THE NON-TECHNICAL CTO

HOW NON-TECHNICAL FOUNDERS BECOME THE TECHNICAL LEADER THEIR STARTUP NEEDS

RYAN RICHARDSON

BOTH SIDES OF TECH

The Non-Technical CTO

How non-technical founders become the technical leader their startup needs.

Ryan Richardson

Contents

The Six-Lever Engine

How the levers connect. Pull one and you move the others. Fix all six and output multiplies.

1. **Context & Communication**
2. **Work Structure**
3. **The Right People**
4. **Location Arbitrage**
5. **Tools & Automation**
6. **Retention & Culture**

The levers work as a system. Fixing one in isolation produces limited results. Getting all six right produces extraordinary ones.

For the founder with a product to build and no technical co-founder to build it. For the person signing the invoices, leading the developers, and quietly hoping they are not being managed. And for everyone who has decided they will lead the technology side of their startup without first becoming an engineer.

Read it cover to cover or go straight to the lever causing you the most pain right now. Either way works.

Foreword

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

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

That dual perspective is not a credential. It is a confession. Because the honest version of that sentence is: I've had a hundred-plus opportunities to watch the same problems play out over and over again — from both sides — and I still find it genuinely maddening every single time.

This book exists because it drives me crazy. That's the real reason. Not because I sat down one day and decided the world needed another business framework. But because I have spent the better part 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's not the technology. The technology is mostly fine. It's not the people. There are talented people everywhere. It's not the budget. Most of the time there is enough money to get something real built.

Meetings about the meetings. Decks about the strategy. Status updates about the status updates. And somewhere in the middle

of all of that, the people who actually know how to build things are sitting there with their hands behind their backs, waiting for permission to do the thing everyone hired them to do.

I've seen this in startups that should be moving fast and enterprises that have every reason to get this right. In mining, in resources, in financial services, in government. The faces change. The technologies change. The size of the numbers changes. The pattern doesn't.

Then AI arrived, and I thought — maybe this is the thing that breaks the pattern. Maybe the sheer force of what this technology can do will cut through the noise.

It didn't. It made it worse. Now we have all the same conversations we always had, plus an entirely new layer of hype to wade through, an entirely new category of vendors promising transformation, and an entirely new set of projects that are almost impossible to evaluate because nobody in the room — including the people selling it — fully understands what's real and what isn't. The people who were already good at leading technical work are getting extraordinary results from AI. The ones who weren't are spending more money, generating more noise, and producing less than ever.

So here's what this book actually is

It's a vent. A long, structured vent from someone who has been on both sides of the table — who has been the client writing the cheque and the practitioner cashing it — and who has accumulated a set of hard-won observations about why this keeps going wrong and what actually makes it go right.

It is not a strict methodology. It is not a framework you should implement exactly as written and expect the same result every

time. Some of what's in here will be immediately applicable to your situation and will make a noticeable difference quickly. Some of it will be directionally right but will need to be adapted significantly to fit your context. And some of it will not apply to you at all — and that's fine, skip it, come back to it later, or ignore it entirely.

What I can tell you is that getting even half of this right will not produce a marginal improvement. The gap between founders who understand how to lead technical work and those who don't is not ten percent. It is not even fifty percent. When you get this right — when the person accountable for the outcome actually understands the system they're trying to lead — you can realistically multiply your output by fifty times. Not because the technology got better. Because you stopped spending ninety-five percent of your energy on everything except the work.

I've watched it happen enough times to be certain of that. The rest of this book is the list of things I wish someone had told me earlier. Take what's useful. Leave what isn't. And if it saves you even one product from quietly fading away into nothing, it was worth writing.

— Ryan Richardson

This Book Is For You If...

- You have an idea you've validated, people want it — and you have no idea how to actually get it built.
- You're spending money on developers or an agency and you're not sure you're getting what you paid for.
- You've been stuck at seventy percent — sometimes ninety — for months, and every fix seems to create two new problems.
- You want to finally ship the thing instead of watching it drift.
- You want to lead the technology side of your startup with confidence — to be the person who drives it — without having to become 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 startup — and your life — that is sitting on the other side of knowing how to close it.

