. If a developer works from home and I never meet him in person, but he lives ten minutes from me, how is that different to him being in another country? I have never found a good answer. The work is the same. The output is what matters. What is different is not the geography. It is everything I have been describing, the layer of legal, commercial, and cultural complexity that you have to hold on top of the work, and that most people quietly hope they can skip.

What actually is different, work culture

One thing is genuinely real and worth understanding before you start. Work culture varies, and it changes how you need to manage.

Some countries have strong hierarchical structures where it is culturally expected that you never challenge your superior. You sit quietly, you nod, and delivering nothing feels less confronting than being seen to step out of place. I have also seen a version that shows up as very public micromanagement, a team lead standing over people all day to demonstrate they are doing their job, because visible control is valued over actual output.

I find this less common in technical people, and less common in the people I hire specifically, but it takes time to undo. Someone who has spent years in an environment where speaking up is dangerous does not immediately trust that yours is different. You have to demonstrate it, consistently, over a long stretch, before the behaviour changes.

And this pattern does not map to geography the way people assume. I have seen it more often in certain US contexts than in Bangladesh. Serbia tends to run stricter hierarchies that take time to shift, but even there it applies maybe half the time, and less with younger people. The younger generation of technical workers grew up with a more global culture around work. Their reference points are international. Their instincts around collaboration and honesty are closer to what you are trying to build than most people expect.

The practical implication, whether this work is yours to do or your partner's to do, is that nobody should assume the cultural work is done because someone said yes in a conversation. It takes building an environment where

disagreement is welcomed out loud, where mistakes are discussed openly, and where the measure of a good contributor is output and honesty rather than deference. Then you wait. The people worth keeping find their footing in it. This is not a one-time setup step. It is a standing act of leadership performed for months before it takes, and it is invisible in any brochure that tries to sell you offshore as a shortcut. If you are choosing a partner, this is a fair thing to ask them how they handle. If you are running it yourself, it is on you.

What this looks like when it goes wrong

I have played the offshore model conservatively and have not had a catastrophic failure. The worst outcome I have hit is nothing, no meaningful work produced, which is containable if the structure is right.

The thing I struggle with is the handoff problem. I find it hard to hand off key client engagements to other people. I tell myself it is about quality. Mostly it is about control. I am telling you that because if I, having done this for years, still fight the instinct to hold on too tightly, you should assume the instinct will be at least as strong in you, and plan for it.

The honest truth is that offshore reduces to the same problem as all hiring. Finding good people and treating them well. The geography changes the logistics. It does not change the fundamentals.

What I am still figuring out

The regulations are a genuine problem, and I do not have a clean answer.

Every country you operate in has different employment law, different contractor rules, different tax obligations, different rules about what constitutes a permanent establishment. The rules change. New guidance gets issued. What was compliant eighteen months ago may not be now. Staying across all of it while running a business is not fully possible. It is a constant approximation.

There are startups trying to solve this. Employer of Record services that handle compliance in specific markets, platforms that manage contractor payments across borders. Some are genuinely good. Most cover some regions and not others. All of them cost more than people expect. None give you the certainty you want.

I have done my best to get it right, and I am not fully confident I have. That is the honest position, and I think anyone running distributed teams who tells you they have it completely sorted is either not looking closely enough or not being straight with you.

The other thing I am still working out is the difference between running a distributed team and managing offshore resources. They sound the same and they are not. Offshore resources are a procurement question, where do I find the right people at the right cost. Running a distributed team is a management question, how do I build a coherent culture and operating rhythm across time zones, languages, and working contexts. Both are solvable. They require different thinking, and I have not separated them in practice as cleanly as I would like.

I am telling you all of this, the unsolved parts included, for a reason. When someone shows you a distributed team that produces enterprise-grade work at a fraction of onshore cost, the temptation is to see the result and assume the path was simple. It was not. It was the legal layer, the lead you spent

years finding, the cultural work you do for months before it takes, the honesty you pay for in uncomfortable conversations, and the regulatory approximation you are never fully sure about, all held together at once. Doing it properly is a discipline. Doing it once, by luck, teaches you almost nothing about doing it again.

The contrarian take, most agencies are running a business model you do not understand

The standard advice when you need software built is to find a reputable agency. Get three quotes. Check the portfolio. Read the reviews. Sign the contract.

Here is what that advice skips. The easiest business model to run in professional services is churn and burn. Acquire a client, get as much revenue from them as you can, cut the cost of delivery as far as it will go, and move on. The long-term view, doing exceptional work and becoming the trusted partner rather than the vendor, is harder to execute and produces a slower return. Most agencies, most consulting firms, most professional services businesses are not running the long-term model. They are running the short-term one and presenting it as the long-term one.

What that looks like in practice. The scope expands in ways that feel reasonable at the time but were always the plan. The team in the pitch is not the team doing the work. The seniors sell, the juniors deliver, and the gap in capability gets absorbed quietly. The deliverable is built to look complete rather than to be complete. The quote is not built up from what the work actually costs. It is back-calculated from what they think you will pay.

The reason it works is low understanding on the client side. If you cannot evaluate the work, you cannot evaluate the vendor. You are always seventy percent done, always needing one more thing, never sure whether the problem is the vendor or whether the work is just this hard. That uncertainty is not accidental. It is the operating environment the model depends on.

I have seen this at every price point. The fifty thousand dollar agency and the five million dollar consulting firm run versions of the same model. The sophistication of the presentation scales with the fee. The underlying dynamic does not. The genuinely exceptional firms exist, and I have worked with a small number of them. The difference is not capability. It is whether the incentives of the engagement align their success with yours. Before you engage anyone, invest in enough context to know what good looks like. Pay a small amount for a small thing first. Treat the result as the only signal that matters.

You will notice that none of the hard parts of this chapter were about writing code. They were about people, trust, law, and the years it takes to build a structure that holds. That is not an accident. The lever that comes next is the one people assume is purely technical, the tools and the automation, and it is the same story in different clothes. The tools are the easy part. Knowing which of them to trust, which to ignore, and where the genuine effort belongs is the part that takes a career. That is where we go now.

Before you move on

- What is your honest current position on offshore, and what is that position based on? Direct experience, or something you heard?

- If someone built your product with people in another time zone, what would you want to know about how that team is run before you said yes?
- Do you have a clear enough brief that someone working asynchronously in another time zone could execute it without a daily check-in? If not, what is missing?

17. Tools and Automation

> *Everyone thinks the build is the whole job. It is the part with the mystery, so they assume the difficulty is all in there, in the code. It is not.*

There are two lies people believe about building software, and this chapter exists to take both off the table.

The first is that the build is hard because the code is hard. The second is that the newest tool, the shiniest bit of AI, is the thing standing between you and shipping. Both are wrong, and believing either is how good products die stuck at eighty percent. The build is mostly assembly. The tools are mostly noise. What is genuinely hard is knowing what to buy, what to build, what to trust, and where to spend the effort you have. That knowledge is not on a pricing page. It is what this chapter is about, and it takes hundreds of builds to internalise rather than one.

Most of your product is a solved problem you can buy

If the earlier levers were done well, the build is the most predictable phase of the whole thing, and it comes together far faster than anyone expects, because most of it is not building at all. It is assembly. The reason so many products die in the build is almost never that the hard part was too hard. It is that the team treated the easy part like the hard part, hand-built things that should have been bought, and

drowned under a hundred custom pieces, none of which were the reason the product existed.

Let me get concrete, because concrete is the point. Every product, almost without exception, needs the same handful of foundational pieces. It needs somewhere to live on the internet, hosting, so that when someone types your address the thing appears and stays up. It needs a way to take money, payments, so a card can be charged without you touching it. It needs somewhere to keep information and get it back reliably, a database. It needs a way to know who a user is and let them in safely, authentication. It needs to send email, the confirmations and receipts and resets that every product sends. It needs somewhere to keep files that users upload. It needs a way to do work in the background and on a schedule, and a way for one system to tell another that something happened. And it needs a record of what occurred, so that when something goes wrong you can see what led to it.

That list is most of your product by volume. Here is the part that should change how you think about the whole build. Not one item on it is unique to you. Every product needs hosting. Every product needs payments. The problem of taking a card payment safely has been solved, exhaustively, by companies whose entire existence is solving that one problem for everybody, and they solve it better than you ever will, because it is all they do, all day, for millions of businesses. The same is true for each of the others. There is a category of company for hosting, a category for payments, a category for databases and authentication, a category for sending email. They are brand-name, proven, used by enormous companies and tiny ones alike, and you wire them in rather than write them.

Across every product I build, that foundation is close to the same. It gets standardised once and reused. Hosting that

deploys the product automatically whenever the code changes. A payments provider that handles the cards and the subscriptions and the receipts. A database and authentication service in the right region for the users. An email service shared across everything. Background jobs, scheduled tasks, webhooks, an audit trail. That is the chassis, and it is the same chassis under product after product, because the needs are the same and the solutions are already excellent. When someone comes to me with a new idea, that seventy percent is not where the time goes. It is close to paid off before we start.

I want to be careful here, because "just buy the seventy percent" sounds like a slogan you could execute on a Saturday, and it is not. Knowing which provider, in which region, wired in which way, with which fallbacks, tuned to which failure modes, is a body of knowledge that took me years and a graveyard of wrong choices to build. The slogan is easy. The chassis that holds under a real product is the thing you get wrong the first several times.

Why custom-everything leaves people stuck at eighty

Now the other path, the one that produces the stuck product. You have probably been on it, or paid for it, and I want you to recognise it cold.

The other path builds that seventy percent by hand. Not out of malice. Usually out of a mix of habit, pride, and a genuine belief that a custom version will fit better. The developer writes their own authentication instead of using the proven one. Their own way of handling files. Their own background job system. Each of these, in isolation, is a

defensible choice. A capable developer can build any one of them. That is exactly the problem. They can, so they do.

And then the small things start to pile up. The hand-built authentication has an edge case nobody thought of. The custom file handling breaks on a large upload. The homemade background system loses a job occasionally and nobody can work out why. None of these is a disaster on its own. Together they are the disaster, because now the team is spending its days maintaining a stack of custom plumbing that a proven tool would have handled for free, and every hour on that plumbing is an hour not spent on the part that is the actual product. The polish never comes, because the team never reaches the part that needs polishing. They are too busy holding the boring parts together.

That is the eighty percent trap, mechanically, close up. It is not one big failure. It is the accumulated weight of a hundred custom pieces that never needed to be custom. A mobile vet product I was brought in to rescue sat at eighty percent for two years, through two developers and thirty thousand dollars, and at no point could anyone name what was wrong, because there was no single wrong thing. There was a pile of small things, each built from scratch when it should have been assembled, and the pile got heavy enough that nobody could see the top of it. When we rebuilt it, the biggest single change was not cleverness. It was refusing to hand-build the seventy percent. We assembled the foundation from proven tools and spent the saved effort on the booking workflow that was the actual product. Live and solid in weeks, from two years stuck, and most of the difference was that one decision.

The custom pieces do not only cost you time to build and maintain. They cost you the pivot. A proven tool that turns out to be wrong can be swapped for another proven tool,

because the whole category is designed to be interchangeable. A custom piece welded into the middle of your product cannot. So custom-everything does not only leave you stuck at eighty percent. It leaves you unable to move even once you realise you need to, which is the worst version of stuck there is.

I want to name something honestly. The developer who did that to the vet product was not a bad developer. Most of the people who leave products stuck at eighty percent are not bad at their jobs. They lacked one specific discipline, the discipline to not build what they were perfectly capable of building. That is a strange, counterintuitive skill, and it is one of the hardest to hire for, because every instinct a capable engineer has runs the other way.

The thirty percent is where all your effort goes

If the seventy percent is where people waste effort, the thirty percent is where they fail to spend it, and the two mistakes are the same mistake seen from opposite sides.

The thirty percent is the part that is genuinely yours. It is the reason the product exists. It is the specific capability nobody else has built the same way, the thing that came out of your years of watching the problem. For one enterprise product, it was the logic that captured a twenty-year expert's judgement and let a novice reach a fraction of his answer in minutes. For an internal tool I built to check Power BI files, it was the scoring and reasoning that a decade of specific audit experience made possible, the part that understood what it was looking at. In both, that core was maybe a third of the whole product by volume, and close to all of it by value.

This is the part where custom is correct. This is the part you build carefully, iterate on, get right, spend the effort the seventy percent did not deserve. And you can only afford to spend that effort here because you did not squander it back there. That is the entire economic logic of the split. Buy the boring seventy so you can afford to lavish attention on the thirty that matters. Teams that hand-build the seventy arrive at the thirty exhausted and out of budget, which is why the unique part, the part that was the whole point, so often ends up the roughest thing in the product. They spent their best energy on authentication.

There is a discipline to keeping the thirty percent flexible even while you pour effort into it, and it matters because the thirty percent is exactly where you are most likely to be wrong. It is the newest part, the least proven, the part no real user has tested yet. So you build it in layers. The general capability sits underneath, and the specific behaviour sits on top as a surface layer you can change cheaply. When you build an AI feature, the model is a general capability you can swap, and your specific instructions to it are the surface layer. So when the model providers ship something better, and they will, you change the layer and keep the product. I learned that one the hard way. In my own startup I spent months building a custom back-and-forth reasoning loop for an AI feature, and by the time we had it working the model provider had released a native capability that did the same thing a hundred times better out of the box. If I had built it to be swappable I would have swapped it in an afternoon. Instead I had built the wrong thing carefully. Even in the part that is yours, you build for the pivot, because the part that is yours is the part you have not yet been proven wrong about.

Everything I just described, the layering, the swappable model, the surface layer, is not something you improvise on a

first build. It is a pattern you arrive at after being burned by the version without it. That is the through-line of this whole book. The methods are learnable. Learning them the slow way, one wrong build at a time, is expensive.

The unit of production has changed

I have been describing the build as assembly, and it is. But I have been quiet, so far, about who does the assembling, and that has changed more in the last year than in the ten before it. It is the biggest shift I have lived through in how software gets made, and most books on this subject were written before it happened, so they teach you to manage a thing that no longer exists.

For as long as anyone reading this has been alive, the unit of production in software was a person working through a list. One developer, one task at a time, in order. You wanted more done, you added more people, and then you paid the tax of getting those people to agree with each other, which is most of why big teams move slowly. That was the shape of the whole industry. It is not the shape anymore.

The way I build now, the unit is not a person working through a list. It is one person directing a fleet of AI agents that work in parallel. I hand out the pieces, many at once, and the agents do the mechanical middle of each one. This is not a small speed-up. It is the reason one person can now ship in a day what used to take a small team a fortnight, because the middle of the work, the part that used to be the whole job, the typing, the wiring, the boilerplate, the following of a known pattern to its end, is the part the agents are genuinely good at. They do not get bored. They do not context-switch. You can run ten of them at once and none of them minds.

I want to be careful not to sell this as magic, because the whole point of this section is that it is not. A fleet of agents pointed at a problem with no discipline does not produce ten times the work. It produces ten times the mess, faster than you can read it. The speed is real, but the speed is not the skill. The skill is entirely in how you direct them, and the rest of this section is that discipline, because without it the new unit of production is just a new and expensive way to get stuck at eighty percent.

The human owns the bookends

Here is the shape that makes it work, and it is worth holding onto, because it is the whole thing.

Every piece of work has three parts. There is the investigation at the start, working out what the real problem is and what to actually build. There is the middle, the making of the thing. And there is the validation at the end, checking that what got made is genuinely right and not merely finished. The agents run the middle. The human owns the two ends. The bookends.

The reflex is to assume that if the machine now does the middle, the human matters less. It is exactly backwards. The middle got faster, so the bookends matter more, not less. When the making of a thing took a fortnight, a vague brief at the start could be corrected along the way, because there was a fortnight of along-the-way to do it in. When the making takes an afternoon, a vague brief at the start just produces the wrong thing by dinner, beautifully built and pointed at nothing. The faster the middle runs, the more the quality of the whole depends on the two ends the human still owns. Getting the problem right before the fleet starts, and knowing the difference between done and right when it stops.

This is a promotion of your judgement, not a replacement of it. And here is the part I most want a non-technical expert to hear, because it is the part the industry keeps getting wrong. The bookends are exactly your strength. The investigation at the start, knowing which problem is the real one and what actually matters, is the thing your years in the domain gave you. The validation at the end, knowing when the answer is genuinely right and not just plausible, is the same knowing, pointed the other way. The middle, the part that just got automated, was never the part that needed you. The bookends, the part that cannot be automated, is the part you were always best at. The machine did not take the valuable work. It took the work around the valuable work, and left you standing exactly where your expertise lives.

That is the same truth the whole book keeps circling, seen from a new angle. The expert who owns the pain is the irreplaceable ingredient. Now here is why, mechanically. The machine can do the building. So the scarce thing, the thing that decides whether the product is any good, is the judgement at the bookends. Which is precisely what you have and the machine does not.

The one rule that makes parallel work safe

Running a fleet is not free of danger, and the danger is specific and worth naming, because it is the one that corrupts everything if you ignore it.

When you have many builders working at once, they must work on strictly separate pieces. Never two agents on the same thing at the same time. If two of them are editing the same part of the product in parallel, they cannot see each

other, and they will overwrite and tangle each other's work, and the result is worse than either would have produced alone, because now it is broken in a way neither of them knows about. This is not a rare edge case. It is the default outcome of pointing two workers at one thing with no coordination, and it happens silently.

The fix is not cleverness. It is isolation. Before the fleet starts, the work is cut into pieces that do not touch, and each agent gets a piece that is genuinely its own. One works on the payments corner, one on the file handling, one on the thing that is the actual product, and the boundaries between them are real, so nothing one agent does can land on top of another's. The plain principle underneath it is this. Parallelism is free only when the pieces do not overlap. The moment they overlap, parallelism stops being a speed-up and becomes a source of corruption. So the work that used to go into managing people getting in each other's way now goes into cutting the work so cleanly that they cannot. Same problem the big slow teams always had. Different, and much cheaper, solution.

Build dark, gate live

The next discipline is about the on-switch, and it is the one that keeps the whole fast machine from doing something fast and wrong out in the real world.

New capability gets built completely and left switched off. The whole thing, done, tested, sitting there ready, and not on. It gets turned on only by a deliberate human step, and often only after a tiny live test with real but trivial stakes, the kind of test where if it goes wrong the damage is a rounding error. A canary. You send a real thing through the real system at a size where being wrong costs almost nothing, you watch it

land correctly, and only then do you open the tap. You never let an automated system loose on real money or real customers on a hope.

For anything that spends real money, I want two independent switches, and both must agree before a single dollar moves. Not one check twice. Two different checks, arrived at separately, that both have to say yes. Because the failure mode you are guarding against is not the honest mistake you would catch on a second look. It is the confident error that sails through one check because the one check was wrong in the same direction. Two independent agreements is how you catch that, and money is exactly where you want to catch it.

This is the same instinct as the drag-and-drop workflow tool from the first lever, and the same as building the thirty percent in swappable layers. Automate the work, but keep a human hand on the on-switch. The fleet can build the machine that spends the money. The fleet does not get to decide to spend it.

Agents that check their own work, and what they must check

The most valuable thing you can point an agent at is not building. It is checking. And what it checks is the whole game.

The move that pays for itself over and over is an automated check that looks at what the customer actually sees. Not a dashboard. Not a comforting green number. The actual screen a real person would look at. An honest check opens the thing the way a customer would and confirms the price on the screen matches the price in the ad, that the buttons actually go somewhere, that the thing works on a

phone and not just on the big monitor it was built on. It confirms these before anyone is allowed to say it is done.

I am specific about this because the alternative is the trap. Aggregate numbers hide broken things. A dashboard can show you a healthy total while a whole category of customer hits a dead button, because the total is an average and the average drowns the failure. The number says fine. The screen says broken. Looking at the real screen catches what the number was built to hide. So the check that matters is the one that goes and looks, at the level of one real customer, at the thing that customer would actually touch.

And there is a second half to this. Every automated action that goes out into the world leaves a trail a human can read. When the fleet does something, ships a change, sends a thing, moves a dollar, it writes down what it did in plain terms, so that afterwards a person can follow exactly what happened and why. Nothing the machine does happens in the dark. This is not bureaucracy. It is the thing that lets you trust a system you did not personally watch, because when you go to check, the record is there, in language you can read, and it either reassures you or it tells you precisely where to look.

The economics that make the discipline obvious

There is a reason all of this is not just prudent but rational, and it comes down to what each thing costs.

The fleet runs on tools the builder already pays for. Running an agent, running ten agents, running a check, running the same check ten more times to be sure, all of this costs almost nothing at the margin. It is already bought. But every dollar of real-world spend is still expensive. An ad that

runs against a broken page costs real money. A wrong build that ships to real customers costs real money and real trust, which is dearer than the money. The two sides of the ledger are wildly out of balance. Checking is nearly free. Being wrong in the market is not.

So the rational move falls straight out of the arithmetic. Spend the cheap thing freely to protect the expensive thing. Check everything ten times over with agents, because the checking is nearly free and the being-wrong is not. Where a person would check once because their time is precious, the fleet checks ten times because its time is nearly worthless, and the tenth check is the one that catches the thing the first nine agreed to miss. This is the same logic as buying the seventy percent instead of building it. Do not spend the expensive resource where a cheap one will do, and here the expensive resource is real-world spend and the cheap one is the fleet. Spend tokens to save dollars. Once you see the ledger, it stops being a discipline you have to remember and becomes the only sensible thing to do.

The frame for every tool decision, AI is estimation

Now let me widen out from the build to the tools you use to run the whole operation, because the same discipline applies, and AI has made getting it wrong more expensive than ever.

I test a lot of tools. Probably more than most people would consider reasonable. And I have landed on a mental model for AI and automation that is simple enough to apply fast and has not let me down. AI is estimation. That is the whole frame. Anytime you need close enough and you need a lot of it, AI is the answer. Anytime you need exactly right, or the stakes of

being wrong are high, or the output requires genuine judgement about context the model does not have, AI is not the answer yet, and pretending otherwise will cost you.

The two categories where I consistently get real value are removal and challenge.

Removal is the more valuable of the two. The best AI tools I have adopted are the ones that completely eliminated something I used to have to think about. Automated meeting notes and action extraction is the biggest single change in how I work in the last two years. I do not take notes in meetings anymore. I do not chase what was agreed. That entire category of cognitive overhead is gone. Not reduced. Gone. And gone is a different outcome to reduced.

The second category is challenge. Brainstorming, testing assumptions, first-pass thinking on problems I have not fully formed. This is where volume and imperfection are exactly what you want. You are not looking for the right answer. You are looking for ten angles you had not considered, three of which are interesting and one of which reframes the whole problem. AI is genuinely good at this.

Code reviews sit in the same category. A quick function, a first draft, a structural check, a good use case. Complex architectural decisions, subtle business logic, anything where being confidently wrong is worse than being slow, not there yet. The honest description of where AI sits with code right now is the very confident first-year student who has just finished their first semester and believes they know everything there is to know about software. Impressive for the level. Dangerous if you forget the level.

Alignment beats the best tool

The single most useful principle for tooling in any team is alignment. Not with best practice, not with the most sophisticated option, but with what already exists and what people already use.

Every tool change has a cost that rarely shows up in the evaluation. The overhead of learning something new. The dip in productivity while habits reform. The resistance from people who were finally comfortable with the previous system. These costs are real, they compound across a team, and they are almost never in the business case for the switch.

Using a slightly worse tool that requires no change is a better outcome than using a better toolkit that requires meaningful adaptation. Keep the structural stuff, project management, documentation, communication, as boring and stable as possible. Nobody ever got a competitive advantage from switching project management tools.

For your personal stack, keep it consistent and avoid swapping too much. Switching costs are real even for one person. The time spent evaluating, migrating, and building new habits around a tool that is marginally better than the one it replaced is almost never recovered in the gain. Every so often something genuinely changes the game, and when that happens the switching cost is worth paying immediately and completely. The hard part is developing the judgement to tell the two apart fast, because the people selling you both use identical language. That judgement is the skill. The tools are just the surface it operates on.

How I actually evaluate tools

Most advice on tool evaluation is aspirational nonsense. Pilot programs, scoring matrices, formal review periods. Nobody

does that, and nobody should.

My process. If I am the one driving it, I drive straight in and try to use it for something real. If I cannot get value out of it fast, sometimes five minutes, sometimes a day, I cancel it and move on. Anything beyond that is aspirational. I will tell myself I will go back and learn it properly. I will not. I have a graveyard of monthly subscriptions I forgot I signed up for. Learn from that.

The five-minute test sounds shallow. It is demanding. If a tool cannot show me something useful in five minutes it either has a serious onboarding problem or it is not the right tool for what I need. Both are real signals. A tool that needs three hours of setup before it does anything useful is a tool that needs three hours of setup every time someone new joins your team. That cost compounds.

Three recent tools that changed how I work. One for LinkedIn automation, set it up, forget about it, never think about it again. A skill for generating slide decks that made my most hated task a great deal faster. Meeting transcription with actions collected automatically into my task list. The pattern across all three is the same. They do not make a task easier. They eliminate the task, or eliminate the thinking required to manage it. That is the standard worth holding new tools to before you commit.

The limit of a leader-driven approach, and how to beat it

Here is the honest admission. I miss things constantly. Tools that are enormous for someone on my team are invisible to me because I am not doing that work every day. An AI-powered pull request review tool that transforms code review

is extraordinary for a developer who does code reviews. I was not doing code reviews, so I missed it entirely. A design tool that changes how fast someone produces visual work will never surface through my testing, because I am terrible at design and do not spend time there.

That is the real limit of a leader-driven approach to tooling. Your testing is bounded by your own work. The multiplier is enabling the same testing instinct in whoever is doing the parts you are not. On a single-product build that is a small number of people, sometimes one developer, sometimes a build partner, but the principle is identical. The person writing the code is far better placed than you to spot the tool that makes writing code faster. The person doing the design will see the design tool you never will. Your job is to want them to look, and to make it easy for them to act when they find something.

The structure that makes this work is removing the friction from acting on what people find. Even at a scale of one or two, that can be as simple as an agreed small amount they can spend on a promising tool without a back-and-forth. What it communicates is that their judgement is trusted and their curiosity is sanctioned. The result is that the people closest to the work stay ahead of what is possible rather than waiting for you to notice first. And if a partner is building your product, this is a fair thing to expect of them, that they are the ones tracking the tools in their own domain, not you. Notice that this is a leadership design decision, not a technical one, which is a theme you are going to see get louder as this book goes on.

What bad tool decisions actually look like

The worst tool decisions I have made are not the ones where I picked the wrong tool. They are the ones where I picked the right tool at the wrong time and built process around it before I understood it well enough.

You build the automation, train the team, integrate it with three other things, and six weeks later you realise the tool does not do the thing you needed, or does it in a way that creates more problems than it solves. Now you have a migration problem on top of the original problem. The cost is not the tool cost. It is the rework, and the team's trust in the next tool decision.

The protection is the same as the general principle. Do not build deep process around a tool until you have used it long enough to know it is staying. Use it manually first. Automate later. The temptation to automate immediately, especially for technically minded people, is strong and usually premature. Get the result first. Then make the result repeatable.

Keep it simple, because simplicity is a defence

Running through this whole phase is one principle that does more than any other to determine whether you ship, and it is the least glamorous thing in the book. Keep it simple. Not as a value. As a defence.

Complexity is what lets small things pile up. Every additional custom piece, every extra case handled before it needed handling, every clever bit of engineering that was not strictly required, is another thing that can break, another thing that interacts with the others, another thing standing between the team and the polish. Technical complexity is exponential. A little more scope is not a little more work and a

little more risk. It can double the number of ways the system can fail, because each new piece can interact with every existing one. Simplicity is not aesthetic preference. It is the thing that keeps the number of possible failures low enough that you can finish.

So through the build, the discipline is to keep asking whether each piece needs to exist at all, and to resist the technical instinct to handle the case that has not come up, to build the abstraction that might be useful later, to add the flexibility nobody asked for. The simplest thing that does the one job, assembled from proven parts wherever the part is not the reason you exist, is the thing that ships and stays shippable.

The support automation from the first lever is the clearest example of this. We deliberately used a drag-and-drop workflow tool for the first version, specifically because it forced us to stay light. You cannot over-engineer in a tool that does not let you, and that constraint was the feature. We got the end-to-end thing working, saw where it broke against real cases, fixed those, and only then thought about hardening it. The path from a working simple demo to a production system is not a rebuild. It is more care around the edges, more error handling, more coverage of the cases that turned out to matter. Same core, more polish on the parts that earned it. You do not start again. You add the care where the real use showed you it was needed.

The whole thing in one frame

Here is the build held in one picture, so it stays with you.

You have a scoped simple version, sorted into two piles. The seventy percent, the hosting and payments and database

and authentication and email and the rest, you assemble from brand-name proven tools that already solve those problems better than you could, standardised once and reused, so it comes together in a fraction of the time and stays swappable when you are wrong. The thirty percent, the one thing that is the reason the product exists, gets all the effort you saved, built carefully but in flexible layers so that even the part that is yours can move when the real user teaches you something. And through all of it you keep the thing as simple as it can be while still doing its one job, because simplicity is what stops small things piling up into the eighty percent trap that has nothing wrong with it except that it will never be done.

That is how a product comes together fast, ships, and stays flexible enough to survive the fact that you were wrong about something. It is not a secret technique. It is the refusal to build what you can buy, the willingness to spend on what only you can build, and the discipline to keep it simple enough to finish. Do that, and the thing that was stuck at eighty percent for two years is live and solid in weeks.

I want to be honest about what that sentence is hiding, though. Each of those three moves is a decision made hundreds of times a day inside a real build, and each one runs against a capable person's instincts. Getting them right, consistently, across a whole product, is not a checklist. It is a judgement you develop, and the only way to develop it is to make the wrong call enough times to feel the cost. That is why the same method produces a stuck product in one team's hands and a shipped one in another's.

The contrarian take, everyone is chasing the wrong five percent of AI

AI is a blanket term for a collection of tools, some decades old and some genuinely new. What has changed is not the underlying logic. It is the availability, the barrier to entry, and the quality of the output out of the box.

Here is the simplest honest description of what AI is. If you want to know how tall someone is, you collect related data, weight, shoe size, arm span, plot it, draw a line through it, and make a guess. Machine learning is about getting fancier with how the line gets drawn, clustering groups, spotting patterns across more dimensions, updating as new data comes in. But it is still, at bottom, an educated guess. The current generation of large models does this at a scale and sophistication that produces genuinely extraordinary output. It does not change what it is.

The reason this matters is confidence. When I run a traditional machine learning model in a high-stakes setting, there is a validation process. Backtesting. A range of scores. An explicit understanding of where the model works well and where it does not. The current AI stack does not come with that calibration built in. I asked a model for a detailed legal document once. It produced one. It looked exactly like a legal document. I had zero ability to evaluate whether it was correct. That is extraordinary and dangerous in equal measure, and the danger is not that AI produces bad outputs. It is that the bad outputs are increasingly hard to tell apart from the good ones.

The instinct most people have here is almost always wrong. The pull is to point your AI investment at the most critical and most sensitive use case, because that feels important and that gets funding approved. Those are also the use cases that require the highest certainty and the most careful risk management. So the projects that get the most investment are the ones where AI is hardest to deploy well

and the failure rate is highest. I watched a mining company invest close to half a million dollars building a central AI platform. By the time it was finished the underlying technology had moved on and the platform did not work properly with the current models. The lesson is the same one I paid for in my own startup. Build for change, not for correctness. The goal is not to build the right thing. It is to build something you can swap out fast when the right thing becomes something different, which in AI it will, and soon.

Here is the thing nobody wants to say out loud. In most organisations, somewhere between ninety and ninety-five percent of what people do every day is administration, coordination, preparation, and organisation. The five percent that is genuinely irreplaceable, the expert judgement, the relationship, the decision that requires real accountability, is surrounded by a vast amount of work that is necessary but not valuable. AI is genuinely good at that ninety-five percent. If you took a team, gave everyone access to a current general model, spent a day training them on what it can do and what not to share, and let them find their own applications, the productivity uplift would outperform any single critical AI project.

The reason this does not happen is governance. Not security. Governance. The decision needs someone willing to own the outcome and make a call that is not individually sponsored. Most AI projects are the opposite, individual sponsors, individual risk budgets, optimised for safety over impact. The people genuinely pulling ahead with AI are not doing it because they found a better use case. They made a broad decision, accepted that some things would go wrong, and gave people permission to find the applications themselves. That is a governance decision, not a technical one.

Before you move on

- What is one task you or your team does regularly that, if it disappeared entirely, nobody would miss, and could it be automated away?
- What is a tool you are using out of habit rather than because it is the best option? What would it take to change?
- What is the AI use case in your specific work that sits in the close-enough-at-scale category, where ninety percent accuracy at ten times the volume would be genuinely valuable?

Everything in this chapter comes back to the same truth. The tools are not the hard part, and the code is not the hard part. The hard part is judgement, knowing what to buy, what to build, what to trust, and where the effort belongs, developed over enough builds that the wrong instincts have been trained out of you. But judgement in one person's head does not ship a product. A team has to carry it, day after day, and a team only carries it if it holds together. Which brings us to the last lever, the one that decides whether any of the others matter. Keeping the people who do the work.

18. Retention and Culture

> *Your job is to get the most out of your team, not to make yourself feel good.*

This is the lever people underestimate the most, and it is the one that quietly decides whether the other five ever amount to anything. You can get context right, structure right, the right people, the right tools, and still watch it all come apart, because the person or two carrying the work walks out the door faster than you can replace them. Losing them does not just cost you a hire. It resets everything they knew, breaks the trust you spent months building, and leaves the product stranded exactly where they left it. On a single-product build this is more dangerous, not less, because there is no bench. The person who leaves may be the only one who understood the part that matters. Nobody puts that in the pitch. I am going to.

I said earlier that this lever could be argued as a result of the others rather than a lever in its own right. That is partly true. If you get context, structure, the right people, and a working model, a lot of what falls under retention and culture emerges on its own. But not all of it. And the part that does not emerge on its own is the part that takes the most discipline, because it is the part that costs you something personally every time you do it right. That is the whole tension of this chapter. The hardest part is not knowing what to do. It is being willing to pay the personal price of doing it, over and over, when it would be easier not to.

Your people are what make you good, full stop

There is a specific failure of perception I see in leaders who are otherwise doing most things right. They are aware of their own decisions, their own strategic calls, their own contribution to what the team produces. They are far less aware of what the team is doing in the background to make those outcomes possible. Over time that asymmetry creates a distorted picture, one where the leader's contribution looms large and the team's becomes ambient background, expected rather than noticed.

There is no place for ego in leading a technical team. The people on your team are the ones in the trenches. They are the ones writing the code, building the models, debugging the thing that broke at eleven o'clock on a Friday night. The product looks good because they made it look good. Remembering that, genuinely remembering it, not just saying it, is the foundation of everything else in this chapter.

The number one place this breaks down is money. When things go well, do you actually follow through on what you implied or promised? The bonus you mentioned when the project was under pressure, does it arrive? The rate you agreed when you needed someone urgently, does it get reviewed when the engagement stabilises? Your team is asking these questions, even when they are not asking them out loud. And the answers they observe decide whether they trust you or just work for you.

On hard conversations, you just have to do them

If retention is a genuine goal and not just something you say, then you cannot make the quick cut when someone is struggling, or sweep a conflict under the rug because addressing it is uncomfortable, or avoid the salary conversation because you are not sure how it will go. You pick up the phone. You have the conversation. You come up with a plan together and you treat the person the way you would want to be treated if the positions were reversed.

This matters beyond the individual relationship. Everyone on your team is watching how you handle it. The culture is not what you intend. It is what you demonstrate under pressure. That last sentence is the whole game, and it is why culture cannot be delegated to a document or a values poster. It is decided in the specific moments when doing the right thing is expensive, and everyone is watching to see whether you actually will.

Accountability is what most people lack

Before you assess whether someone made a mistake, ask yourself whether the target was clear enough. Whether they had what they needed. Whether they were overworked or under-supported. Whether you were providing enough coaching or visibility to make the standard obvious before the error happened rather than after. Most performance problems, traced back honestly, have a leader decision somewhere near the root cause.

Offshore makes this worse, because the blame shift is so easy and so available. Something goes wrong, the offshore team is involved, and suddenly the story becomes about the risks of offshore, the cultural differences, the time zones, the communication challenges, rather than about whether the

work was briefed clearly. It is one hundred percent on you. Lead from the front. Always.

The first two weeks

The goal in the first two weeks is not to onboard someone into a process. It is to get on the same page as a person. I run through our principles and expectations, but most of the expectations are things like this. Do not respond if I message you and you are not working. Do not work outside your hours. I will never send something urgent without first asking permission.

The signal that it is working is when they disagree with me. Not rudely, but genuinely. When I float an idea and they push back, challenge it, offer a better version, I know the relationship is functioning correctly. If they can openly hold me accountable for deliverables, they will be comfortable with everything else. The accountability runs both ways, or it does not work at all.

I want to flag how counterintuitive that is, because it is the opposite of what most onboarding optimises for. Most onboarding wants a smooth, compliant, grateful new hire. What you actually want is someone who will tell you that you are wrong in week two. Building an environment where that happens, and happens safely, is not a template you download. It is a thing you have to model, repeatedly, before anyone believes it is real.

What I personally do

The hardest moments for me are when I am overworked or the finances are tight. Both states produce the same instinct,

to cut the personal investment short, to skip the check-in, to push for something urgent without the care I would normally take. I try to treat those moments as signals rather than excuses. If I am overworked enough that I am starting to compromise on how I treat my team, something in the structure is wrong and I need to fix the structure rather than absorb the overflow at the team's expense.

The genuinely hard version is the urgent situation. When it happens I acknowledge it is outside our normal norms, make clear it is an exception and not a new expectation, and explain what I am personally changing so it does not happen again. Not the client caused this. I missed a planning step that caused this, and here is what I am changing. That last part is what most leaders skip. The acknowledgement without the specific personal accountability is just an apology. The accountability is what makes it mean something.

Protecting the people who build for you

I told the story earlier of the client I cut loose because of how they treated my team. There was another like it, smaller, a website favour where a client was openly contemptuous toward one of the best designers I have ever worked with, and I stopped the work the same way and for the same reason. I am not going to retell it in full. The pattern is the one that matters. When someone you are paying is being treated with contempt, you protect the person, not the invoice, and everyone watching learns whether the rules you set were real.

One thing about it is worth saying directly, because it is the part that applies to you even if you never employ anyone. This protection is the first thing that falls off the radar with a contractor and largely does not exist at all with an offshore builder. The check-ins do not happen. The personal

investment does not happen. The protection from bad behaviour does not happen. They are treated as a resource rather than a person, and everyone quietly accepts that as the appropriate standard for that category of engagement.

It is not. If you want the same quality of output, the same ownership, the same willingness to run hard and speak up when something is wrong, you have to create the same conditions. This holds whether it is one contractor, a build partner, or the first person you bring on. You cannot expect the culture to apply to some of the people building your product and not others. And I want to be honest that protecting someone can have a bill attached. Stopping a paying engagement to do it costs revenue and an awkward call. That is the point. The moments that build culture are the ones where doing right has a cost, and people can tell the difference between someone who pays it and someone who talks about paying it.

What hard conversations actually look like

My first instinct when something was wrong with someone on the team was to manage around it. Find a way to fix the problem without having the conversation. Restructure the work so the friction point disappears. Hope it resolves itself. It never resolves itself. I have tested this theory more times than I would like to admit.

What I do now is simpler. I reach out in whatever way that person prefers, some want a call, some want a message, some want to talk face to face. The medium matters less than the directness. And I am direct.

Something like this. Hey, I am struggling to figure out how to say this, but the honest version is that your communication style with your juniors is hurting more than helping. I get it. But it is burning them. You are doing great work and I get that they need to step up, but that approach is not working. Can I call you both and we figure out a plan together.

That is it. Not a formal performance review. Not a documented warning. A direct message that names the problem, acknowledges what is good, and moves straight to what happens next. The person on the receiving end knows exactly where they stand. They are not blindsided. They are not being managed at a distance through hints and restructured responsibilities.

The thing I had to learn is that directness is not unkind. Avoiding the conversation is unkind, to the person, to the people affected by their behaviour, and to you. The avoidance approach draws out the problem, builds resentment on all sides, and usually ends badly anyway. The direct approach is uncomfortable for about ten minutes and then most of the time you move forward.