> **The gap is costing you more than you think.**

01

The Problem

> **Four different founders. The same gap. The same problem underneath.**

You're in a meeting.

The developer is talking. Something about dependencies. Something about technical debt. You stopped understanding about thirty seconds in.

You're nodding anyway.

You've been nodding for four minutes. You don't know if what they're describing is a real problem or a very convincing excuse. You don't know if three more months is reasonable or if you're being managed. You don't know if this person is exceptional or quietly mediocre and very good at looking busy.

You say something about keeping momentum. You ask for an update Friday. You walk out still having no idea if your product is on track.

That's a bad place to be when you're the one signing off the budget. And yet here we are.

You're staring at your bank account.

Eighty thousand dollars. The agency says you're ninety percent done. They said that six weeks ago. Every time

something gets fixed, two more things appear. You're starting to think ninety percent done isn't a real number. It's just the number that keeps you paying.

But you've already spent eighty thousand dollars.

So you approve the next invoice. Again.

You have an idea.

Good idea. You've validated it. People want it. You know exactly what problem it solves.

You have no idea how to build it. You've talked to two developers. One quoted forty thousand and couldn't explain what that would actually produce. One quoted twelve thousand and something about it doesn't feel right. You're starting to wonder if the idea will still be relevant by the time you figure out how to start.

You built the thing.

It's good. You know it's good. You understand exactly how it works, what it does, how much better it is than what they had before.

They're not using it.

Four meetings. Four rooms full of nodding. Then everyone went back to the spreadsheet they've used for six years. You can't tell if they didn't understand what you built, if they understood and didn't care, or if you built the wrong thing entirely without realising it.

You're starting to think it might be the third one.

Four different founders. Same problem. You have the vision. The technical side has the capability. Nobody can move between the two.

That gap is where budgets disappear. Where products get built that nobody uses. Where good founders spend a year feeling ripped off and good technical people spend careers feeling invisible.

Nobody is the villain. Both sides are trying. Both sides are frustrated. And almost nobody has been taught to close the gap — because closing it means understanding both sides at once, and most careers are built entirely inside one.

I've been in all four of those rooms. Sometimes in the same week. Sometimes I was the problem.

This book is about closing that gap. Not making you technical. Not turning you into an engineer. Just showing you exactly where it opens, why it keeps opening there, and what fixing it actually looks like — so you can lead the people who build your product instead of hoping they lead themselves.

Let's get into it.

The baseline problem

Most teams that are underperforming technically do not know they are underperforming. Not because they are not measuring — they are measuring plenty. They have trackers and check-ins and quarterly reviews and vendor scorecards. But they are measuring against their own history, their own sense of what normal looks like. And that baseline is, in a lot of cases, a joke.

I have seen technical people who do thirty minutes of genuine productive output a day. Not because they are lazy — because the environment they are operating in has no mechanism for identifying that as a problem. The meetings fill the calendar. The status updates get written. The project moves at the pace it has always moved. Nobody is alarmed because nobody has a reference point for what alarmed should feel like.

Technical work is different from almost every other category of work in one specific way: the gap between low performance and high performance is not incremental. The same number of people, working on the same problem, with access to the right tools, the right structure, and the right context, can produce one hundred times the output of a team operating at the baseline most people have quietly accepted as normal. You will not find out in the interview. You will find out six months later.

The reason most founders never experience the ceiling is not that they lack the talent or the technology. It is that nobody has ever shown them what the ceiling actually looks like. The first thing this book asks you to do is question that baseline. That decision is the starting point for everything else in this book.

The fifteen-minute developer

There is a stereotype that developers only work fifteen minutes a day. I want to address it directly because it is one of those jokes that is uncomfortably close to the truth — and the honest version of why it is true is more useful than either defending it or dismissing it.

The reality is a blend of three things happening simultaneously in most technical environments.

The first is the system. Deep technical work — genuinely hard thinking, the kind that produces something real — cannot be sustained for eight hours. It does not work like that for anyone, and pretending it does produces an environment where developers fill the gaps between genuine output with the appearance of productivity. Meetings. Slack messages. Ticket updates. Code reviews that nobody acts on. The calendar looks full. The output is fifteen minutes of something that actually mattered. This is not a character flaw. It is what happens when you design an environment around presence rather than output.

The second is the nature of the work itself. The best technical people in the world work in short, intense, highly concentrated bursts. Two hours of genuine flow state from a senior developer can produce more value than two weeks of fragmented, interrupted, context-switching effort from the same person. Most teams have never created the conditions for that flow state to exist.

The third is honesty. Some developers — not all, not most, but enough — have figured out that the baseline is so low and the accountability so diffuse that fifteen minutes is genuinely sufficient to be considered a solid contributor. The environment has taught them, slowly and without anyone intending it, that nothing more is expected. They are not bad people. They are rational actors responding to the incentives in front of them. The incentives are wrong.

The reason this matters is that the fix is not to hire different people. The developers you already have are almost certainly more capable than the environment you have built allows them to demonstrate. Fix the system, create the conditions for concentrated output, and the fifteen minutes becomes two hours of something genuinely extraordinary.

BEFORE YOU MOVE ON

- If your technical team's output multiplied by ten tomorrow, what would that actually look like — and what is currently stopping it?
- Where in your startup are smart, capable people achieving significantly less than you know they are capable of? What is sitting between them and that output?
- Think about your current technical setup. If you had to describe the baseline honestly — not the aspirational version — what would you say?

> **Both sides think they are aligned. They aren't.**

02

How to Close the Gap

> **Most people try to fix the gap by working harder on their side of it. That is exactly why it stays open.**

Every product that goes wrong, every technical team that underperforms, every founder who cannot get their developers to build the right thing — they all have something in common. At some point, in some specific interaction, the technical side and the business side stopped understanding each other. And nobody named it clearly enough to fix it.

That gap is not random. It opens in predictable places, for predictable reasons, and it can be closed with predictable interventions. I know this because I have watched it happen hundreds of times across data engineering, reporting, SaaS builds, enterprise applications, support functions, and everything in between — on both sides of the table.

The six levers in this framework are not arbitrary categories. Each one is a specific place where the translation gap between technical and non-technical thinking opens up — where the business side and the technical side make incompatible assumptions, optimise for different things, or simply talk past each other without realising it. Fix that specific gap, and the

output from that part of the system changes immediately. Fix all six, and the whole thing compounds.

I did not design this framework. I accumulated it. Every lever came from hitting a specific pain point, finding a fix that worked, testing it on the next project, and refining it until it was reliable enough to repeat. The breadth of contexts — mining enterprises, funded startups, government data teams, offshore teams across four countries — turned out to be the thing that made the pattern visible. When you see the same gap open in the same way across completely different industries and company sizes, you stop thinking it is contextual and start understanding it is structural.

Why these six — and why in this order

Each lever maps to a place where the two sides of the gap — technical and non-technical — make assumptions that do not match. **Context and communication** is where the gap first opens: both sides think they are aligned and neither has tested that assumption. **Work structure** is where it widens: the leader is involved in decisions that belong to the technical team, or absent from ones that need business judgment. **The right people** is where it compounds: hiring for technical skills alone, without the ability to translate between worlds. **Location arbitrage** is where assumptions about quality and culture replace actual evaluation. **Tools and automation** is where the technical side optimises for capability and the business side optimises for familiarity, and neither gets what they actually need. And **retention and culture** is where the accumulated cost of all the other gaps shows up — in the people who leave, disengage, or stop bringing their full capability to work.

The core tension running through all six is a trade-off between quality of context and cost of delivery. Get that balance wrong in either direction and you fail — but in completely different ways.

The high-context trap

Some teams do context brilliantly. Every requirement is documented. Every decision goes through the right people. The senior person who understands the business is involved in everything. The work that comes out is accurate, considered, and genuinely useful. It also takes forever.