Not always. Some conversations do not end well. But they end, and you know where you stand, and the rest of the team watches how you handled it and draws conclusions about what kind of place this is. I am spelling this out because “just have the hard conversation” reads like simple advice and is one of the hardest disciplines to hold. Every instinct pushes you toward the softer, slower, more comfortable option that quietly rots the team from underneath. Overriding that instinct, consistently, is a skill I built by getting it wrong for years first.

What actually makes people stay

The honest answer to what makes people stay is simpler than most leadership writing suggests. They feel like they are getting better, they feel like what they do matters, and they feel like you are straight with them. That is close to all of it.

The money follow-through matters enormously. When someone does something that warrants a pay rise or a bonus, do it without them having to ask. This sounds obvious, and it is. Most people still do not do it. They intend to. Something else comes up. The moment passes. The person noticed, and they will not forget.

In the early years I was not great at this. I would mean to revisit it and not. I thought intent was most of the job and that reasonable people would understand things were busy. What I learned is that intent does not pay anyone's rent. The follow-through is the signal, not the intention. When you do it without being asked, you tell someone something important about how you see the relationship. When you do not, you tell them something equally important.

The other thing I learned is that people stay when they have room to be honest. When they can tell you something is not working without it becoming a problem. When they can disagree with your approach and have it received as useful rather than threatening. The signal I watch for is when someone pushes back on something I have proposed. Not rudely. Just genuinely. That means they believe their opinion is worth having in the room. When that stops happening, I know something has shifted and I need to work out what.

The last thing worth saying. Some people will leave no matter what you do. The job is not right for where they are in their life. The work is not what they want to be doing. This is not a failure of retention. It is reality. Your job when it happens is to make the exit clean, treat them well, and mean

it. Not as a performance. Because it is the right thing to do, and because everyone who stays is watching.

What this looks like when it goes wrong

The hardest version of losing someone is not the dramatic one. It is the one where nothing went wrong.

I had a young guy on the team. Capable, motivated, did good work. Just did not like software development. Wanted to be outside. You cannot fix that. What you can do is make the exit good, be supportive, help them land somewhere better.

I worked at a consultancy once where resignations were treated like betrayal. The moment someone handed in notice they were cut out, talked about, managed out aggressively. Everyone who stayed watched. The turnover was enormous, and nobody could work out why. It was not a mystery. People do not stay in a place where they have watched the door slam behind everyone who left. How you treat people on the way out is one of the clearest signals your culture sends, and everyone who stays is reading it.

What I am still figuring out

The thing that makes this whole lever work is also the thing that does not travel well. It is personal.

The directness, the follow-through on money, the space to disagree, the sense that the person you are working with genuinely gives a damn, all of it depends on being close enough to the relationship to hold it. For you, building one product with a partner or a small number of people, that closeness is available in a way it is not to someone running a large company, and that is an advantage worth naming. You

can actually know the handful of people building your thing. You can actually protect them. The temptation, once things are going well, is to start treating them as a function rather than people, to let the check-ins lapse because the work is flowing, to assume the goodwill is banked. That is the exact moment it starts to erode.

The honest part I have not solved sits one level up from your situation. I have watched what happens when a business grows past the point where one person can hold every relationship, and the usual answer, hire managers, delegate culture, build processes, keeps the business functioning while quietly losing the thing that made the small version good. Most people call that scaling. I am not sure that is the right word. You may never need to worry about it. But the same erosion happens in miniature the moment you stop being present with the one or two people who are actually building your product, and the fix is the same at every size. Stay close enough to notice. I do not have a cleaner answer than that, and I am watching it closely myself.

Before you move on

- Think about the last time someone building your product left or disengaged. Looking back honestly, what were the signals you saw and did not act on?
- What is a commitment you have made to someone building for you, explicit or implied, that you have not followed through on? What is that doing to the relationship?
- Who among the people building your product receives the least personal investment from you right now? What would change if you reversed that for thirty days?

Six levers, one hard truth underneath

Step back and look at all six levers together. Context and communication. Work structure. The right people. Location. Tools and automation. Retention and culture. Read the list and a pattern shows up that has almost nothing to do with technology.

Getting a product built is a process challenge, not a technical one. It is scoping the simple version, buying the boring seventy percent, spending real effort on the thirty that is yours, keeping it simple enough to finish, finding the right people wherever they are, and holding a team together long enough to ship. Every lever is a place the process leaks, and every fix is a discipline that runs against a natural instinct. Build what you can buy, because you can. Skip the hard conversation, because it is uncomfortable. Chase the shiny tool, because it feels like progress. Optimise the wrong five percent, because it feels rigorous. The method is not exotic. It is the accumulated refusal of a hundred tempting mistakes, and the reason it is rare is that each refusal costs something in the moment.

I have run this loop across seven businesses and more than a hundred technical projects, and I still get parts of it wrong. That is not false modesty. It is the honest shape of the work. The value of having done it hundreds of times is not a secret blueprint that guarantees the outcome. It is pattern recognition, knowing which mistake is about to happen before it does, and having paid for the lesson already so you do not have to. A single success teaches you one path and convinces you it was the only one. Doing it again and again is what turns luck into method.

That is the honest version of what it takes to turn expertise into a real product. Not because you are not capable.

You are. But because the standard way of building optimises the wrong thing, and unlearning that, lever by lever, is harder and more skilled than anyone selling you a shortcut will admit. Now you know what it actually takes. The next decision is who you want in the room while you do it.

19. Plan the Fail

> *Fear of failure keeps people in one of two places. Either they never start, or they never stop. The cure for both is the same, and it is a number you write down before you are attached to the answer.*

Picture two people standing at the door of the same room.

The first one walks in because the thing feels good. The idea is exciting, the tech is clever, the future version of it is vivid in their head. They do not know what would tell them they were wrong, because they have not thought about being wrong. Being wrong is not part of the plan. So when the room turns out to be the wrong room, they cannot leave. Leaving would mean the whole thing was a mistake, and admitting the whole thing was a mistake feels worse than staying, so they stay. They add a feature. They rewrite the landing page. They tell themselves the market just has not found them yet. Months go by. The room is still the wrong room. They are still in it.

The second person decided the exit before they walked in. Before they spent a dollar or a weekend, they wrote down the specific thing that would mean stop. If fewer than four of the first five people I talk to would genuinely pay for this, stop. If it costs more than this to get one customer, stop. If it is not doing the one job by day sixty, stop. They walk in calm, because they are not gambling their pride on the outcome. They are running a test with a known fail line. When the signal comes back below the line, they walk out without a

fight, because leaving was always allowed. Nothing was a mistake. It was information, bought cheaply, on purpose.

This chapter is about being the second person. Not by being braver. By being more mechanical. The book has already told you, more than once, that your first plan is wrong somewhere and you do not yet know where. That is true and it is not enough, because most people nod at it and then build as though it does not apply to them. This chapter makes it a procedure. By the end you should be able to sit down and write your own kill numbers, the real ones, the ones you would actually honour when they hurt.

Decide the kill numbers first

Here is the single most valuable habit I have, and it is almost embarrassingly simple. Before I start, I write down the specific signals that mean stop, or change direction. In advance. While I still have no skin in the game and can be honest.

You remember the coaching product I killed, the one where the technology could not see the thing that mattered. What I did not tell you earlier is how I killed it. I killed it on a number I had decided in advance.

Before I built the real thing, I wrote down a number. The whole product lived or died on how accurately the technology could read what it needed to read. I decided, in advance, the accuracy threshold below which the product was not worth building. I picked the line while I was calm, before I had spent the months, before it was mine. Then I tested it cheaply. And the number came back well below the line. Not close. Not a maybe with a bit more work. Below.

So I killed it. It stung, and I killed it anyway, because the honest thing to do with a number you set in advance is to obey it. The version of me who set that threshold was smarter and more objective than the version of me who would have been staring at the disappointing result, already attached, already reaching for reasons the number did not really mean what it plainly meant. That is the whole trick. A number you decide before you are attached is honest. A number you invent after you are attached is a rationalisation wearing a number's clothes.

Contrast that with the founder who has no number at all. They keep going because stopping feels like failure, and because with no line drawn in advance there is never a specific moment that says stop today. There is always a slightly better version just ahead. Every result is ambiguous enough to be read as encouraging if you need it to be, and they need it to be, so it is. They do not stop when the evidence says stop. They stop when they run out of money, or energy, or marriage. That is a far more expensive way to arrive at the same place.

You do not need one heroic number. You need a small handful, each measuring a different way the thing can be wrong. Three kinds cover most of it.

There is the validation number. If fewer than some count of the first real people say yes, I would pay for this, then stop. Not a survey. Not a like. A real conversation where a real person with the real problem is asked to say, out loud, whether they would hand over money. Pick the count in advance. Four out of five. Three out of five. Whatever you decide, decide it before you start hearing the answers, because once you are hearing them you will quietly move the line.

There is the money number. If it costs more than some amount to get a single paying customer, the math does not

close and no amount of belief changes that. This one is beautiful because it is arithmetic and arithmetic does not have feelings. If a customer is worth two hundred dollars to you over their life and it costs three hundred to acquire one, you do not have a marketing problem, you have a business that loses money on every sale. Decide the ceiling in advance. Then when the real cost comes in above it, you already agreed what that means.

There is the time-boxed number. If it is not doing a specific thing by day thirty, or sixty, or ninety, it is the wrong thing. Note the shape of that sentence. Not if it is not perfect, not if I do not feel good about it. A specific thing, by a specific day. Live and used by ten real people by day thirty. Producing the core result reliably by day sixty. Its first paying customer by day ninety. The date and the milestone are both fixed in advance, so the day arrives and either the milestone is there or it is not, and there is no room to negotiate with yourself about it.

Write yours down now, before you go further, while you are still calm. That sentence, written before you are attached, is the most valuable line in your whole plan.

The cheap test before the expensive commit

Kill numbers only work if you can read them before you have spent the real money. So the game is to buy information in small amounts, cheaply, at each step, before you earn the right to spend more. Here is the actual staircase I climb, and every single step can end it.

The first step is a quiet smoke test you run yourself. Before anyone else sees it, before any money moves, you build the

smallest possible version and you run it end to end, by hand, on real inputs. You are not testing whether people want it yet. You are testing whether the thing can even do its one job at all. This is where the coaching product died, and it died here on purpose, because this is the cheapest possible place to find out. A test you run yourself, quietly, costs almost nothing and can save you everything.

The second step, if the smoke test holds, is a tiny handful of real potential buyers. Not a room of a hundred. Four or five people who genuinely have the problem. And the bar here is high and specific. Four out of five have to genuinely say they would pay, in a real conversation, not a polite nod at a dinner party. The polite nod is the enemy. People will tell you your idea is great because they like you and because saying so is free. That is not signal. Signal is when you ask directly, would you pay this amount for this, and they either lean in or they go vague. The vague ones are a no. Count the real yeses honestly against the line you set.

The third step, and only if the first two held, is a small, fixed, deliberately capped advertising test. I use a five hundred dollar cold test, and the rule is five hundred, never more. Both halves of that rule matter. Not less than five hundred, because below that you cannot read the signal, the numbers are too small and noisy to tell you anything, and you will draw a confident conclusion from what is basically static. And never more than five hundred, because above that you are over-committing before you have earned the right to. Five hundred dollars buys you a clean read on whether cold strangers, who do not know you and owe you nothing, will click and care and convert. If they will, you have earned the next step. If it comes back to silence, you have learned that for five hundred dollars instead of fifty thousand.

Look at the shape of the staircase. Each step costs a little more than the last. Each step is only unlocked by passing the one before. And every step has a fail line that ends it. You are never betting the farm on faith. You are buying information in small amounts, and only spending real money once the cheap steps have told you the real money has a decent chance of coming back. That is not caution for its own sake. It is refusing to pay for an expensive answer when a cheap answer is available.

The gated build

Say the tests passed and you are building the real thing. Building is where people imagine the risk finally has to become one big leap into the dark. It does not. A good build is a series of gates, and each gate has a way to stop, so you keep meeting failure early and cheaply instead of all at once at the end.

The first gate is the audit, before you build anything. Look hard at what is actually there before you start changing it. If you are rescuing or extending something that already exists, do not trust the story you were told about it, and do not trust your own first glance. Get in and look. Skipping this is the single most common reason people build the wrong thing, because they pattern-match the situation to one they have seen before, start solving that familiar problem, and only discover months later that the real problem was a different shape the whole time. An hour of looking hard at what is really there, before you touch it, saves you from confidently fixing something that was not broken while the actual break sits untouched.

The second gate is verifying the claims before you spend a cent on advertising. This one is rare and I want you to actually

do it. Before the ads turn on, someone checks every promise in the marketing against what the product actually does and where the data actually lives. Every promise. Because marketing copy almost always runs ahead of the truth, not from dishonesty but from enthusiasm, and the gap is easy to miss from the inside. A real example. The copy said your data never leaves the country. Sounded true, felt true, everyone believed it. But when we checked the claim against where the system actually sent things, part of the processing ran overseas. So the honest line was narrower. The file's contents never leave the country. Still a strong claim, still true, and now it would survive contact with a sharp customer or a regulator. Catch that before a customer catches it and loses trust, or before a regulator catches it and it becomes a real problem. The discipline is boring and it is exactly the kind of boring that saves you.

The third gate is shipping it in front of real people while it is still a bit embarrassing, and instrumenting it so you can actually see what happens, before you pour money in. The book has already made the case for shipping the rough version, so I will only add the part specific to planning the fail. Instrument it. Put in the small amount of measurement that lets you watch the real behaviour instead of your imagination of it. Where do people stop. What do they never click. Which step loses them. Without instrumentation you ship the rough version and then guess about it, which is the worst of both worlds. With it, the rough version becomes a machine for producing facts. Watch the real behaviour, not the story you would prefer.

The fourth gate has a name, and it is a law I hold to. Three failed fixes means the architecture is wrong. When you have tried to fix the same thing three times and it keeps coming back, stop patching it. The problem in front of you is not the

real problem. The whole shape is wrong, and the bug you keep swatting is just the shape complaining. I learned this the hard way, on a flow that kept breaking in a way that felt like it should be a small fix. So I made the small fix. It broke again. I made another. It broke again in a slightly different spot. By the third time, the count itself was the message. The count is the signal. Not a signal to try harder on a fourth patch, a signal to stop and change strategy, because a fourth patch on a broken shape just buys you a fifth. When you catch yourself reaching for fix number four, that is the moment to step back and admit the architecture, not the bug, is what needs to change. The number does the deciding for you, which is the whole point, the same way the kill numbers do, so that a tired and attached version of you does not get to talk you into one more patch.

Habits instead of infrastructure

A quick word on how I actually run, because it matters to the point and because the tempting version is the wrong one.

I run lean on purpose. No elaborate safety scaffolding. One live environment, not three. I ship straight to the real thing. And I do this deliberately, not because I am reckless, but because at my scale a parallel safety universe costs more to maintain than it protects. Three environments that have to be kept in sync, staging that drifts from production until it lies to you, ceremony that exists to make being wrong impossible and instead just makes being anything slow. For a large company with a hundred engineers and real blast radius, that scaffolding earns its keep. At my scale it is a tax I do not need to pay. So I do not pay it. I compensate with discipline instead of ceremony. The kill numbers, the cheap staircase, the gates,

the three-strikes law. Those are habits, and habits are lighter than infrastructure and they travel with me.

Here is the tell that keeps this honest, and it is the part I most want you to steal. Every rule I keep comes with the condition that would retire it. We do it this way until this specific thing happens, and then we change. One environment until we have paying customers whose data a bad deploy would actually harm, then we add a second. Ship straight to production until the cost of a public break outweighs the cost of the ceremony, then we slow down. Every rule has an expiry condition written next to it. They are not commandments carved in stone. They are decisions with a stated shelf life.

That matters because a rule without an expiry condition becomes dogma, and dogma is how the lean founder eventually turns into the bloated one, still running the scrappy playbook long after it stopped fitting, or worse, still carrying heavy ceremony long after the reason for it went away. Hold your rules the way you should hold your plan. As the best current answer, kept only until the thing that would change it shows up. Write the expiry condition down next to the rule. Then you will actually notice when it arrives, instead of following a rule out of habit that stopped being true a year ago.

What you have when you are done

Go back to the two people at the door.

The difference between them was never courage, and it was never talent. It was that one of them decided, in advance and in numbers, what failure would look like and what they would do when they saw it. That single act does two things at once. It makes starting easier, because a test with a known

exit is far less frightening than an open-ended bet on your own worth. And it makes stopping possible, because when the number comes back below the line there is a prior agreement to honour instead of a fresh wound to argue with.

Fear of failure is what keeps people from starting and what keeps them from stopping, and both of those are cured by the same medicine. Decide in advance, in numbers, what failure looks like. Then build so that you meet failure early and cheaply, while it is still information and not yet a catastrophe. The quiet smoke test before anyone sees it. The four out of five before you spend. The five hundred dollars and never more. The audit, the verified claims, the instrumented rough version, the three-strikes law. Every one of those is a place where being wrong costs you a little instead of costing you everything, and where the cost buys you a fact you can use.

That is what it means to plan the fail like a professional instead of praying it does not come. You are not being negative. You are not expecting to lose. You are simply refusing to be one of those people who cannot leave the wrong room because they never gave themselves permission to. You gave yourself permission before you walked in. You wrote the number down.

Now, with the fail lines drawn and the gates in place, you get to do the thing all of this was protecting. Ship the rough version into real hands, read what actually comes back, and change the plan the moment it tells you to. That is next.

20. Ship, Test, Pivot

> *The plan you started with is wrong. You just don't know which part yet. Shipping is how you find out.*

You have done the hard thinking. You mapped the context until the shadow cleared. You scoped the right thing instead of the whole wish list. You picked proven tools and left the genuinely new part small. You have a version of the product that works. Rough at the edges, but it works.

Now comes the part where most people go quiet for two years.

They stop shipping and start polishing. The thing works, but it does not feel ready. There is a screen that looks a bit off. A flow that takes one more click than it should. An edge case that fires one time in fifty. So they go back in. They fix the screen. They smooth the flow. They handle the edge case. And while they are in there, they notice three more things, because you always do, and they fix those too. The launch date slides. Nobody says it slid. It just quietly stops being a date and becomes a feeling. We'll ship when it's ready.

It is never ready. That is the trap. There is always one more thing, and the standard I described in the earlier chapters, the one that trains a technical person to chase the last four percent, does not switch off just because the product mostly works now. If anything it gets louder. The closer you get to done, the more the small imperfections stand out, and the more tempting it is to keep going in private where nobody can see the seams.

This chapter is about not doing that. About getting the thing into real hands while it is still a bit embarrassing, reading what happens instead of guessing, and being willing to tear up the plan the moment the plan turns out to be wrong. It is the least glamorous move and it is the one that separates the products that ship from the products that are forever almost done.

Let me name the pieces up front, because the order matters and most people get the order backwards. There are four. Getting it live and into real hands fast. Reading real signal instead of imagined signal. Changing direction when the signal tells you to, without ego. And the goal state you are building toward, where change is so cheap that shipping and fixing stop being separate events at all.

Ship the ninety-five. Ship it now.

I said it earlier and I will say it here because this is where it bites. Give a technical person the choice between a ninety-nine percent solution that takes two years and a ninety-five percent solution that ships in two days, and they pick the ninety-nine. Not because they are lazy. Because they are trained to. Scores, not business.

The thing nobody stops to check is whether that last four percent is worth two years. It usually is not. The ninety-five was already enough to change your life. People wildly underestimate the ninety-five.

Here is why the ninety-five is not just acceptable but better. Not a compromise you settle for. The actual right call.

The ninety-five in the market beats the ninety-nine in your head, because the ninety-five gives you something the ninety-nine never can. Contact with reality. The moment a

real person uses your product, you learn things no amount of thinking could have told you. You find out which features they ignore. You find out which problem they actually came for, which is often not the one you built for. You find out the screen you agonised over does not matter and the throwaway one they cannot live without. Every hour past ninety-five that you spend in private is an hour spent refining a guess. Every hour after you ship is an hour spent on facts.

There is a second reason, and it is one technical founders in particular need to hear. The ninety-nine percent you polish in private is polished against your imagination of the user. The ninety-five you ship is corrected against the real one. So the private ninety-nine is frequently a worse product than the shipped ninety-five, because it is exquisitely refined in the wrong direction. You spent two years perfecting a thing nobody asked for. This is the same pattern from a few chapters back, the ex-university-executive's business that wound itself up in complexity chasing marginal accuracy, and it is worth remembering that the pattern does not only kill the build. It killed the business around it.