When you maximise context — when every piece of work requires full senior involvement at every stage — you create a bottleneck that is physically impossible to clear. The extreme version is the ex-CEO consultant on a five to ten thousand dollar day rate. You are buying genuine expertise. You are also paying for someone whose involvement in a project, paradoxically, often slows it down. Because when someone that expensive is in the room, everyone defers. Decisions that should be made quickly get turned into workshops.

The offshore-at-scale trap

The opposite failure mode looks very different but has the same root cause. You discover that offshore talent is genuinely good and cheap. You lean in hard. Then, six months in, something starts to go wrong. Reports are technically correct but analytically meaningless. Requirements that seemed clear get interpreted in ways that make sense from a coding perspective but make no sense from a business one.

The extreme version is the race to the bottom. You keep cutting rates, keep finding cheaper markets, keep squeezing

cost. Quality drops. Attrition rises. The rework costs exceed what a reasonably priced local team would have cost.

Most of you are somewhere in the middle

Most founders don't live at either extreme. You have a reasonable mix of senior and junior capability, some local and some offshore, some documented process and some not. And you are still significantly underperforming. Not because you are doing something catastrophically wrong. But because you are pulling on these levers without being explicit enough about it. The gap between reactive and deliberate decisions is not marginal. It is the difference between a technical function that absorbs your budget and one that multiplies your output.

On offshore — because almost everyone has an opinion

The majority of people I talk to about offshore delivery have either never tried it or have tried it once, had a bad experience, and written off the entire concept. Both positions are understandable. Neither is useful.

There are development hubs in Bangladesh where the technical capability of the average developer outstrips most of what you will find in the Australian market. There are engineers in Eastern Europe who have spent their careers working with large European enterprises at standards that would be considered exceptional anywhere in the world. The assumptions people carry about offshore — that the work will be lower quality, that cultural differences will make collaboration difficult — are not wrong in every case. They are just broad. And broad assumptions about a global talent market of hundreds of millions of people are not a strategy.

The principles required to give someone in Bangladesh the context, autonomy, and structure to run hard are exactly the same principles required to give someone in Sydney the same thing. That is what the six levers make possible. Not a mandate to offshore everything — a framework for understanding which decisions are actually connected to each other.

CONTRARIAN TAKE

Both sides are wrong about each other, and it is costing everyone

There is a conversation that happens in almost every company that has both technical and non-technical people in it. On the business side it sounds like: they never deliver on time, they always overcomplicate everything, I cannot tell if they are actually working. On the technical side it sounds like: they do not understand what they are asking for, they change their mind constantly, they have no idea how difficult this is. Both sides are partially right. Both sides are mostly wrong.

What the business world gets wrong about technical people. The default position of most non-technical founders toward their technical teams is low trust dressed up as oversight. Technical work is hard to see — you cannot look at a codebase the way you can look at a sales pipeline and immediately understand whether it is healthy. That invisibility is uncomfortable, so leaders reach for proxies. Speed. Responsiveness. The appearance of progress. None of those things are reliable indicators of technical quality and most experienced technical people know it — which means they are being assessed by standards they know are wrong and cannot easily correct without sounding defensive.

I watched a team take a process delivering two or three fixes a year at high cost and rebuild it to deliver seventy fixes at twenty percent of the cost. Output multiplied by thirty, cost dropped by eighty percent. The response from the business side was: why did that last fix take three hours when it should have taken one? That is not an unusual story. It is the default story.

I received an email once that said: there are significant issues, I need a developer in front of me as that is the only way we will fix this. It was a ten second fix. What it would have taken instead was a phone call — I noticed these issues, what do you think? We chat through the gap, I fix it while we are talking, done. Instead it came as a formal escalation demanding physical presence. That is what low trust costs in practice. Not just the relationship damage — the actual time and energy spent managing a dynamic that should have been a conversation.