I want to be precise about what shipping the ninety-five does not mean. It does not mean shipping something broken. It does not mean shipping something insecure, or something that loses people's data, or something that falls over the first time two people use it at once. There is a floor, and the floor is real. The floor is that it works, it is safe, and it does the one thing it is for. Above that floor, everything is negotiable, and most of what people treat as below the floor is comfortably above it. The extra login option. The settings page nobody has asked for. The animation. The report that covers the case that happens once a quarter. All of that ships later, or never, depending on what the market tells you.

Think about the difference between a presentation you build for a colleague and one you prepare to send to an external audience. Same core content. The external one gets more care around the edges. More security, more error handling, a cleaner finish. But you do not rebuild the deck from scratch, and you do not hold the internal version hostage until the external polish is done. You use the internal version now, with the people who will forgive the rough edges, and you learn from it while you tidy up. Development is no different. The ninety-five is the version you show the people who will forgive it. Shipping is how you earn the right to build the last five percent against something real.

Getting it into real hands, fast

Fast is a specific claim, so let me be specific about what it means. Fast does not mean rushed. It means you compress the distance between the thing existing and a real person using it to the smallest interval you can manage. Not a private beta that sits in a folder for six weeks while you decide it is ready. In front of a human who has the actual problem, doing the actual task, this week.

In the support automation project I described earlier, the first hands were our own team. We built the thing that handled ninety percent of the work, ran it end to end on a demo, and then ran it for weeks against real support tickets, finding the gaps, fixing them, testing again. We did not wait until it was bulletproof to let it touch a real ticket. We let it touch real tickets in order to find out where it was not bulletproof. The tickets told us things our planning had not. They always do.

There is a discipline in choosing the tool that keeps you honest here. We used n8n for that workflow, drag and drop,

because it forces you to stay light. When the tool makes it easy to over-build, you over-build. When the tool makes over-building annoying, you ship the simple version and see if it holds. Pick the tool that makes shipping the small thing the path of least resistance, and you will ship the small thing.

The other reason to ship into real hands fast is that it changes your relationship with the work. A product with no users is a thing you are protecting. Every rough edge is a source of anxiety, because you are imagining a stranger judging it. A product with real users, even three of them, even users who are on your side, is a thing you are improving. The anxiety becomes information. The rough edge becomes a ticket. You stop guarding the product and start running it. That shift is worth more than any two months of private polish, and you only get it by putting the thing in front of someone.

Reading real signal, not imagined signal

Here is where it gets subtle, because shipping is only half of it. The other half is reading what comes back, and most people read it wrong.

When you ship, two kinds of signal come back at you. Real signal is what people do. Imagined signal is what you decide their behaviour means. The gap between the two is where products die slowly, because the imagined version is almost always the more flattering one, and the flattering story is the one you want to believe.

Someone signs up and never comes back. Imagined signal says they got busy, they will return, the onboarding email will bring them back. Real signal says the product did not do the thing they came for, or did not do it fast enough for them to

see it was worth continuing. Someone uses one feature obsessively and ignores the other five you spent months on. Imagined signal says they just have not discovered the others yet. Real signal says you built one product they wanted and five they did not, and you should probably be building more of the one.

The reason the imagined version wins so often is that you are emotionally invested in the plan. You spent weeks on the feature they ignored. It is genuinely hard to look at the fact that nobody uses it and conclude that the fact is the truth and your weeks were the mistake. So you reach for the story that keeps your weeks meaningful. This is human and it is expensive.

The correction is to decide, before you ship, what you are going to measure and what number would change your mind. Not a dashboard with forty metrics. One or two things that actually tell you whether the product is doing its job. For a support tool, it might be the share of tickets that get handled without a human touching them. For a product a founder is trying to sell, it might be whether the person who signed up did the one thing that means they got value, the thing that, if they did it, they tend to stick around, and if they did not, they tend to vanish. Pick the number that maps to the real job, write down what result would tell you the plan is wrong, and then believe the number when it arrives.

I have failed at this myself, and the way I fail is instructive. We had a predictive modelling project where the ask was specific, it was in the client's budget, they had a demo of a competitor's version, every signal said proceed. So we proceeded. Somewhere in the middle I realised they had no idea what predictive modelling actually was. Not a partial understanding. None. The signals I had read as validation were real signals of something, they just were not signals of

what I thought. The demo meant they had seen a shiny thing, not that they understood it. The budget line meant someone had written a number in a spreadsheet, not that the need was real. I read the imagined version because the imagined version let me start building, which is what I wanted to do. The real version would have taken a thirty second question. Do you actually understand what this produces and what you would do with it. Not asking it cost weeks.

That is the same failure at the scoping stage that shows up again at the shipping stage. You see the signal you want. The fix is the same in both places. Decide in advance what would prove you wrong, and then be honest enough to see it when it happens.

Pivoting the moment the plan turns out wrong

Now the hardest part, because it asks something of you that goes against every instinct you have after months of work. Changing direction.

When shipping shows you where the plan was off, the only useful response is to change it. Not to defend it. Not to explain why the market is wrong. To change it.

I can say this cleanly because I have built the same category of thing enough times to know that the first plan is never fully right. I have watched it more than a hundred times, across data engineering and reporting and SaaS builds and enterprise applications, in mining and resources and financial services and government. The specifics change every time. The fact that the first plan needs correcting does not. Somebody who has built one product tends to be set in stone on the one way it worked, because for them it did work, once.

But one success is survivorship bias. A hundred million dollar company might have found the best path or might have hit the lottery, and from the inside those two look identical. The way you tell them apart is by building the flexibility to be wrong into the plan from the start, and then using it.

There is a real example of this that runs through this whole book, and it is one of my own. The Power BI tool I built for myself. We automated our own support work, reviewing files, tracking changes, writing documentation. Genuinely useful internally. So we productised it, put it in front of the market, ran ads. Crickets. The tech was cool, genuinely unique, and it turned out nobody cared, because the market was small, it took real hands-on effort to make it work, and there was no dedicated partner who owned the pain and would drive it. That is real signal. The imagined version was that we just needed better ads, a better landing page, one more feature. The real version was that the plan itself, ship a cool tool and let the market come, was wrong at the root. Nobody cares how cool your tech is if it does not make a clear pain point better for someone who feels that pain every day. Reading that signal honestly changed the plan from keep pushing the product to find the person who owns the pain and hand them most of the company. That is a pivot. It cost some pride. It was the right call.

The block that stops people pivoting is almost always sunk cost dressed up as commitment. I have spent so long on this. I cannot throw it away now. But the time you spent is gone whether you pivot or not. The only question in front of you is what produces the best outcome from here, and the answer to that question does not care how many weekends you already burned. The weekends are not an argument. They feel like an argument. They are not one.

The other block is that pivoting feels like admitting the whole thing was a mistake, and it is not. The context you built is not wasted. The tools you stitched together are not wasted. Most of what you learned digging the wrong tunnel is at least partly useful for digging the right one. A pivot is not starting over. It is turning the ship you already built onto a heading that works. The founder who can do that without flinching, who can look at a feature they loved and cut it because the market did not, who can look at a whole go-to-market and change it because the ads came back empty, that founder ships. The one who cannot is the one whose product is technically fine and commercially dead, perfectly built in the wrong direction.

The goal state: twenty-seven changes overnight

Everything in this chapter builds toward a specific end state, and I want to describe it concretely because it is the thing you are aiming at, and it is better than most founders imagine is possible.

There is a vet, a mobile vet, who spotted a real gap in the market and did the hard part well. Solid product thinking, genuine design instinct. Then it came time to build, and it went the way these things usually go. The designer tried to build it and could not, and it turned into a vibe-coded mess. A second developer took over and went in circles. Two years passed. Around thirty thousand dollars went in. And the whole time it sat at eighty percent, forever almost done, and the worst part, the part that tells you exactly what was wrong, is that at no point could anyone actually name what was broken. Not a clear failure. A fog.

We kept their design thinking, which was good, and rebuilt from the ground up. Live and solid in weeks, for a fraction of what had already been spent. But that is not the part that matters for this chapter. The part that matters is what happened after.

Before, a small change took a month. One change. A month. Now they send through twenty-seven major changes and every one of them is actioned, tested, and released overnight. Twenty-seven overnight versus one a month. That is not a small improvement. That is a different relationship with the product entirely.

That difference is the whole point of everything in this book, and it is worth being clear about why it happened, because it did not happen by accident and it did not happen because of some clever piece of technology. It happened because the product was rebuilt as a system rather than a pile of features, so that change was cheap and safe by design. It happened because the simple version shipped instead of the complete one, so there was a real thing in real hands generating real signal. And it happened because the whole setup assumed change was coming, and was built to absorb it, instead of assuming the first plan was right and breaking every time it turned out not to be.

When you get there, shipping and fixing stop being separate events. There is no big scary launch followed by two years of dread. There is a product that is live, and a stream of changes that flow through it fast, and a founder who can respond to what the market says this week and see it in the product tomorrow. That agility is the goal state. It is what ship, test, pivot is for. Not a single heroic launch, but a product that has become so cheap to change that being wrong about something stops being a crisis and becomes routine.

Twenty-seven changes overnight is not a stretch target. It is what happens when the levers before this one were done properly and the shipping move is done without ego. Ship the ninety-five. Read what actually comes back. Change the plan the moment it is wrong. Do that on a product built to be changed, and this is where you land.

Which brings me to the thing I have been careful to keep separate this whole time. Almost everything I have said assumes you are building the kind of product where speed and simplicity win, where the market will forgive a rough edge, where shipping fast beats polishing long. Most products are that kind. But not all of them. There is a whole category where the rules bend, where the buyer is not the user, where the customisation is the point and the simplicity is non-negotiable at the same time, and where a two hundred dollar rescue and a hundred and fifty thousand dollar build are answers to genuinely different questions. That is enterprise, and it is a different sport. That is next.

21. When It's Enterprise, It's a Different Sport

> *Everything you just read about shipping fast and simple still applies. It just applies differently, and getting the difference wrong will sink you.*

I have spent most of this book telling you to ship the ninety-five, keep it simple, and pivot fast. I stand behind all of it. But I would be doing you a disservice if I let you walk away thinking there is one shape to this and it is the same shape every time. There is a category of product where the rules do not break, exactly, but they bend hard enough that if you apply the startup playbook without adjusting, you will fail. That category is enterprise.

I want to be honest about how hard and how different this is, because most of what gets written about building software either pretends enterprise does not exist or pretends it is just a bigger version of the same thing. It is not a bigger version. It is a different sport played on the same field with a different set of rules, and I have played it, so I can tell you what changes.

The example I am drawing this from is a real one. A twenty-year enterprise expert, someone with a rare skillset built over two decades inside serious organisations, decided to productise that expertise. Not a lifestyle SaaS. Nine-figure ambition. We built it from zero. Full team, multiple enterprise sales, scope that shifted under us, consulting support alongside the product, a compliance track running in parallel.

The outcome was over three million dollars in the first year, built for around a hundred and fifty thousand, against what a conventional team would have spent two to three million and three to four years to produce. That number is real, and I mention it not to impress you but because the size of the spend is part of the lesson. In enterprise, a hundred and fifty thousand dollar budget can be exactly right, where for the vet's rescue it would have been insane. Different sport, different everything.

Here is what actually changes when you cross into enterprise. Four things, and each one inverts an assumption from the earlier chapters.

Stricter and more customised at the same time

Start with the contradiction that catches everyone out. In a normal product, strict and simple pull in the same direction. You narrow the scope, you nail the one thing, you keep it clean. In enterprise, the buyer wants it stricter and more customised at once, and those two pull against each other hard.

Stricter means the standards are higher and less forgiving. Security review. Data handling that satisfies a compliance team who will read the fine print. Uptime the business can depend on because real operations run on it. Audit trails. Access controls. The floor I described in the last chapter, the one that says it works and it is safe, sits much higher here, and it is not negotiable. You do not ship the ninety-five past an enterprise security review by explaining that the last five percent is coming. The last five percent is the part they are checking.

More customised means the thing has to bend to how this specific organisation actually works, which is never how the last one worked. Their process, their terminology, their hierarchy, their existing systems it has to talk to, the arbitrary way they have always done a particular thing that they will not change for you. A consumer product can tell the user how it works and the user adapts. An enterprise product adapts to the customer, because the customer is large enough and paying enough to insist.

Hold those two together and you see the problem. You are being asked to build something rigorous enough to pass strict review and flexible enough to reshape itself per customer. The instinct is to solve this the expensive way, by building everything custom for each client, which is exactly the tunnel-digging that sinks people. The way through is the systems-not-features discipline from the earlier chapters, pushed harder. You build a rigorous core once, properly, to the strict standard. Then the per-customer customisation lives in a configurable surface layer on top, not in the core. The core is strict. The surface is where the flexibility lives. Get that separation right and you can be strict and customised at once. Get it wrong and every new customer forks your codebase and you drown.

The end user is mandated, so the interface has to be invisible

This is the one people find most counterintuitive, and it is the one that matters most for how the thing gets built.

In a normal product, the person who uses it chose it. They signed up because they wanted the thing it does. That means they arrive with some motivation, some willingness to learn,

some tolerance for a bit of friction because they are getting something they came for. You can ask a little of them.

Enterprise software is sold centrally and mandated downward. The person clicking through your screens did not choose your product. Someone above them bought it and told them to use it. They have no motivation to learn it, no patience for friction, and in many cases a quiet resentment about having one more system imposed on their week. If your interface asks anything of them at all, they will resist, they will do it wrong, they will complain up the chain, and the buyer who championed you will start to look bad for having picked you.

So the interface has to be dead simple. Not simple as a nice-to-have. Simple as a survival requirement. Simpler, often, than a consumer product, because a consumer product gets to assume a motivated user and an enterprise product cannot assume anything. Every extra field is a place someone gets confused. Every extra click is a place someone drops out. Every clever bit of interface that a motivated user would enjoy is a wall to a mandated user who just wants to finish the task and get back to their actual job.

This creates a strange shape. Underneath, the product is more complex than a startup product, because of the strictness and the customisation. On the surface, it has to be simpler, because of the mandated user. The complexity all has to be hidden. The hard engineering goes into making a genuinely complicated system present as a handful of obvious screens that a resentful, unmotivated, untrained person can get through without thinking. That is a much harder job than making a beautiful interface for someone who wants to be there, and it is where a lot of the real work goes.

The real work is compressing a career into clean software

Here is the part that surprised me most, and it is the heart of what makes an enterprise product like this hard in a way that has almost nothing to do with technology.

The value of the twenty-year expert's product was not in the code. It was in twenty years of hard-won judgment about how a particular kind of work should be done. Two decades of knowing what matters, what to check, what goes wrong, what good looks like, what the shortcuts are and which ones are safe. That knowledge lived in one person's head, accumulated slowly across a career, most of it never written down because it never needed to be. They just knew.

The build was the act of getting that knowledge out of the head and into software. Not just recording it. Compressing it. Taking a career's worth of context and turning it into a system clean enough that someone with none of that experience could follow it and get an expert result. That is a genuinely difficult thing to do, and it is not a coding problem. It is a translation problem, the same translation problem that runs through this whole book, but at maximum difficulty. You are translating twenty years of tacit expertise into clean, simple, mandated-user software, and most of the expert's knowledge is the kind they cannot easily articulate because they stopped noticing they knew it years ago.

This is why the expert is essential and cannot be replaced by the builder. I could build the system. I could not supply the twenty years. The whole thing only worked because the expert owned the knowledge and I owned the compression, and neither of us could have done it alone. The reason it built for a hundred and fifty thousand instead of two to three million was not that we cut corners. It was that we spent the effort in

the right place, on getting the expert's judgment cleanly into a flexible system, instead of on custom-building every feature and chasing marginal polish. The expense was matched to where the value actually was.

You clear arbitrary hoops the buyer invents on the spot

The last difference is the one nobody warns you about, and it is the one that will test your patience.

Enterprise buyers invent requirements. Not from a spec, not from a real need, but on the spot, in the room, because someone senior asked a question and now there has to be an answer. A hoop appears that was not there yesterday and has no clear reason to exist, and you have to clear it, because the deal runs through the person who invented it. It might be a specific compliance document nobody actually reads. A feature one stakeholder wants because a competitor mentioned it. An integration with a system they are about to retire. A way of doing something that makes no sense but is how they have always done it and they will not be moved.

You can be right that the hoop is pointless and it will not matter. In a startup you can tell the user their request is wrong and often you should, because you are protecting the simplicity of the product. In enterprise, the hoop is part of the sale, and the sale is the point. The skill is not winning the argument about whether the hoop should exist. The skill is clearing it cheaply, without letting it corrupt the core of the product, and moving on. This is another reason the systems-not-features discipline matters so much here. If your architecture is flexible, an arbitrary hoop is a surface-layer change you make in an afternoon. If your architecture is rigid,

every invented requirement is a crisis. The flexibility is not just for the pivots you choose. It is for the hoops you did not.

Different goal, different spend, different funding

Step back and the whole shape is different, which means the numbers around it are different too, and this is where founders coming from the startup world get it most wrong.

The goal is different. A consumer product is often chasing scale, many small customers, speed to market, viral spread. An enterprise product like this is chasing a small number of large, high-value contracts, where one sale can be worth more than thousands of consumer signups. That changes what you invest in. Getting the product right for a handful of serious buyers is worth more than getting it in front of a million casual ones.

The spend profile is different. This is where a hundred and fifty thousand dollar budget is the right budget, not the reckless one. For the vet's rescue, spending anywhere near that would have been madness, because the goal was a fast simple product for a small market and the whole win was doing it for almost nothing. For an enterprise product aiming at nine figures, with strict standards to meet, real compliance to clear, and contracts worth millions on the line, a hundred and fifty thousand to get to over three million in year one is a bargain. Same builder, same principles, wildly different appropriate spend, because the goal set the number, not the other way around. Match the spend to the goal. A rescue and an enterprise flagship are different goals, and the honest answer to what should this cost is different for each.

The funding is different. A startup is often funded on speculation, money in ahead of revenue, betting on a future that has not arrived. An enterprise product can frequently be funded by the enterprise sales themselves, revenue coming in as you build, the contracts paying for the work. That is a healthier position and it changes how you make decisions, because you are not burning a fixed pile of speculative cash against a clock. You are building against real committed demand. Not every enterprise play works this way, but when it does, it is a materially different and safer game than the startup default.

Being honest about how hard this is

I do not want to leave you thinking enterprise is just the same thing with bigger numbers and you should go chase it. It is harder. The strictness is real and unforgiving. The compression of a career into software is genuinely difficult and depends entirely on having the right expert who owns the knowledge. The mandated-user simplicity is a harder engineering problem than it looks. The arbitrary hoops will test your patience past its limit. And the sales cycles are long, the stakeholders are many, and the scope shifts under you while you build.

If you are the twenty-year expert with the rare skillset and the nine-figure ambition, this is your game and it is worth playing, and the principles in this book still hold, they just get applied at a higher standard with a bigger budget matched to a bigger goal. If you are a founder with a smaller idea and a smaller market, most of this chapter is a warning, not an instruction. Do not build enterprise-grade rigour into a product that does not need it. Do not spend a hundred and fifty thousand where two hundred dollars and a fast rescue

would do. The whole point of matching spend to goal is that you have to know which goal you actually have.

Knowing which sport you are playing is the first decision, and getting it wrong in either direction is expensive. Build a startup product with enterprise heaviness and you drown in complexity nobody asked for. Build an enterprise product with startup lightness and you fail the first security review. The principles are the same. The application is not. Judgment about which one you are in is the thing that keeps you out of the ditch on either side.

And that judgment matters because the failure modes are quieter than you would expect. Almost nobody fails at either sport in a single dramatic moment. They fail slowly, from an accumulation of small things that nobody stacked simplicity against, in a fog where nobody can point at what is actually broken. That slow, foggy, almost-done death is the same whether you are building for a vet or for a global enterprise, and it is the most common way products die. It is worth understanding exactly how it happens, so you can see it coming. That is the last thing I want to show you.

22. The Small Things That Kill Products

> *It is almost never a big dramatic failure. It is a hundred small ones that nobody noticed adding up, until one day the product is dead and nobody can say exactly when it happened.*

You expect products to die the way things die in stories. A moment. A crash. The big investor pulls out, the key developer quits, the competitor launches the thing you were building. Something you can point at and say, that. That is when it went wrong.

It almost never happens that way.

The way products actually die is quieter and stranger than that, and because it is quiet, it is much harder to prevent. There is no moment. There is no crash. There is a slow accumulation of small things, none of them fatal on their own, none of them worth stopping the whole project over, until one day you look up and the thing has been at eighty percent for two years and nobody can point at what is actually broken. It is not failing. It is not succeeding. It is just always almost done. That is the worst place to be. Not a clean failure you can learn from and move past. A fog you cannot find your way out of because you cannot see the walls.

I want to show you exactly how this happens, because once you have seen the mechanism, you can catch it early, and catching it early is the whole game. The vet's product I mentioned lived here for two years before we got to it. Around

thirty thousand dollars and two years, forever eighty percent, and the detail that tells you everything is that at no point could anyone name what was wrong. That is the signature of this kind of death. Not a problem you can describe. A general sense that it is not right, that it is taking too long, that every fix seems to create two more, and no single thing you can put your finger on.

How the accumulation happens

Let me walk you through the actual mechanism, step by step, because it is not mysterious once you slow it down. It is a chain of individually reasonable decisions that add up to an unreasonable place.

It starts with a small shortcut. Not a bad one. A sensible one, under pressure, at the time. A thing built custom because it was quicker in the moment than finding the proven tool. A structure that was fine for the three cases it needed to handle on the day it was written. A decision to skip the systems thinking and just build the feature, because the feature was due and the system was abstract. None of these is wrong. Each one is the kind of call any competent person makes on a normal Tuesday.

Then a change comes, because changes always come, and the shortcut that was fine for three cases now has to handle a fourth it was never shaped for. So you build around it. Not through it, around it, because going back to fix the original would take time you do not have and the thing around it is quicker. Now there are two things where there should be one, and the second exists only to paper over the first. Another change comes. You build around that too. The structure grows a layer, then another, each one reasonable, each one solving

the immediate problem, each one making the next change slightly harder than the last.

This is the tunnel problem from the earlier chapters, playing out slowly. You dug a tunnel in a direction that was fine at the time. The direction turned out to be slightly wrong. Instead of climbing out and digging the right one, which felt wasteful, you kept extending the tunnel you had, bending it a little each time to head somewhere closer to right. After enough bends, the tunnel is a maze, and the maze is the product, and nobody can hold the whole shape of it in their head anymore, because it was never designed. It accreted.

The reason nobody stops it is that no single step is worth stopping for. That is the trap in one sentence. If the product broke, you would fix it. If a change took a month when it should take a day, you might notice. But each individual small thing costs you an hour, a slightly clumsier bit of code, one more special case. An hour is not worth a crisis meeting. So no crisis meeting happens, and the hours pile up invisibly, one reasonable decision at a time, until the sum of them is a product that takes a month to change and nobody can say why.

Why nobody can point at what is broken

The reason this state is so hard to escape is worth sitting with, because it is the specific thing that makes it worse than a clean failure.

When a product fails cleanly, you know what to do. The payment system does not work, you fix the payment system. The feature everyone hated, you cut it. There is a target. You can aim at it.

In the accumulation death, there is no target. Nothing is broken. Everything works, sort of, most of the time. The product does technically do the thing. It is just slow to change, fragile when you touch it, confusing to work in, and vaguely wrong in a way that resists description. Ask the developer what is broken and they cannot tell you, because nothing is broken, exactly. Ask the founder and they say it just is not right, it is taking too long, it does not feel finished. Neither of them is lying. Neither of them can point at the thing, because the thing is not a thing. It is the sum of two hundred small things, and a sum is not something you can point at.

This is why the standard responses make it worse. The founder, unable to name the problem, concludes the developer must be the problem, and demands more oversight, more updates, a developer in front of them because that is the only way we will fix this. I have received that exact email, and it was a ten second fix underneath, but the escalation assumed a big hidden problem when the reality was a scatter of small ones. More oversight does not clear the fog. It adds meetings, which add nothing, and the small things keep accumulating underneath the meetings. The developer, meanwhile, unable to point at the problem either, keeps building around it, because building around it is the only move that produces visible progress, and everyone is desperate for visible progress. So the exact behaviour that caused the fog gets applied harder as a cure. The maze grows. The fog thickens.

It is a process problem, not a technology problem

Here is the thing I most want you to take from this chapter, because it is the thing that reframes the shame most stuck founders are carrying.

When you are in the fog, the story you tell yourself is a technology story. The tech is too hard. The developer was not good enough. I picked the wrong stack, the wrong agency, the wrong platform. If only I had a better technical person, this would be solved.

It is almost never a technology story. It is a process story. You saw this earlier in the book, the point that a stuck product is rarely the sign of a bad developer. It is competent people who never had the discipline that prevents accumulation. They built everything custom instead of using proven tools. They optimised the wrong details. They built features instead of systems. They assumed the first plan was right and broke a little every time it turned out not to be. Every one of those is a process failure, not a skill failure, and you could hand the same code to a genuinely excellent developer and if they carried the same habits, they would produce the same fog.

The vet's product was not stuck because the problem was too hard. It was not too hard. We rebuilt it and it was live and solid in weeks. It was stuck because a designer tried to build it and produced a vibe-coded mess, and then a second developer took over and went in circles, and neither of them had the process that keeps a build from accreting into a maze. The technology was never the constraint. The technology is almost never the constraint. It is the process, and process is exactly the thing that gets skipped because it is invisible, because it does not produce anything you can demo, because in the moment it always feels faster to just build the next thing.

This is the single most freeing idea in this book for a stuck founder, so I will state it plainly. The reason you are stuck is

not that you are not technical enough to have built it yourself. The reason you are stuck is that the build lacked a process that stacks simplicity and plans for change, and that process is learnable, hireable, and repeatable. Your developers were not idiots. They were missing a discipline. The discipline is the whole thing.

How to avoid it

You cannot prevent every small thing. You are going to take shortcuts, build around a decision or two, ship something that is not perfectly clean, and that is fine, because the goal is not a perfect build, the goal is a build that does not accumulate faster than you clear it. So the defences are not about never taking a shortcut. They are about keeping the accumulation visible and keeping the sum small. Here is what actually works.

Build systems, not features, wherever you reasonably can. This is the biggest one, because it attacks the accumulation at the root. A feature is a specific answer to a specific question, and every new question needs a new feature built around the old ones. A system is a flexible frame that generates many answers from one underlying logic, so a new question is a surface change, not another layer in the maze. The vet's product went from one change a month to twenty-seven overnight for exactly this reason. It was rebuilt as a system, so change stayed cheap, so the accumulation stopped. When change is cheap, small things do not pile up, because you fix them as you go instead of building around them.

Use proven tools instead of inventing your own. Roughly a third of most software is genuinely unique to the problem. The rest is the same bones every product has, and those bones already exist, built and tested by people who spent years on

them. Every time you build a custom version of a solved thing, you take on a shortcut that will need building-around later. Custom-everything is the fastest road into the fog. Stitching proven tools together and spending your custom effort only on the part that is actually unique is the slowest.

Plan for the fail. Assume you will be wrong about something, and build so that being wrong is cheap. This is the anti-lottery from the earlier chapters. Somebody who built one product one way is set in stone on that one way, but a single success is survivorship bias, and building rigid because it worked once is how you guarantee the fog the second time. Keep the stacks flexible. Keep the core clean and the customisation on the surface. When you assume change is coming and build to absorb it, the small things that would otherwise accumulate get flushed through instead of trapped.

Watch the cost of change, not the state of the code. This is the practical early warning, the one signal that cuts through the fog while you can still act on it. You cannot see the accumulation directly, because it hides in a hundred places. But you can see how long a small change takes, and that number tells you the truth. When a change that should take a day starts taking a week, that is the accumulation talking, and it is talking early, while the maze is still small enough to fix. The founder who tracks the cost of change catches the fog in month two. The one who only tracks whether the product works catches it in year two, when it is a maze and the money is spent and nobody can name what is wrong. Same product. Different founder. The difference is which number they watched.

Keep the person who owns the process close. The vet had good design thinking and the wrong builders, and no process discipline anywhere in the loop, and that is why it fogged. The fix was not a better developer in the abstract. It was someone

whose actual job was stacking simplicity, using proven tools, and building for change, applied from the start. That discipline is a role, not a personality trait, and if nobody in your build owns it, the accumulation has nothing standing against it, and it will win, because entropy always wins when nothing is holding it back.

The fog, and the way out

I want to be clear about why the fog is the trap it is, because it is not obvious and it changes how you should feel about your own situation.

A clean failure is recoverable. You learn what did not work, you take the lesson, you move on, and often the next attempt is better for it. A running success needs no rescue. The fog is worse than both, because it is not a failure you can learn from and it is not a success you can build on. It is a limbo that consumes time and money indefinitely while producing a thing that is always almost done and never actually done. It is expensive precisely because it never gives you the clean signal to stop, so people stay in it for years, pouring in money against a target they cannot see, waiting for a finish that the process they are using cannot produce.

The way out is not more effort inside the fog. More effort inside the fog thickens it, because it means more building-around, more layers, more meetings about the layers. The way out is to stop, recognise that the problem is process and not technology, and rebuild the process, which usually means rebuilding on top of what was good, the design thinking, the domain expertise, the real understanding of the problem, while throwing out the accreted maze underneath it. That is what we did for the vet. Kept the good thinking. Threw out the fog. Rebuilt as a system, with proven tools, planned for

change. Weeks, not years. Twenty-seven overnight, not one a month.

If you are in the fog right now, reading this, the thing I want you to take away is that it is not a verdict on you or on your idea. Your idea can be good and your understanding of the problem can be right and you can still be stuck in the fog, because the fog comes from the process of the build, not the quality of the vision. The vision is probably fine. The process was missing. And a missing process is the most fixable problem there is, because it is not about being cleverer or more technical or luckier. It is about building the right way, on top of the good thinking you already have, with someone who owns the discipline that keeps the small things from piling up.

That is the whole method, really, seen from the failure end instead of the success end. Ship simple. Use what works. Plan to be wrong. Watch the cost of change. Keep the process close. Do those things and the small things never get the chance to accumulate into a fog. Skip them and no amount of talent or money or technology will keep you out of it. It was never a technology problem. It was always a process one. And that, at last, is the good news, because a process is something you can get right, starting now, with the thing you have already built and the expertise only you have.

23. AI, and What Actually Changed

I have to be honest about something before this chapter starts. Most of what gets written about AI is either hype dressed up as insight or fear dressed up as caution, and neither is much use to you. What I want to give you instead is the most accurate picture I can of what changed, for someone in your position, trying to turn what you know into a real product.

I will also say this is the fastest-moving area I have ever worked in. Something I write today may be partly wrong by the time you read it. What I am confident will hold is the underlying logic, because the logic is not about specific tools or models. It is about how decisions get made, and that changes far more slowly than the technology does.

Here is the short version, and then I will earn it. AI made it easier to ship the simple thing. It also made it easier to generate an enormous amount of noise. And it widened the gap between the people who can lead a build well and the people who cannot, because it multiplies whatever you already are. If you were pointed in the right direction, it made you dramatically faster. If you were pointed in the wrong direction, it got you to the wrong place dramatically faster, and left you more tangled when you arrived.

What it is

AI is a blanket term covering a pile of tools, some of which have existed for decades and some of which are genuinely new. What changed is not the underlying logic. It is the availability, the low barrier to entry, and the quality of output you can now get without a serious technical setup.

The simplest honest description is that AI is estimation at scale. If you want to guess how tall someone is, you gather related information, their weight, their shoe size, their arm span, plot it, draw a line, and make a guess. Machine learning is getting more sophisticated about how that line gets drawn. The current generation of language models does this across language, at a scale and a level of polish that produces output that genuinely feels extraordinary. It does not change what it is underneath. Good guesses, made fast, across a wide range of inputs.

Why that matters to you in practice is confidence. When you run a traditional model you get a validation process. Backtesting, accuracy scores, an explicit sense of where it works and where it fails. The current tools do not come with that built in. They produce output that looks authoritative whether it is right or wrong, in the same confident tone either way. That is both what makes them so useful and what makes them dangerous when the stakes are high and you cannot tell the difference.

It got easier to ship the simple thing

This is the genuinely good news, and it is real. The distance between an idea and a working first version has collapsed. A person who has never written code can take a clear picture of what they want, run it through the right tools, and get something that works and looks close to what they imagined. Work that used to take a developer a week can happen in an

afternoon. First drafts, first passes, rough working versions of simple things, all of it is faster and cheaper than it has ever been.

For the expert with an idea, this changes the early part of the journey in a way that is worth taking seriously. The cost of testing the smallest version of your assumption has dropped close to nothing. The thing I keep telling you to do, build the simple version that already changes things and get it in front of real users, is easier to do now than at any point in my career. If your product genuinely is simple, well-scoped, and clear, the tools can carry you a long way toward it, fast.

I want to be straight that this is not the whole story, because the same drop in cost that helped you also helped everyone selling you something, and it opened a trap that is hard to see until you are already inside it.

It got easier to generate noise

The barrier to entry falling is not only good news. It means an enormous amount of output can now be produced by anyone, at almost no cost, and a lot of it looks finished without being real.

I have had the same conversation over and over in the last while. Someone comes to me and says I was moving so fast at the start, and now I have been stuck at ninety percent for months. Every small change takes forever, breaks something else, and I cannot see a way through. In almost every case the same thing has happened underneath, and it is a direct product of how easy the tools made the early part.

What gets built this way is a collection of single features that do not fit together. Each one was produced in isolation. Each one does exactly what it was asked to do. None of them

were built with the others in mind. The underlying structure is not a structure at all. It is a series of individual answers to individual prompts, stitched together until something that looks like a product appears. Then the changes start getting slower, the fixes start breaking other things, and eventually even the tools themselves get lost in it, because there is no coherent shape for anyone, human or machine, to reason about.

The useful way to think about the current tools is as a fast, capable junior who does exactly what you ask. Give a junior a small task with a clear brief and they will nail it. Start adding real complexity, systems that have to talk to each other, state that has to be held across the whole thing, edge cases that need an understanding of the whole rather than the part, and the junior gets stuck. That is not a criticism of juniors. It is a description of where experience earns its keep. The tools are somewhere between a capable junior and a mid-level developer depending on what you ask, and they are climbing that scale fast. But right now, past a certain complexity, you need a human who can hold the whole picture in their head, and the tools cannot tell you where that line is. They will confidently keep going straight past it.

So the same technology that made it easy to ship the simple thing also made it easy to build yourself into a corner that looks, from the outside, exactly like real progress. Fast at first, then slower, then stuck, and by the time you feel it you are already tangled. The trap is not that the tools are bad. It is that they are good enough to get you a long way down the wrong road before the road runs out.

The developer thing everyone gets wrong

There is a version of the AI argument that sounds compelling and is mostly wrong. It goes: AI makes developers faster, so you need fewer of them, so your costs go down. I have watched people build whole plans around this. It does not survive contact with reality.

A good developer still costs what a good developer costs. Their rent did not fall because the tools got better. What changed is not the price. It is the output that is possible from the same person with the right setup. A good developer with genuine AI fluency, working from a clear brief, is not ten percent faster. In the right conditions they operate at a different category of pace entirely, many times faster than the same person was a couple of years ago, and I have watched it happen rather than read about it.

So the opportunity was never cost reduction. It was output multiplication. The person you are already paying can now build what used to need a team. But that multiplication only appears when three things are true at once. The right person, paid well enough that they are not distracted or already looking elsewhere. The right tools, properly set up and integrated into how they work rather than bolted on. And a genuinely clear scope, a specific and well-defined problem with success criteria both sides have agreed on, not a vague direction to explore.

When those three line up, the multiplier is enormous. When any one is missing, it collapses, and here is the part that should get your attention. AI does not fix a broken process. It accelerates it. Most builds run on a vague brief and a lot of iterative guessing, and that already worked slowly. Point the tools at that and you now have a faster path to the wrong place. Working inefficiently faster is not a productivity gain. It is a more expensive version of the same problem, arriving sooner.

That is the whole reason this widened the gap rather than closing it. The person who could already lead a build, hold the scope clear, keep the thing simple, bring in the right person and point them precisely, got handed a multiplier. The person who could not got handed the same multiplier pointed at a mess, and the mess grew faster. Same tool. Opposite outcome. The variable was never the tool. It was the hand on it.

What is working

I will be straight with you. I have not seen much custom AI deployment that genuinely works yet, and the out-of-the-box tools are not great for a lot of uses either. What I have seen work is people learning what the tools are genuinely good at, which is different from what they are sold as, and then reshaping their work around that.

What they do well: drafting, summarising, reformatting, first-pass reviewing, generating options, handling repetitive language work at volume. What they do not do well: anything that needs real judgment, institutional context, accountability, or a decision where being wrong carries a serious cost.

The uses that are genuinely delivering value are almost embarrassingly ordinary. Drafting emails. Meeting notes with the actions pulled out automatically. First-pass documents a human then reviews and sharpens. Pulling together a few slides in an hour that used to take a day. Reviewing a document for gaps or inconsistencies before it goes out. None of it is impressive. None of it gets funding approved. All of it is being used, which is more than most of the impressive projects can say.

The pattern is always the same. Low-hanging, repetitive, language-based work with a brief clear enough that a fast, capable junior can take a useful first pass at it without deep context. The value is not the output. It is the time reclaimed, which the person can now spend on the five percent of their work that genuinely needs them.

How this made the book

I will be transparent about how this book was made, because it is a better illustration of what AI is good for than anything I could invent.

I used the tools heavily. What that looked like in practice was a long stretch of iterating on structure, what to cover, how to order it, how to frame ideas for different readers. I used AI to walk through it section by section and capture my thinking, and to help me develop rough notes into fuller ideas.

What did not work was using it to turn my notes into finished text. Every time I tried, the output was technically competent and completely flat. It oversimplified. It lost whatever was alive in the original. It read like a summary of what I wanted to say rather than me saying it. The words in this book are mine, all of them, and that was not the plan at the start. I assumed I would use it more directly for the writing. I was wrong about that.

What worked well was editing. Checking consistency across chapters. Finding redundancy. Testing whether a section would land for someone who had not read the earlier ones. Flagging where an argument had a gap. All of that was genuinely faster with the tools than without them.

The end result is that I wrote every word, but the process was far faster and cheaper than the traditional route. It does

not always feel like the tools did any of the lifting, because when you are the one still doing the hard thinking and the hard cutting, it can feel like you did all of it. Step back, though, and the honest estimate is that it saved me around eighty percent of the time and cost. That is the real case for AI as a tool. Not that it does the work for you or replaces your judgment or your voice. It removes the friction around the work and leaves you with more room for the parts that are yours alone. What comes out depends entirely on what you put in and how clear you are about what you are trying to make.

What this means for you

So here is where it leaves the expert with an idea and no easy way to build it.

The good part is real and you should use it. The cost of testing the simple version of your idea has never been lower. If your thing is genuinely simple and clear, the tools can get you a long way, fast, and I would rather you tried than waited.

The trap is also real and it wears the same face as progress. The tools will happily carry you past the point where they can hold your product together, confidently, without warning you, and leave you stuck at ninety percent wondering what you did wrong. You did not do anything wrong. You hit the complexity line, and the tools do not know where it is any better than you do.

The practical rule is this. Use the tools aggressively for the simple, well-scoped part. Be honest about the complexity of your specific thing. And when you feel the resistance start, when changes get slower, when fixes create new problems, when you cannot quite explain why it is not working, that is

the signal that you have crossed the line where the tools alone are not enough. Bring in someone who can hold the whole picture before you are fully tangled, not after. The earlier they come in, the cheaper the untangling, and the difference in cost between early and late is large.

That is the honest shape of what changed. AI did not remove the need for someone who can lead a build. It raised the value of that person, because it made the reward for leading well larger and the cost of leading badly faster to arrive. The technology got extraordinary. The thing that decides whether it helps you or buries you is the same thing it always was. The judgment pointing it at the right problem, keeping it simple, and knowing where the line is. That did not get automated. If anything, it is worth more now than it has ever been.

24. Hard Lessons: What This Actually Cost Me

Most business books are written from the far side of the mistakes. The author has already made them, absorbed the lesson, and now hands you the clean framework that fell out the other end. The mistakes themselves get tidied up on the way through. Reframed as growth. Given a neat arc that ends in insight and a pull quote.

I want to be more honest than that. Not because honesty is a nice brand position, but because the tidied-up version of these stories is genuinely less useful than the real one. The tidy version teaches you that mistakes lead to wisdom. The real version teaches you what the mistake felt like while you were inside it, still paying for it, still telling yourself it was going to turn around.