What the technical world gets wrong about business people. The ego that exists on the technical side is real and rarely acknowledged honestly. I have watched technical people decide on a solution while the business person is still describing the problem. The business side is three sentences in and the technical person has already picked the approach, mapped the architecture, decided what success looks like. Nothing that comes after that moment is actually heard.

The clearest example I have seen personally: two exceptional data scientists, a year of genuinely impressive work, a model that was technically extraordinary — deeply optimised, academically rigorous. Completely commercially useless. Unstable in production. Impossible to explain to the people who needed to trust it. We replaced it with a model that had six input features. About five percent less accurate on a backtest. It ran in seconds, never broke, anyone in the business could understand why it made the predictions it made, and it actually got used. The year of work it replaced was not bad work. It was the wrong work, optimised for the wrong success criteria, defined entirely by the people doing the building rather than the people who needed to use it.

> The people who operate most effectively across this gap share one characteristic. They are genuinely curious about the other side. Not tolerant of it. Curious about it. That curiosity is not a personality trait you either have or do not. It is a decision. And it is the decision that everything in this book is ultimately asking you to make.

BEFORE YOU MOVE ON

- Think about a technical project that underperformed. Looking at it through the six levers — which one was the real problem?
- Where in your startup are you making decisions about people, tools, or structure in isolation from the other levers? What would change if you looked at them together?
- Where are you closest to the high-context trap — where the people who know the most are least available to the people who need them most?

A note for the technical person

> **This part is for you specifically.**

Most of this book is written toward the business side of the gap. That is where the budget decisions get made, which means that

is where the pain is loudest. But the gap does not open from one side. And the most unfair part of how it plays out falls entirely on you.

Here is the honest version of what happens. You build something. You apply real skill to it — the architecture, the logic, the accuracy, the design. You are judged on none of that. You are judged on whether the deliverable lines up with an expectation that was never fully articulated, by someone who does not have the context to evaluate what you actually built. When it does not line up, the credibility loss is yours. Not theirs.

That is genuinely unfair. I want to name it clearly before I say anything else, because most advice skips straight to telling technical people to communicate better without acknowledging that the standard they are being held to is not the standard they signed up for.

Here is the analogy that makes it concrete. Think about a developer handing requirements to a designer. The designer interprets the brief, does the work, delivers something. If it does not match what the developer had in their head — frustration. The designer just did not understand. Sound familiar? That is the exact same dynamic, one level further down the chain, with a larger understanding gap and higher stakes. The business hands you a brief. You interpret it, do the work, deliver something. If it does not match what was in their head, the conclusion is the same: you did not understand.

The difference is that in most organisations, only one side of this equation has consequences.

I have watched this play out more times than I can count. Someone builds something genuinely impressive. A model that runs in seconds and does something that would have taken weeks manually. A tool that solves exactly the problem it was

asked to solve. And the feedback comes back: the colour is wrong. Not the right brand colour. They cannot get past it.

You try to explain that it is a demo, that the colour is a placeholder, that the actual value is underneath the colour. They cannot see it. In their world, the colour is not a placeholder — it is a signal about whether you listened, whether you understood the brand, whether you are going to embarrass them in front of their stakeholders. They are not evaluating your architecture. They are evaluating what their boss is going to say when they see it.

I have bitten my tongue on this one more times than I would like to admit. The instinct is to explain why the colour does not matter. The explanation makes it worse. The only thing that works is to fix the colour, remove the distraction, and let them see the actual value underneath.

That is not a compromise. That is understanding what they are actually optimising for. They are not optimising for technical correctness. They are optimising for what their leads will say, what the end user will see, what they will have to defend in the next meeting. Once you understand that, fixing the colour stops feeling like capitulation and starts feeling like removing an obstacle between them and the value you built.

The tipping point

The moment a technical person crosses from being seen as a cost centre to being seen as a strategic asset is almost always the same moment. It is when they stop explaining what they built and start speaking to what it does for the person in front of them.

Not the features. The outcome. Not the accuracy of the model. The decision it makes easier. Not the elegance of the

architecture. The problem it removes from someone's week.