So here is what building seven businesses cost me, in the categories that still sting. I am putting these here for a reason. Every one of them is a place I got hurt because I did not know the pattern yet. You are about to be in some of these rooms. I would rather you walked in already knowing where the floor gives way.

The partnership I did not paper properly

I had a business partner once, and it ended the way these things end when money is involved and the trust underneath it breaks. I exited. I walked away from work I had spent years on. The details are not ones I will put in print, because there

are agreements that bind me and there are people involved who deserve their privacy. But the shape of it I can give you, because the shape is the lesson.

The most expensive thing I have ever done in business is fail to define what happens when things go wrong, before they went wrong. Not what happens when the business wins. Everyone wants to have that conversation. What happens when it does not. Who owns what. What an exit looks like in practice. How you decide when the two of you flatly disagree and neither is willing to move. Those conversations are uncomfortable to have while things are going well, and they feel unnecessary, almost rude, like you are planning for a divorce at the wedding. They are the only conversations that matter when things stop going well.

I lost years of work. I started again from close to nothing. The framework I use now for any partnership, any significant relationship, has more legal structure in it than most people think is reasonable. They tell me it is excessive. Those people have not lost years of work yet. I hope they never do. If they do, they will understand.

I bring this one up first because it is the one that maps most directly onto you. You are, potentially, about to enter a build relationship with someone technical. A developer, an agency, a partner. The instinct, when the energy is good and everyone is excited about the idea, is to keep it light and get moving. That instinct is exactly the one that cost me years. The relationships that survive are the ones where you had the awkward conversation early, while you still liked each other enough to have it kindly.

The employee I could not get rid of

I had a salaried team member who was let go from a client contract. Under the terms of the arrangement, that made them my problem rather than the client's. For six months I had a person on payroll who was not working, who understood exactly what the legal and contractual situation gave them, and who used it. It cost me six months of salary for nothing. It cost me a significant amount of legal time. And it cost me the specific kind of low-grade, always-there stress that makes everything else in your week harder to do well.

The lesson I took was structural, not personal. Salaried employment creates obligations that do not scale cleanly with performance. The probation period that is supposed to protect you rarely does the job in practice, because by the time you have gathered enough evidence to act cleanly, you are usually already past the point where acting is clean. You end up choosing between a mess now and a bigger mess later.

Everything I do around hiring now, the small starts, the contained trials, the deliberate refusal to ask anyone to burn a bridge before we both know it is real, exists because of situations like this one. It is not a philosophy I arrived at by thinking hard. It is a scar I organised my behaviour around so I would not get cut in the same place twice.

The sob story hires

I have a genuine soft spot for people who just need a shot. I have hired on that instinct more times than I should admit, and the pattern is consistent enough that I can describe it precisely, because I have lived it enough times to know the whole arc before it plays out.

Someone comes in with a compelling story. Difficult circumstances. Real talent that only needs the right

environment. A history that explains why the CV does not reflect what they are capable of. I give them the shot, because I remember needing one. The performance does not show up. The story expands to explain why it has not shown up yet. I invest more, because I am now emotionally committed to being the person who believed in them. The performance still does not show up. By the time I accept what is happening, I have spent far more than the original mistake would have cost, and the exit is harder because now there is history.

I do not think giving people a shot is wrong. I still do it. What I learned is the difference between giving someone a shot inside a contained structure and giving them one inside an open-ended one. A small engagement, a clear deliverable, a defined window. If it does not work, it costs you two weeks and you both move on with the relationship intact. The open-ended version costs you months and a friendship. I know which one I kept reaching for, because my heart is louder than my process. That is also in the graveyard.

The legal threats

I have had legal action threatened over an agreement that was vague enough that neither party knew what it covered. I have had an offshore staff member threaten legal action and demand six months of pay after being let go for genuine, documented performance problems. In both cases the real legal exposure turned out to be smaller than it felt in the moment. In both cases the cost in time, energy, and pure distraction was large regardless of how it eventually resolved.

The pattern was the same in each. A vague agreement made when the relationship was warm, which turned into a weapon the moment the relationship went cold. Specificity in an agreement is not bureaucracy and it is not distrust. It is the

thing that means a bad outcome costs you a known amount instead of an unknown one. Vagueness feels generous while everyone is friendly. It is the most expensive kind of generosity there is.

The projects that never ended

I have started more projects than I have finished. Some of them deserved to be stopped. The market was wrong, or the timing was wrong, or the original assumption did not survive contact with reality. Those I am at peace with, because stopping them was the right call and I made it.

What I am less at peace with is the projects that ran on past the point where I knew, in the honest part of me, that they were not working. They ran on because stopping felt like failure and continuing felt like persistence, and I could never quite tell the two apart in the moment. I still cannot, reliably. That distinction is genuinely hard to make in real time, from the inside, with your own money and your own pride in the mix.

I have spent real money on product builds that still do not make a dollar. I have watched the same pattern from both chairs, as the founder pouring money in and as the person on the outside watching a founder pour money in. The sunk cost trap is not a clever bias that afflicts other, weaker people. It is something I fall into, have fallen into, and will fall into again. Knowing it exists does not protect you from it. The only thing that has ever helped me is having someone outside it. Someone with no emotional stake in the project who is willing to say the plain thing that is obvious from where they stand and invisible from where I stand.

I want to sit on that for a second, because it is one of the quiet reasons a real partner is worth more than the hours they bill. It is almost impossible to see your own sunk cost. You are standing inside the tunnel you already dug. Every month you spend makes it harder to walk out, not easier, because now walking out means admitting the months were wasted. The person who can save you is almost never you. It is someone who was not there when you started digging and does not feel the loss the way you do.

The vendors

Nine in ten of the vendors I have used have landed somewhere between disappointing and genuinely terrible. I have had contractors hold a finished deliverable hostage and demand fifty percent more than we agreed before they would hand it over. I have had agencies produce work that needed to be completely redone before it could be used at all. I have spent more money than I should have testing tools that did not work, on the strength of promises that did not survive the first month of real use.

The honest reason this kept happening is not that I have bad judgment about vendors. I have decent judgment about people. It is that the vendor selection process is structurally broken in a way that favours the vendor and works against you, every time, no matter how sharp you are. You are evaluating a sales presentation, not the work itself. The work only becomes visible after you have signed and paid, and by then the power has moved entirely to their side. You have committed. They know it. The dynamic you carefully built during the sales process quietly inverts the day the money clears.

What I do now, and what I would do from the very first day if I were starting over, is pay a small amount for a small real thing before I commit to a large one. Then I treat the result of that small thing as the only signal that means anything. Not the pitch. Not the portfolio. Not the confidence in the room. The small paid deliverable, done, in my hands, is the only honest preview of what the large one will be.

I am telling you this plainly because you are going to be on the buying side of exactly this, probably soon, and the deck is stacked against you in a way that has nothing to do with how smart you are. The people quoting you are practised at the part you see and you are a beginner at the part you cannot. That is not a fair fight. Knowing it is not fair is the first protection you have.

The work I should not have taken

I have taken consulting projects I hated because I needed the revenue and the revenue was real. I have been pushed into selling work I was not comfortable selling, because a client wanted it and the commercial pressure was right there in the room. I have stayed in engagements long past the point where they were good for me or my team, because ending them felt harder than continuing and I chose the easier hard thing.

The cost of taking work that is wrong for you is never just the direct cost of doing work you do not want to do. It is the opportunity cost of the time and energy that work eats, which could have gone toward something better. It is the signal it sends your team about the standard you will hold when money is on the line, as opposed to the one you talk about. And it is the slow erosion of the position you are trying to build, because every time you say yes to the wrong thing you

make it a little harder to say yes to the right thing when it finally shows up.

I am more disciplined about this now than I have ever been. I am not perfectly disciplined and I never will be, because the pressure is real and the bills are real. But I have enough scar tissue from the alternative to know that the money from the wrong engagement is almost never worth what it quietly costs you everywhere else.

Why I am telling you all of this

I could have left this chapter out. It does not make me look good. Read straight through, it is a list of times I got it wrong, sometimes badly, sometimes more than once in the same way.

I left it in because I think the honesty is the whole point. Every mistake in this chapter came from not knowing a pattern yet, or from knowing it and letting my own impatience or soft-heartedness override it. That is what makes this hard. Not the technology. The technology, as I keep saying, is mostly fine. What is hard is the judgment, and judgment is expensive to buy on your own, because the only way to get it firsthand is to pay for the mistakes yourself, one at a time, in real money and real years.

That is the honest case for not doing this alone. Not that you could not eventually learn all of this. You could. It is just that the tuition is brutal, and it is paid in the exact currency you have the least of when you are getting started. Time you cannot get back. Money you needed for the build. Trust in yourself that takes a hit every time one of these goes wrong and whispers the old story that maybe people like you are not meant to build things.

You are meant to build things. You are just not meant to pay full tuition on every one of these lessons yourself. That is what someone who has already paid it is for.

I have made every mistake in this chapter so that, if we work together, you do not have to. And if we do not, at least now you know where the floor gives way. Walk carefully in those spots. I did not, and it cost me.

25. The Other Side of the Door

I want to do something different in this chapter than I have done in any of the ones before it.

Every chapter up to here has been about the machine. What gets people stuck, why the standard build is tuned against you, how a technical partner thinks, what the method looks like when someone runs it properly. You have all of that now. The case is made. If you closed the book here and went and did the work yourself, you would have enough.

In this chapter I want you to come and stand somewhere else with me for a while. Not the method. Not the proof. The life.

The version of your week, your inbox, your bank account, your sense of yourself in your industry, that exists a year or two after you decide to do this. And so you can see the choice for what it is, I am also going to show you the other version. The same week, in the life where you did not build it. Same desk. Same coffee. Completely different texture underneath.

I want you to hold both at once, because the gap between them is the real decision in front of you. You cannot see the decision properly until you have sat with both places it leads.

I am not going to make this sentimental. That is not how I write and it is not how any of the people I work with talk. This is just two Tuesdays, described plainly, so you can feel the difference in your gut instead of arguing about it in your head.

A Tuesday in year two

It is a Tuesday morning, sometime in the second year after the product went live.

You open your laptop. The first thing you see is a number. Not a report you had to build, not a dashboard someone promised you three months ago. A number. It is the count of people who used the thing overnight while you were asleep. It went up. It goes up most nights now, which is the part that took the longest to get used to, because for years the only way anything you knew turned into money was you, personally, in a room, saying it out loud to someone who was paying for your time.

That is the change underneath all the other changes. The product works whether you are in the room or not.

You do not open it and hold your breath anymore. In the first months you did. You watched every error, refreshed every screen, waited for the small thing that would break and prove the doubters right. That stopped. Not because the thing became perfect. Because it became stable in the way a system is stable when it was built to be moved rather than built to be finished. Something breaks, someone flags it, it gets fixed that day, and the fix does not knock over three other things you did not know were connected. You have stopped bracing.

You spend twenty minutes on your inbox. The character of it has changed, and it took you a while to notice why. There is a message from a customer describing, in their own words, the exact problem you built the thing to solve, and thanking you for it. There is a note from someone in your industry who found the product, understood in about a minute that it was built by someone who knows the work from the inside, and wants to talk about doing something together. There is a support question, and it is a real one, and it is interesting, because the boring ones now handle themselves.

None of these emails are fires. That is the thing you keep noticing. For years your inbox was a list of things going wrong and people asking you for hours you did not have. This inbox is mostly people meeting the version of your expertise that finally exists outside your own head.

Around eleven you have a conversation you would not have had two years ago. Someone wants to build on top of what you made, or use it inside their own operation, or partner on the next thing. You did not chase them. They came because the product told them who you were before you ever spoke. You ask a few questions. You tell them honestly whether it fits. Half an hour, and the conversation is about whether you want to, not whether you can. That is a different kind of conversation to be in.

In the afternoon you do something that still feels slightly illicit. You leave. You pick up your kids, or you go for a walk, or you work on the next idea instead of servicing this one, and the business does not stop because you stepped away from it. The revenue does not require your presence in the way your old work required your presence. You built an asset. Assets do not need you sitting on them.

That is the Tuesday. It is not dramatic. Nobody is popping champagne. It is quieter than that, and better than that, because the thing that changed is not one big win. It is the texture of every ordinary day.

The same Tuesday, in the other life

Now the same morning, in the version where you never built it.

You are still you. Same expertise. Same twenty years of watching the problem up close. Same certainty that you are

right about it, because you are. Nothing about your capability is different in this version. The only thing that is different is that the product never got made, because something always got in the way.

One year you were flat out. The next year the developer you tried stalled at eighty percent and you did not have it in you to start again. The year after that you told yourself you would get to it once things calmed down, and things did not calm down, because things do not.

You open your laptop. There is no number that went up overnight, because there is nothing running overnight. There is your calendar, and your calendar is the same shape it has always been. It is full of you. Every dollar in your week is tied to an hour you personally have to be present for. That is the ceiling you have been pressed against for years, and it does not move, because the only way you know to make money is to sell your time, and there are only so many hours.

Your inbox is the inbox you have always had. A prospect who wants a quote and is talking to two other people. A meeting that could have been a message. A request for free advice from someone who has no intention of paying for it. Someone asking you, again, to do the thing you have done a hundred times, for roughly the same rate you charged three years ago, because there is nothing about your position that has changed to justify charging more.

You still know more than almost anyone about your subject. The market has no way of knowing that, because you never put it into a form the market could pick up and use. The knowledge is still trapped in your head, where it has always been, doing nothing while you sleep.

At eleven you take the price-shopping call. You already know how it goes. You have had this call a hundred times. You go deep, you are generous, you hope the depth closes them,

and it does not, because depth is not the thing that closes people who met you as one option among three. They tell you they need to think about it. You write another proposal. Your coffee goes cold. Half the morning gone.

You used to think this was just how the work went. It is not. It is the natural behaviour of the position you are standing in, and the position never changed, because you never built the thing that would have changed it.

There is nothing tragic here. You are fine. The bills get paid. But there is a low hum underneath the whole day, and you have carried it so long it feels like part of the furniture. It is the hum of the thing you were going to build and did not. Late at night, in the few minutes before you sleep, you think about it. This year, finally. Then the morning comes and the inbox is the inbox and the year is the year, and the hum stays exactly where it has always been.

What is different between the two

I want to be precise about the difference, because it is easy to make it sound like a fairy tale and it is not one.

The difference is not that you got smarter. In both versions you are the same person with the same expertise. The difference is not that you worked harder. If anything the person in the second Tuesday works harder, because they are still trading every hour by hand.

The difference is that in one version, the thing you know got turned into an asset that works without you, and in the other version it stayed a service you have to keep performing in person forever.

An asset and a treadmill feel similar for the first few months. Both are hard. Both cost you time and money you

would rather not spend. The difference shows up later, and then it never stops showing up. The treadmill gives back exactly what you put in, hour for hour, and it stops the moment you do. The asset keeps running when you step off it. It compounds. The customers accumulate. The reputation accumulates. The thing gets better while you are doing something else, because you built it so that small improvements sit on the surface and do not require you to rebuild the engine underneath.

That is what equity in a real product means, stripped of the finance language. It means you own a share of something that produces value whether or not you show up. It means the ceiling on your income stops being the number of hours in your day. It means that when you take a Tuesday afternoon off, the business does not hold its breath waiting for you to come back.

Most experts never get this, and it is not because they are not good enough. It is because nobody ever helped them cross the gap between knowing the thing and shipping the thing. They stayed on the side of the door where expertise is a service you rent out by the hour, when the whole time, a few weeks of the right work away, there was a version where the same expertise became something they owned.

Hold both Tuesdays

Before you turn the page, hold both of them in your head for a minute, because the rest of this book is about the gap between them.

A year from now, in one version, the product is live. It works while you sleep. Your inbox is full of opportunities instead of fires. Your income is no longer capped at the

number of hours you can personally sell. You own something. When someone asks what you do, there is a thing you can point at, out in the world, doing the work.

A year from now, in the other version, you are the same expert you are today. Same skills, same certainty, same years behind you. The product that was on the list this year is still on the list. The hum is still there. The price-shopping call is still on the calendar. The knowledge is still in your head, where it has always been, worth nothing to anyone until you decide otherwise.

Both of those are real. Both of them happen to real people, and which one happens to you is close to entirely a function of what you do with what you have just read.

I have shown you the two places the choice leads. What I have not done yet is deal with the thing quietly stopping most people from making it. The belief that the reason they are still on the wrong Tuesday is that they are not technical enough to get to the right one. That belief is wrong, and it is the most expensive wrong thing you believe. So let me take it apart.

26. It's Not About the Tech, It's About You

I want to tell you about the time I built something good and it went nowhere, because it is the most useful thing I can show you about what matters here, and it happens to be a story where I am the one who got it wrong.

For years my consulting business did a lot of work supporting other people's data and reporting systems. Reviewing files, tracking what changed between versions, keeping the documentation straight, catching the small breakages before they became big ones. Unglamorous work. Real value. The kind of thing that is boring to describe and genuinely painful when nobody does it.

At some point I did what this whole book tells you to do. I took that expertise, the thing I understood cold, and I turned it into a product. I automated the work. I built the tool. And it was good. It did something genuinely useful, and it did it in a way nothing else on the market did. Unique technology, solving a real pain, built by someone who understood the problem from the inside.

I ran ads to it. I put it in front of the people who had the exact problem it solved.

Crickets.

Not a slow start. Not a soft launch that needed patience. Close to nothing. The technology was good and the market did not care. And the reason it did not care is the single most important thing I can teach you in this chapter, so I want to sit on it.

No one cares how good your tech is

Here is the lesson I paid for. No one cares how clever your technology is if it does not make a specific pain plainly better for a specific person who feels that pain every day.

The tool was clever. It was also solving a problem that, for the market I aimed it at, sat somewhere in the middle of a long list of annoyances rather than at the top. There was no one on the other side of it whose whole world was that problem, who woke up thinking about it, who would tell everyone they knew the moment something finally fixed it. And there was no one on my side driving it either. I did not live inside that pain anymore. I had automated it precisely so I would not have to think about it. So I had built a good answer to a question that neither I nor my target market was losing sleep over.

The technology was never the thing that was missing. What was missing was a person who owned the pain and would not shut up about it.

I want to be careful here, because this is easy to hear as discouraging and it is the opposite. The point is not that products are a lottery and mine lost. The point is that I had the ingredients backwards. I led with the tech and hoped a driven owner would appear. It does not work in that order. The driven owner comes first. The tech serves them.

The flip

Now flip it around, because this is where you come in, and this is the part I most want you to understand about your own position.

Everything I lacked in that story, you have.

You have the pain. You have watched it up close for years. You know exactly who feels it, how much it costs them, what they have tried, why the existing answers are not good enough. You know the language they use to describe it, which is a thing no amount of market research buys, because you learned it by living in the room with them. When someone finally builds the fix, you are the person who will recognise it instantly and be unable to stop talking about it, because it is your problem.

That is not a small thing. That is the rarest and most valuable thing in the entire equation, and it is the one thing that cannot be manufactured, hired, or faked. The tech can be built. Good developers are everywhere. What is scarce is the person who owns the problem so deeply that they will drive the thing through every hard patch, understand every piece of customer feedback the moment it lands, and know the difference between a real objection and noise.

My tool failed because it had the technology and no driver. Your product will work because it has the driver, and the technology is the part I can bring.

Read that again, because it inverts the story you have probably been telling yourself. You have been walking around believing you are the weak link because you are not technical. You are not the weak link. You are the essential ingredient. The part you think disqualifies you, the deep, specific, unglamorous mastery of one real problem, is the part that everything else is built to serve. The technology is the commodity. You are the scarce input.

This is the other face of the ten percent rule. Earlier I showed you that ninety percent of any product already exists and only a thin ten percent is genuinely new. The same is true of the people. The ability to build the ninety is a commodity now, everywhere, cheap. The scarce ten is the person who

owns the problem so completely that they can see the right ten percent to build in the first place. That is you. The rule is not only about which parts of the software to spend on. It is about where the value lives at all, and it lives with the expert, not the engineer.

I have seen this from both ends now. I have been the person with the good tech and no pain to anchor it, and I watched it die. And I have partnered with people who had the pain and no way to build, and watched the same kind of technology, in service of a real driver, turn into something worth millions. Same category of tool. Wildly different outcomes. The only variable that changed was whether a person who owned the problem was standing behind it.

Why the expert has to own the majority

This brings me to the part of the arrangement that people find hard to believe when I first say it, so let me say it plainly and then explain why it is not generosity, it is just arithmetic.

When I partner with someone to build their product, the expert keeps the majority. I take a minority stake. The person who brought the domain knowledge owns the larger share of the thing we build together, and I own the smaller one.

The usual reaction to that is suspicion, because it sounds like it is tilted the wrong way. The technical person, the one doing the build, should hold more, shouldn't they. Let me tell you why they should not, using the only maths that matters here.

A hundred percent of nothing is worth a lot less than a minority of something real.

That is the whole thing. If I insisted on the majority, I would be optimising for control over a business that, without

the right driver, probably becomes another one of my crickets. I have a folder of clever technology that went nowhere. I do not need more of it. What I need is the rare input I cannot supply myself, which is you, and the only way to get you is to make the deal one where you have every reason to drive it as hard as it is yours to drive, because it is yours.

If you own the majority, you behave like an owner, because you are one. You wake up thinking about the thing. You push through the hard patch instead of drifting. You bring the customer conversations and the market knowledge and the relentlessness that a minority partner, however skilled, will never bring to someone else's idea. The structure of the deal is not a nicety bolted on at the end. It is the thing that makes the product likely to work at all.

So the split is not me being generous and it is not me being soft. It is me having learned, expensively, that the driver is worth more than the build, and pricing the deal to match reality. I would rather own a smaller piece of a business that runs hot because the right person is behind it than a larger piece of one that quietly dies because I own it and nobody who cares does.

This is also the honest answer to the question you should be asking, which is: what is in it for me, and why would I trust the incentives. The incentives are aligned because the deal only works if the thing works, and the thing only works if you drive it, and you only drive it like that if it is mostly yours. I do well when you do well. There is no version of this where I win and you lose, because my share is worth nothing unless your business is worth something. Read the structure and you can see it for yourself. It is built so that the only path to my upside runs directly through yours.

The thing this frees you from

There is a weight most experts carry that they have stopped noticing, the same way you stop hearing a fridge until it switches off.

It is the weight of knowing you have something real, something the market genuinely needs, and having no way to get it out of your head and into the world. You have watched people with worse ideas and shallower knowledge get further than you, because they had a technical partner and you did not. You have watched the window on your idea narrow while you looked for the right developer, the right agency, the right stack, and found only people who quoted you a fortune, stalled at eighty percent, and left you more convinced than ever that this was not for people like you.

I want to take that weight off you here, the same way I tried to at the start. You were never stuck because you lacked the technical ability. You were stuck because you were missing one specific partner, and you had been told, by an industry that profits from the confusion, that the missing thing was inside you. It was not. The missing thing was outside you, and it is findable.

The expertise is yours. It always was. What comes next is deciding whether to keep it trapped in your head, rented out by the hour, or turn it into something you own that works while you sleep.

If that is starting to sound like something you want, the reasonable next question is how it works in practice. Not the philosophy. The mechanics. What the partnership is, who it is for, who it is not for, and what happens first. So let me lay that out, plainly, with nothing held back.

27. How the Partnership Works

I told you on the first page that at some point in this book I would turn to you and tell you what I want, and that I would flag it plainly when it arrived rather than smuggle it in. This is that part. I am telling you out loud so there is no sleight of hand.

Some of you, by now, are going to want to build the thing with me instead of building it alone. This chapter is for you. It is the plainest, most honest description I can give of what that looks like, including the parts that are less flattering, because a partnership that starts with a sales pitch is not one you want to be in.

I am going to tell you what I do, who it works for, who it does not work for, and what the first step is. No urgency, no scarcity theatre, no countdown. If it is right for you, it will still be right for you next week. If it is not, I would genuinely rather you knew that now.

What I do

The simplest way to say it is that I become the technical side of your business so you do not have to go and find it.

Not a contractor you brief and manage. Not an agency you chase for updates while wondering if you are being managed. A partner. You bring the thing the market cannot fake, which is your deep knowledge of the problem and the people who have it. I take over the build. I plan it, I ship it, I run the

technical side of it, and I keep running it as the thing grows. The role is closest to a technical co-founder, someone who owns the build the way you own the domain, with a real stake in whether it works rather than a bill to send you regardless.

That is the shape of it. You are the person who knows. I am the person who builds. Between us we have both halves of the thing that has been missing, which up to now has been split across two people who could never understand each other.

Everything in this book is how I think about the work, and it is the same way I would think about yours. Ship the simple version that already changes your life instead of chasing the perfect one that never arrives. Use proven tools instead of inventing new ones so we move fast and nothing exotic breaks. Build the thing so it can be changed cheaply, because you will be wrong about something and we will need to move. Plan for that from the start instead of pretending we will get it right first time. This is not a script I follow. It is a way of working I have run across a lot of builds, and it is what I would bring to yours.

Why there is no fixed package

I do not have a menu. I want to explain why, because in a world of tiered pricing and packaged offers, the absence of one can read as evasive, and it is the opposite.

Every project I take on is genuinely different. The goals are different. Someone building a serious enterprise platform aimed at a nine-figure outcome is in a different sport to someone who wants a clean, simple product that runs itself and pays them well. The budgets are different, because the goals are different, and the spend should match the ambition

rather than a number I picked in advance. The starting points are different. Some people come to me with nothing but a clear idea and a deep understanding of the problem. Others come stuck at eighty percent, two years and real money into something that stalled, needing a rescue rather than a build from scratch.

I have done both ends of that range. I have taken something that was a stalled mess and had it live and solid again in a matter of weeks for almost nothing, because the fix was surgical. And I have run a build that cost around a hundred and fifty thousand dollars, with a full team, multiple enterprise sales happening alongside it, and a compliance track running in parallel, because that is what the goal required and the goal was worth it. That one did over three million in revenue in its first year, built for a fraction of what a traditional team would have spent over three or four years.

You cannot put those two things on the same price card. It would be dishonest to try. A fixed package would mean either overcharging the person who needs the small thing or underserving the person who needs the big one. So instead of a package, the work starts with a conversation, where we figure out together what you are trying to build, what it is worth building it to, and what the right shape and spend are for that specific thing. The price falls out of the goal. It does not lead.

Who this is for

Let me be specific, because a partnership like this is not for everyone and I would rather tell you now than pretend otherwise.

It is for the domain expert who is stuck. Anywhere on the range. Stuck at zero, where you know the problem cold and you know it is worth solving and you have no idea how to start. Or stuck at eighty percent, where you tried, sank real time and money, and have been living for months inside a thing that is always almost done and never finished.

It is for the person who owns a real pain. You have watched the problem up close for years. You know who has it, what it costs them, and why the existing answers are not good enough. You are the driver I described in the last chapter, the essential ingredient, the one input I cannot supply myself.

It is for the person who wants to own the thing rather than rent themselves out forever. You are not looking for another consulting gig or a nicer job. You want an asset. You want the version of your expertise that works while you sleep, and you understand that building it is worth doing properly.

And it is for the person who wants a real partner, not a vendor to manage. You are relieved rather than threatened by the idea of handing the technical side to someone who will own it, because you have better things to do with your time than pretend to understand dependencies and technical debt in meetings.

Who this is not for

Just as important, and I mean this.

It is not for someone who wants a cheap contractor to execute a fully specified brief while they keep control of every decision. That is a different relationship, and there are people who do it well, and it is not me. The whole value of a partner is that they think alongside you, and that only works if there is room for them to think.

It is not for someone who is looking for a guarantee. I do not sell certainty, because certainty in this work does not exist and anyone offering it is either lying or does not understand the work. What I offer is a way of building that plans for being wrong and moves fast when it is, which is the opposite of a guarantee and far more useful than one.

It is not for someone whose idea is, underneath, a feature, or a hobby, or a thing they are mildly curious about. The driver has to be real. If you can take the problem or leave it, you are not the owner this needs, and the thing will drift for the same reason my own tool drifted.

And it is not for someone who wants me to take the majority and carry the whole thing while they watch. That is not the deal, for the reasons I laid out. If you are not going to own it and drive it, I am the wrong person, because I have learned exactly what happens to good technology with no one behind it.

I would rather tell you honestly that it is not a fit than take your money and have us both find out slowly. That is not a line. It is the single most important thing I can offer someone in my position, and I have turned down work I wanted because the fit was not there. There is a whole story later in this book about a deal I badly wanted and said no to, because the honest answer was no. I mean it when I say I would rather lose the work than sell you something that will not work.

What happens first

So here is the mechanics of starting, with nothing dressed up.

It starts with a conversation. Not a pitch, not a proposal ambush, not a sit-through-my-slides call. A scoping conversation, where the point is to figure out what you are

trying to build, where you are stuck, what it would be worth to get unstuck, and whether what I do is the right fit for it. Some of those conversations end with a clear shape for a build. Some of them end with me telling you honestly that you do not need a partner for this, or that the timing is wrong, or that someone else is a better fit. Both are fine outcomes. A conversation where I only ever say yes is a conversation you should not trust.

If it is a fit, we work out the shape together. What the simple version that already changes things looks like. What we ship first. What the spend should be for the goal in front of you. The structure of the partnership, including the split, which as you know now is built so that you own the majority and I do well only if you do.

From there it is the work. And the work is what the rest of this book has been describing, run on your problem instead of someone else's. Ship simple. Use proven tools. Build so we can move. Plan for the parts we will get wrong. Get the thing live and solid and working while you sleep, and then keep improving it on the surface without tearing up the foundation every time you want to change something.

That is the whole partnership. There is no hidden second half, no upsell waiting behind the first call, no locked module where the real method lives. You have already read the method. This chapter is just me telling you what it costs to have me run it with you, which is a conversation, and what it is worth, which is the difference between the two Tuesdays I showed you.

Planning for it going wrong

There is one more thing you should know before you decide, and it is the thing I learned the most expensive way.

Deals like this rarely go wrong at the start. They go wrong because of how they start. You have a good conversation, then another. Momentum builds, an urgent client appears, real money gets spent, and before either of you has noticed you are deep into something real with nothing written down. It always feels premature to spend money on lawyers while it is still just talk, so you put it off. Then a year in a circumstance changes, or a disagreement lands, and you reach for the document that governs what happens next, and there is no document. That is the trap, and I have been in it. I have lost real work and real years to a partnership I never put on paper properly, because papering it never felt urgent until it was far too late.

So I do the opposite now, and it is the same discipline as the rest of this book, pointed at the relationship instead of the build. The boring paperwork goes in early, on purpose, before the momentum makes it awkward to raise. We sit down at the start and think through the changes of circumstance, all of them, while it is cheap and unemotional to do so. What happens if you want out. What happens if I underdeliver. What happens if it fails. What happens if it works far better than either of us expected. What happens if your life changes and you need to step back. Each one gets thought through and written down before we are in deep.

That is what actually protects you. Not my good intentions, which you have no reason to trust on a first read. The document does. Planning for it to go wrong, in writing, at the start, is how you make sure that if it does, it is a clause and not a catastrophe.

Let me tell you the least commercial thing about me, because it is the truest thing I can offer you before you decide.

I find it genuinely hard to charge the people I most want to help. When someone comes to me stuck, standing exactly where I once stood, wanting nothing more than for one person to sit down and make the hard part make sense, everything in me wants to just do that. Left entirely to my own instincts I would help everyone for free and build products for nothing forever, because I remember too clearly wishing someone would do that for me. That is not a boast. If anything it is a flaw in me as a businessman. The reason I run this as a real business, with real prices, is not greed. It is the people who work with me, whose livelihoods depend on this thing being a business and not a charity. They are the reason I put a number on it at all. So when you do a deal with me, understand what is on the other side of the table. Not someone whose first instinct is to extract from you. Someone whose first instinct is to help, held to sensible ground by the fact that other people are counting on him. I would rather you trusted that than any promise I could make.

I have shown you the machine, and I have shown you the door. In the last chapter I want to stop talking about the work and talk about you. Specifically you, and what is on the table now that was not on the table when you opened this book.

28. The Opportunity

I want to do one last honest thing in this final chapter, and I want to name it before I do it, the same way I named the whole game on the first page.

Every book like this ends here, with the writer turning to the reader and pointing at the door. Not with a hard sell. With a clear one, because a book that has done its job has earned the right to tell you what to do next with what you have learned. This is the machine's last move, and I am narrating it while it runs, on you, exactly as I promised I would.

And knowing that changes nothing about whether it works. If you are the right reader, what comes next is going to land, not because I arranged it cleverly, but because everything I showed you across this book is real. Every number was a real number. Every failure I described was one I had. You have spent a few hours watching something true, and true things do not stop working when you can see how they are built.

So this chapter is not about me. It is about you, and what is available to you right now that was not available before you opened this book.

What happened while you were reading

Something changed over the last few hours, and I want you to register it plainly.

You walked in carrying a story about why you were stuck. The story was that you were not technical enough. That

people who know an industry but not software are not meant to build things. That the reason your idea was still sitting in a folder was a gap inside you.

That story is gone now, or it should be, because I took it apart in front of you. You are not stuck because you are not capable. You are stuck because the standard way of building software is tuned for the wrong thing, and because you were missing one specific partner, and because you were told the missing thing was inside you when it was not.

You know now that building something that works is a process problem, not a technology problem, that most of a product is proven tools stitched together well, and that the ninety-five percent that ships in days usually beats the ninety-nine that takes years. You know the ten percent rule now, that you are only ever building a tenth of the thing and the whole game is choosing the right tenth and spending on that alone. You know now that the person who owns the pain is the essential ingredient, not the technology, and that the deal can be structured so the incentives point the same way and you keep the majority. You read all of that a few hours ago. It is yours now.

The door

The door is a real, specific transition between the version of your business that exists today and the version that exists a year or two from now if you walk through it.

On the other side of it, the thing you know is no longer trapped in your head. It exists outside you, in a form the market can pick up and use, working while you sleep. You are no longer renting yourself out by the hour with a hard ceiling over your income. You own an asset, and assets compound.

The customers accumulate. The reputation accumulates. The product gets better on the surface while the foundation holds, because it was built to be moved rather than built to be finished.

And underneath the commercial part there is the quieter thing, the one nobody warns you about until they have lived it. The weight of the unfinished thing lifts. The idea you have carried for years, the one you knew was right and could never get out, is finally out. That particular pressure, the one that has been running in the background so long you stopped hearing it, switches off. People who have built the thing describe feeling lighter, not heavier, even though they just did real work, because the thing they were carrying is now standing on its own.

That is what is on the other side of the door.

Your next step

Let me close plainly, and a little bluntly, because bluntness is the honest way to end a book that has been honest the whole way through.

Here is the one thing that does not fade. You cannot un-know any of this. The picture in your head of what it takes to turn your expertise into a real product has been changed, permanently, and it will not change back. From here on you are going to notice something you did not notice before. Every time you see someone in your industry pull ahead, every time you see an expert with a product doing the work while they sleep instead of selling their hours by hand, you are going to see the structure underneath it. You are going to see who has crossed the door and who has not. You will not be able to stop seeing it.

So what you do with that is the only question left.

I cannot take on everyone who reads this and wants to build something. The work is real and it takes real hours, and a partnership is not a thing you do at volume. So I take on a small number of people at a time, and I say no to more than I say yes to, including to work I want, when the fit is not there. That is not a tactic. It is just what doing this properly looks like instead of cramming it in and doing it badly.

So you have three ways to close this book.

You can close it and book a scoping conversation with me about what you are trying to build, where you are stuck, and whether what I do is the right fit for it. Some of those end in a build and some of them end in me telling you honestly that it is not the right time or the right fit, and both are fine.

You can close it and go and do this on your own, using everything I showed you. I meant it when I said that was a completely good outcome. The method is all here. Nothing was held back. If you take it and build the thing without me, I will count that as the book having worked.

Or you can close it and put it on the shelf with the other books you underlined and meant to act on, and let the picture soften over the next few weeks until it becomes just another thing you read once. That happens too. I would rather tell you it is one of the outcomes than pretend it is not.

Here is the only honest thing I will say about timing, and it is not a countdown. The ideas in this book are sharpest right now, in the hour after you finish. That is just how it works. The clarity you have on this page is the most you will ever have, and it fades a little every day you do not act on it, whether you act with me or with someone else or on your own. So while it is sharp, picture the year in front of you if you move on it. The product architected in a month. Live and working not long after. Compounding by the time the year is

out. A position in your market that did not exist before, and a business that no longer resets to zero every Monday morning.

That is a year, on one decision, made while the idea is as alive as it is for you right now, on this page.

I have shown you the whole machine, on you, while it ran. I told you what it was on the first page and I told you again on the last. There is nothing left behind the curtain. The method is yours to keep either way.

The gap between the two Tuesdays is not abstract anymore. It is a conversation, a build, and a thing that ends up out in the world with your name on it.

The next move is yours.

> *Run your idea through the decision engine at onwardslabs.io/decide, then book the conversation at onwardslabs.io/call.*

Real Questions, Straight Answers

This part is different from everything else in the book. No argument to follow, no through-line, no method. Just the questions I get asked, by experts at different stages of being stuck, and the most useful answer I can give to each one.

The questions are written the way people ask them, not the way a textbook would clean them up. If a question sounds blunt or a little paranoid, that is because the honest version of it usually is, and the honest version is the one worth answering. Most of these are the questions people are too embarrassed to ask out loud, because asking them feels like admitting you do not know something you think you are supposed to know. You are not supposed to know it. Nobody taught you. That is the whole reason the gap exists.

I have split them into two groups. The first is for the person still deciding, trying to pick a direction before they commit real time or money. The second is for the person already in it, who has spent the money, done the work, and cannot tell if they are close or lost.

If you are still deciding

Am I being ripped off?

I am putting this one first because it is the one everybody is thinking and almost nobody says out loud. The honest answer is that you probably cannot tell on your own, and that is not a failure on your part. It is the structure of the thing.

You are evaluating people whose work is invisible to you until it is finished, using signals that do not correlate with quality. Speed. Confidence. How fluent they sound in words you do not know. None of those tell you whether the work is good, and the people quoting you know that, whether they are being straight with you or not. So the feeling that you might be getting managed is not paranoia. It is a rational response to being in a room where you cannot see the thing you are paying for. The way out is not to become technical enough to see it. It is to change the structure so the quality becomes visible early, which is what the next few answers are about.

How much should this cost?

Less than you have been quoted, in most cases, and here is the honest reason why. Around thirty percent of your product is genuinely unique. The rest, the logins, the payments, storing data, sending an email, showing a list, is solved. It has been built a thousand times and the solutions are sitting there, cheap and reliable, waiting to be used. A good build spends almost all its money on your thirty percent and reaches for the boring proven thing for everything else. A bad build rebuilds the seventy percent from scratch because custom feels more thorough, and you pay for every hour of that reinvention plus every future hour of maintaining a hand-built version of a thing that already existed and worked.

So the number is not about the size of your idea. It is about how much of the solved seventy percent someone is about to reinvent on your dime. When you get a quote, the question beneath the number is not how clever are they. It is how much of the boring part are they planning to rebuild, and the answer to that can be the difference between weeks and two years.

I will name the exception, because if I do not you should distrust the rest. Enterprise is a different sport. Selling into

large organisations means security reviews, procurement, compliance tracks, and users who are mandated to use what you build rather than choosing it. That drags cost up for reasons that have nothing to do with the product being good. A serious enterprise build can legitimately cost a hundred and fifty thousand or more, and be worth every dollar, because the goal demanded it. The point is never that everything should be cheap. The point is that the cost should be driven by your goal, not by someone running the meter on the part that never needed rebuilding.

Why have I been quoted wildly different prices for the same thing?

Because software scoping is genuinely hard, and each of those people is quoting a different thing while using the same words. One quoted the smallest version that technically counts. One quoted everything they think you might eventually want, based on things you said in passing. One added a large cushion for all the ways builds go wrong. One is doing a fixed price and has baked their risk into the number so they do not lose money if it runs long.

Ask each of them to describe, in plain language a friend could follow, what their quote includes and what it deliberately leaves out. The gaps between those descriptions will tell you far more than the gaps between the numbers. A person who cannot describe what they are and are not building has not thought about your problem well enough to be trusted with it, at any price.

Offshore or local? I keep hearing both.

Most people I talk to about offshore have either never tried it or tried it once, had a bad experience, and written off the whole idea. Both positions are understandable and neither is useful, because the truth is that offshore is not a quality tier, it is a global market of hundreds of millions of people that

ranges from the best developers you will ever work with to the worst, at every price point.

There are development hubs where the average engineer would outclass most of what you would find locally. There are also people who will take your money and produce nothing usable. The broad assumption that offshore means lower quality is not wrong in every case. It is just too broad to be a strategy. What determines the outcome is not the country. It is whether the person leading the build gives that developer the context, the clear brief, and the structure to do good work, and whether they built the relationship so a small problem surfaces as a conversation instead of a crisis. Give a great offshore developer a vague brief and low trust and you will get a mess, and you will blame the country. Give a local one the same thing and you will get the same mess. The variable that matters is almost never the map.

The hidden cost people forget when they chase the cheap offshore rate is the management time, the communication overhead, and the price of getting it wrong. Cheap with poor communication routinely costs more than reasonable with good communication, once the rework is counted. The rate you see is not the cost you pay.

How do I know if a developer is any good?

This is the question I get most, and the honest answer is that you cannot tell from the things you would instinctively look at. Not the CV, which is a history document describing tools that may already be out of date. Not the interview, where everyone performs. Not the confidence, which is often inversely related to skill.

The things that do tell you are harder to fake. Ask them to explain a recent technical decision they made. What the options were, why they chose what they chose, what the tradeoff was. A good one can do this plainly to someone non-

technical, because they have done it before and they understand it well enough to make it simple. Then ask what they would change about a system they built recently. If they have no answer, or only positive answers, that is a signal, because good developers always see things they would do differently. And then, more than any of it, ship one small real thing together before you commit to anything large. How someone works under a little pressure, how they communicate when something goes slightly wrong, is worth more than any interview. The problem is that running this evaluation well requires you to already know what a good answer looks like, which is the thing you do not have yet. That is not a trick of this question. It is the core of why leading a build alone is so hard.

Agency, freelancer, or hire someone?

Agency: faster to start, more accountability, higher cost, quality that varies, and you are one client among many. Good for a well-defined scope. Weak for anything that needs deep context about your specific business, because you will never be their most important account.

Freelancer: more flexible, lower cost, higher personal accountability, and a single point of failure. If they get sick, go quiet, or get a better offer, you are stuck. Good for contained work. Risky for the critical ongoing build your whole thing depends on.

Hire: highest cost to start, highest loyalty, builds capability you own over time. Good for your core product. Wrong if you are not yet sure what you need, because you will be paying full-time for the privilege of figuring it out.

Most people start with a freelancer or agency and hire once they understand what they genuinely need. The trap in all three is the same one this whole book is about. Each of these relationships still needs someone to own the context,

judge the work, and keep it pointed at the goal. Picking the structure does not fill that seat. Until you can afford or attract that leadership, the seat is yours, whichever box you tick.

Should I build custom or buy something off the shelf?

Default to buy. Build only when you have honestly exhausted the off-the-shelf options and can say plainly what they cannot do that you need. Custom software is expensive to build, more expensive to maintain, and slower than quoted, every time. Off-the-shelf is imperfect and needs compromise and does about ninety percent of what you want. Ninety percent today, working, beats custom in twelve months in almost every case.

The one exception is the part that is your actual advantage. Your thirty percent. The thing that makes your product yours and cannot be bought because nobody else knows what you know. Build that. Buy everything around it. The mistake that sinks people is inverting this, buying the special part and lovingly hand-building the commodity part, because the commodity part felt safer to control.

How long should this take?

A simple informational site, one to two weeks. A basic web app with accounts and a few core features, six to twelve weeks. A complex product with integrations, payments, and multiple types of user, four to eight months. Any of these can double if the brief changes, the data is messier than it looked, or the team is part-time. Almost every quote you get assumes nothing goes wrong. Something always goes wrong. The data is always messier than it looks. Budget for that, not because your builder is bad, but because the honest version of the world includes surprises and the quote rarely does.

If you were me, with this idea and this budget, what would you do first?

Talk to ten of the people who would use it before you write or commission a line of code. Not their manager. Them. Not to validate the idea, you already believe in it. To understand the problem so deeply that you know exactly what matters and what does not, in their words, not yours. Then build the smallest possible version that tests the single most important assumption. Not the vision. The one thing that, if people use it and value it, tells you the rest is worth building.

Most first builds are far too big. The discipline of building less is harder than it sounds and matters more than almost anything else, and it is the first thing that goes when excitement takes over. It is also, not by accident, one of the hardest things to do to your own idea, because you are too close to it to cut it. That is not a weakness in you. It is the reason a second set of hands who is not in love with your idea is worth so much this early.

If you are already in it

We have been building for eight months and it is still not done. Is that normal?

For a genuinely complex product with a small team, it can be. For most things, no. The question is not how long it has taken. It is whether you have something working that real users have touched. Eight months of steady progress on a working product that is still being refined is a wholly different situation from eight months with nothing anyone has used. If no version of this has ever been in a real user's hands, that is the problem, and the timeline is just the symptom.

Our developer went quiet. What do I do?

Give it forty-eight hours, then send one direct message. Something like: I have not heard from you in a few days, can

you let me know where things stand by end of tomorrow. If nothing comes back, call. Not another email. Call. If you still cannot reach them, you have a real problem and you need to start thinking about contingency. Do not catastrophise before you have tried to reach them like a human being, because people go quiet for many reasons and most are not sinister. But do not wait two weeks hoping it fixes itself either. Two weeks of silence is not a plan.

It has been almost done for three months.

Almost done is not a project status. It is a feeling. Ask for a specific list of what is left. Not a few things. Every item, with an estimate against each. If they cannot produce that list, they do not know where they are, which is the real problem. If they can produce it, you can finally have an honest conversation about time and cost. The almost done that stretches on forever nearly always means the scope was underestimated and nobody wants to be the one to say what that means. You are going to have to be the one who asks for the list. That is part of owning the seat.

Every time I ask for one thing I am told it needs five other things first.

Sometimes that is true. Software has real dependencies, and most people quoting you those extra things are being accurate, not difficult. The question is whether the dependencies are real or whether the scope is quietly expanding. Ask them to separate the two, in plain words. What do we genuinely need to do the specific thing I asked for, and what are you recommending we also do, and why. Someone who cannot separate those two is not thinking clearly about your problem. Someone who can hands you back the decision, which is where it belongs.

We have spent way more than we budgeted and I cannot tell if we are close.

Get an independent technical person, not your current developer, to assess where you stand. Someone with no stake in the answer, who can look at what has been built and tell you plainly what works, what does not, how much is left, and whether the approach is even right. This costs money and your current developer will not enjoy it. Do it anyway. Right now you are making expensive decisions with no information, and the assessment is you buying information. It is almost always the cheapest money you will spend, because it either saves the project or stops you spending more on one that cannot be saved.

A new developer looked at our code and said it needs a full rebuild. True, or are they just after the work?

Sometimes true, sometimes not, and there is a clean way to tell. Ask them to be specific about what the problems are, how long each would take to fix incrementally, and what a rebuild would cost and take against that. Then ask what they would do if it were their own money on the line. The one who is generating work for themselves gives you a vague answer about fundamental architectural problems and a strong nudge toward the rebuild. The one being straight with you gives you a specific list and an honest view of which path makes more sense, and it is not always the one that pays them more. If they cannot articulate the problems specifically, get a second opinion before you agree to tear anything up.

How do I know if my developer is working or stringing me along?

Ask for something specific by a specific date. Not can you give me an update. Can you show me the login flow working by Thursday. Someone who is working will either show you the thing or tell you in advance why it is not ready and when it will be. Someone who is not will find reasons that specific

thing is not the right thing to show you right now. The pattern of what gets built, and when, is the signal. Not the updates, not the reassurance, not the busyness. The output.

I gave detailed requirements and they still built the wrong thing. What went wrong?

Because requirements on paper are not the same as a shared understanding of the problem. Your document told them what to build. It did not tell them why, who for, what the person using it genuinely needs, or what good looks like. So they built what was written, literally, and it came out correct to the spec and wrong to the need. This is one of the most common and most demoralising things that happens, and it is almost never bad faith. The fix going forward is to review working software early and often, not documents. Show them what you mean instead of describing it, and put the end user in the room for the review, not just you. It is also, honestly, a sign of how hard the translation is, because you did the responsible thing, you wrote it all down, and it still did not carry across. The document was never the hard part. The shared picture behind it was.

I need to cut scope and I do not know what to cut.

Cut everything that is not required for the core loop to work. The core loop is the single thing a user must be able to do for the product to have any value at all. Write down every feature currently in scope and ask, for each one, if we removed this would the product still do that one core thing. If yes, cut it. You can add it back later. You cannot get back the months you spend building things nobody needed yet. The reason this is hard is that you love the features you cut, and every one of them felt essential when you added it. Doing this well usually needs someone who does not share your attachment to them.

My developer wants to rewrite everything in a different language. Do we?

Rarely. This is a common recommendation that is occasionally right and often self-serving. The real question is whether the current language is causing a genuine problem or is just not the developer's favourite tool. Real reasons look like performance issues that genuinely cannot be solved another way, or being unable to hire anyone who knows the language. It would be cleaner in the other one is not a reason. Get a second opinion from someone with no stake in the rewrite before you agree to throw away work that functions.

I just want something that works. Why is this so complicated?

Because software is complicated, and most of the people building it were never taught to protect you from that complexity. The ones who can keep it simple are usually the most experienced, not the youngest and not the ones with the most impressive vocabulary. Experience makes things simpler. Inexperience makes them more complex, because the inexperienced builder reaches for the clever custom answer where the experienced one reaches for the boring proven one. Find someone who has built things that worked and kept working for years, and ask them what they would cut. Their instinct to remove, rather than add, is the thing you are paying for.

That last question is the one underneath all the others, so I will answer it honestly one more time. It is complicated because it is genuinely hard, and because the incentives of most people you would hire push it toward more complicated, not less. Everything in this book is an attempt to hand you the judgment that cuts through that. You can build a lot of that judgment yourself, and some of you will. But there is a reason the shortest, truest answer to why is this so hard is that it is

supposed to be, and that having someone beside you who has already done it a hundred times is not a luxury. It is the difference between the two Tuesdays I keep describing. The one where a small change takes a month, and the one where it ships overnight.

The Workbooks

The book is the argument. These are the worksheets. Each one takes an idea from the method and turns it into something you can actually do this week, on your real product, not in the abstract.

You do not need to work through all of them. The most useful thing you can do is pick the phase you are sitting in right now, the one causing the most pain, and do that one carefully. Then come back to the others when you are ready. There are no right answers in here, only honest ones, and the honest ones are usually the uncomfortable ones you have been half-avoiding. Write them down anyway. A thing written down is a thing you can act on. A thing you keep in your head just circles.

A note before you start. Some of these will surface how hard this genuinely is. That is on purpose. The point is not to make you feel that you cannot do it. It is to make you see clearly where the hard parts are, so that whether you lead the build yourself or hand it over, you are doing it with your eyes open instead of finding out later.

Worksheet 1: Am I stuck at zero or at eighty?

Before anything else, work out where you actually are, because the two look nothing alike and need different first moves.

The questions.

- Have you built anything at all yet, or is it still entirely in your head and your conversations?
- If you have built something, has a single real user, not you, ever used it for its actual purpose?
- When you imagine the next step, is the problem that you do not know how to start, or that you know how to keep going and it keeps not working?
- How much time and money is already sunk into this? Write the real number, not the comfortable one.
- When you think about the project, is the dominant feeling blank uncertainty, or is it frustration with a thing that is almost there and will not close?

The checklist.

- ☐ I can say, in one sentence, whether I am stuck at zero or stuck at eighty.
- ☐ I have written down the real amount already spent, in money and in months.
- ☐ I have named the specific thing that is stopping the next step.

The decision. Which are you? If you are stuck at zero, your next worksheet is the idea. If you are stuck at eighty, do not start over on instinct. Your next move is an honest assessment of what already exists, which is Worksheet 6. Starting again feels like progress and is often the most expensive mistake available.

Worksheet 2: Getting the idea right

The idea in your head is not the product. It is the raw material. This worksheet is about pressure-testing it before

you spend a dollar building it.

The questions.

- Who exactly has this problem? Not a category. A specific kind of person you could name and find.
- How do you know they have it? Have you watched it up close, or are you assuming?
- What do they do about it today, in the absence of your product? There is always a current answer, even if it is a spreadsheet or nothing.
- What does it cost them to keep doing it the current way? In money, time, or pain.
- What is the single thing your product would let them do that they cannot do now?
- Why has nobody solved this properly already? If your answer is that nobody thought of it, dig further, because that is rarely the real reason.

The checklist.

- ☐ I have described the problem clearly enough that someone who was not in the room could understand it without asking a clarifying question.
- ☐ I have spoken to at least a handful of real people who have this problem, recently, and in their own words.
- ☐ I can state the one core thing the product must let them do.
- ☐ I have separated the thing that is genuinely mine to know from the parts anyone could build.

The decision. Can you write the problem in one clear paragraph, grounded in real conversations rather than your own certainty? If yes, move to scope. If no, you are not ready

to build, and building now will cost you the most. Go and have the conversations first. This is the cheapest phase to be slow in and the most expensive to skip.

Worksheet 3: Scoping the simple version

This is the worksheet that saves the most money and is the hardest to do to your own idea, because you love the whole thing and cutting it feels like betrayal. Do it anyway.

The questions.

- What is the core loop? The single thing a user must be able to do for the product to have any value at all. Write it as one action.
- List every feature currently in your vision. All of them. Do not filter yet.
- For each one, ask: if I removed this, would the product still do the one core thing? Mark each keep or cut.
- Of the ones you marked keep, which are genuinely part of the core loop and which just feel essential because you are attached to them?
- What is the smallest possible version that tests your single most important assumption?

The checklist.

- ☐ I have written the core loop as one sentence.
- ☐ I have a list of everything that is not in the first version, parked for later rather than deleted.
- ☐ The first version tests one real assumption, not the whole vision.
- ☐ I have resisted at least one feature I badly wanted to keep.

The decision. Is your first-version scope small enough that it feels slightly uncomfortable, like you have cut too much? If it feels comfortable, you have not cut enough. Almost every first build is too big, and the discipline of building less is harder than it sounds. If you cannot bring yourself to cut it down, sit with that resistance, because it is telling you how attached you are, and attachment is exactly what makes a first version bloat.

Worksheet 4: Building for the pivot

You will be wrong about something. You do not yet know what. This worksheet is about building so that being wrong is cheap instead of catastrophic.

The questions.

- What are you most confident about in this build? Now assume you are wrong about it. What breaks?
- Which parts of the product are most likely to need changing after real users touch it?
- Is the plan built so those parts can change cheaply, or is each change going to fight the structure?
- Where are you tempted to build something custom that a proven, boring tool already does?
- If a user asked for a small change next month, is it a surface change or would it mean surgery?

The checklist.

- ☐ I have named the assumption I am most confident about, and accepted it might be wrong.
- ☐ The parts most likely to change are the parts easiest to change.

- ☐ I am using proven tools for the seventy percent that is not unique, not rebuilding them.
- ☐ Small changes are surface changes, not structural ones.

The decision. If real signal told you tomorrow that a core assumption was wrong, could you pivot in days, or would it mean tearing up the foundation? If the answer is tearing up the foundation, the build is too rigid, and every future change will cost what the first one did. The whole difference between the product where a change takes a month and the one where it ships overnight lives in this worksheet. It is the least glamorous decision in the build and the one that quietly decides everything downstream.

Worksheet 5: Ship, test, pivot

A build that never meets a real user is not a product. It is a guess with a nicer interface. This worksheet is about the loop that turns the guess into the thing.

The questions.

- What is the smallest thing you can put in front of a real user this month?
- Who specifically will you put it in front of, and how will you watch what they actually do, not what they say?
- What would you need to see to know the assumption held? Write it before you look, so you cannot move the goalposts afterward.
- What would you need to see to know it failed? Be as specific as you were about success.
- When the signal comes back, are you set up to change fast, or will changing be a project in itself?

The checklist.

- ☐ Something real is in front of a real user, or has a date to be.
- ☐ I decided what success and failure look like before I looked at the result.
- ☐ I am watching behaviour, not just collecting opinions.
- ☐ I can act on what I learn quickly, because the build was made to move.

The decision. Are you shipping to learn, or shipping to be finished? If you are treating this as the end rather than the start of the loop, you are about to stop exactly when the useful information begins. The founders who win run this loop many times. The ones who stall run it once, hear something they did not like, and freeze.

Worksheet 6: The rescue assessment (for the stuck-at-eighty build)

If you are stuck at eighty and something already exists, do not rebuild on instinct and do not throw good money after the current path either. Assess first. This is the worksheet that stops both expensive mistakes.

The questions.

- What actually works right now? List the parts that genuinely function, honestly.
- What does not work, specifically? Not it is a mess. Name the concrete problems.
- Of the design thinking and domain knowledge already in this, what is worth keeping regardless of the code

underneath?
- Is the problem the foundation, or the things built on top of it?
- Has anyone with no stake in the answer looked at it and told you plainly where it stands?

The checklist.

- ☐ I have a specific list of what works and what does not, not a vague feeling.
- ☐ I have separated the valuable thinking from the code that happens to carry it.
- ☐ I have an independent assessment, or I have booked one.
- ☐ I have resisted both the urge to start over and the urge to keep pouring into the current path, until I have the facts.

The decision. Rebuild, refactor, or rescue? You cannot answer this honestly from inside the tunnel you have been digging, because every month spent makes walking out feel more like waste. This is the single worksheet where an outside pair of hands is not a nice-to-have. The person who can see your build clearly is almost never the person who built it, and almost never you.

Worksheet 7: The buy-versus-build gut check

Before you commission anything custom, run it through this, because the most common expensive mistake is building the part that could have been bought.

The questions.

- Is this thing your actual advantage, the part only you can make, or is it something a thousand products already have?
- Have you genuinely looked at the off-the-shelf options, or dismissed them early because custom felt more like yours?
- What can the off-the-shelf option not do that you truly need, in specific terms?
- Would ninety percent of what you need, available today, beat one hundred percent in twelve months?

The checklist.

- ☐ I have separated my unique thirty percent from the commodity seventy percent.
- ☐ I am building the unique part and buying the rest.
- ☐ I have a specific reason for anything I have chosen to build custom, not just a preference.

The decision. Are you about to lovingly hand-build something that already exists and works? If so, stop. Build the part that is yours. Buy everything around it. Every hour spent reinventing the solved part is an hour not spent on the part only you can make, and it is the part only you can make that the whole thing lives or dies on.

How to use what you just wrote

You now have a stack of honest answers about your specific product. That is more than most people ever put on paper, and it is worth something on its own.

Read back over what you wrote and notice where the honest answer was uncomfortable. Where you could not cut

the scope. Where you could not tell if you were close or lost. Where the decision needed a set of eyes that were not your own. Those spots are not failures. They are the map of where this is genuinely hard, and where the judgment of someone who has done it many times is worth the most.

Take the one thing that surfaced most sharply and act on it this week. Not the whole method. The one thing. The rest follows from starting, and starting on the real thing beats planning the perfect thing every time.

About the Author

Ryan Richardson has built seven businesses and delivered over a hundred technical projects, from funded startups to tier-one enterprises. His early career was in data science and analytics consulting, building reporting and analytics systems across major mining and resources companies, and doing technical work that touched global banks, large technology firms, and government.

He is the founder of Onwards Analytics and Onwards Labs, and the technical co-founder of an enterprise software company he helped take from an idea in an executive's head to a real product with paying enterprise customers. Across his own ventures and his clients' he has done the same thing over and over: taken a smart person who knows their industry cold, and helped them turn that into a product that ships.

He works with a small number of domain experts at a time, as their technical partner. He lives in Australia.

The Next Step

If you have read this far, you already know how this works, because I told you at the start and then showed you the whole machine while it ran on you.

Some of what is in this book you will take and run with on your own. That is a good outcome, and I mean that.

Some of you have the thing that cannot be bought. You know your industry cold. You see the product nobody else can see. And you do not want to spend the next two years learning to lead a build you could partner your way through instead.

If that is you, the next step is a conversation. Not a pitch. We talk through what you are trying to build, whether there is a real fit, and what it would take. If we are not a fit, I will tell you, the same way I would tell you a project will not work before you spend a dollar on it.

You can book that conversation here: onwardslabs.io/call

Or start by running your idea through our decision engine first, at onwardslabs.io/decide, and bring what it tells you to the call.

Worst case, you get a straight answer and never hear from me again.

Ryan