This is not dumbing it down. It is understanding the translation layer. The business side is not incapable of understanding technical work — they are capable of understanding exactly as much of it as is relevant to their decision. Your job is to figure out what that is and lead with it. Everything else is detail they did not ask for.

The technical people who become genuinely indispensable — the ones who get called into strategic conversations rather than just handed tickets — are the ones who have figured out this translation. They have not stopped being technical. They have added a layer on top of it. They can move between the two. That is the capability that is almost impossible to hire for and almost impossible to replace once you have it.

For the technical founder specifically

If you are a technical person building your own product, the gap does not go away — it inverts. Now you are on both sides of it simultaneously. You have to generate the brief and execute it. You have to decide what to build and build it. And the single most common mistake is optimising the build before you have validated the brief.

Technical founders build things that are technically impressive and commercially adrift. Not because they lack capability — because the skills that make you good at building are almost orthogonal to the skills that make you good at figuring out what to build. One requires depth. The other requires breadth, comfort with ambiguity, and a willingness to stay in the problem longer than feels productive.

The fix is the same as it is on the other side of the gap. Get out of your own domain earlier than feels necessary. Talk to the

people who would use it before you have built the thing they would use. Let the conversation be messy. Resist the urge to spec it and build it until you have heard the same problem described the same way by enough different people that you stop being surprised by it.

The technical skill is not the constraint. It never was. The constraint is the translation — and that is as true when you are translating for yourself as when you are translating for someone else.

> **Lead the system. Not the work.**

03

Context & Communication

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

I want to start this chapter with a story because it illustrates the problem better than any principle I could write.

An operations lead needed a tool built. He was a good leader — pragmatic, supportive, genuinely invested in making things work. And because he was a good leader, he did what good leaders do when working with a technical team. He tried to make their lives as easy as possible. He documented everything. He laid out all the options. He mapped out 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 that he had given the team everything they needed.

The technical team received it and thought exactly the same thing. They had been given a comprehensive brief. Their job was to build what was in front of them, as completely and robustly 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 touch tool he could update every few weeks as site processes changed. Something simple enough that his team would trust it immediately. Something he could launch once and not have to re-launch. 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. Robustness. Stability. Something built properly that would not need to change. So they built something comprehensive, technically sound, and almost completely misaligned with what the operations lead actually needed.

The context conversation that would have resolved 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 haven't written down yet? Once that question was asked, the operations lead answered it immediately and clearly. He was an extremely 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. Technical work is exponential in its complexity. Adding a small amount of additional scope does not add a small amount of additional work. It can double or triple the complexity of the entire system. The difference between a simple version of something and a comprehensive version is often the difference between one week and six months. Which means that doing less — but doing the

right less, precisely defined and clearly understood — is almost always the highest-leverage decision available on any technical project.

Tech needs to exist 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 in a project 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 partially 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. Going back to the operations lead example — once the real requirement was clear, the pattern underneath it became obvious. The system needed to change frequently. 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 decreases as the system matures — because changes are surface level only. Quick, safe, and easy.

The AI example makes this 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 toward your specific use case. When something needs to change, you change the layer that needs to change — usually the prompt — without touching anything else.

THE HOW — 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 a specific 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 partially 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 critical thing — and this is counterintuitive if you have a bias toward getting started quickly, which I absolutely 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 comprehensive interrogation — one specific question, asked at the right moment, that surfaces the assumption that was about to become an expensive mistake.

WHAT I PERSONALLY DO

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 counteract 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 been 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 nuanced, 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 extraordinary is this: he can take an ask, get underneath it to the real context, change thought 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.

CASE STUDY — The Support Automation Project

The goal sounded straightforward: automate as much of the work involved in our support contracts as possible. From email with an issue to resolution — make that process better and faster. That's broad. So we didn't start by designing a system. We started by poking at the problem.

First pass: we thought managing emails would be the key leverage point. 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 hadn't really solved anything. We'd 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 hadn't 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 and put it 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 that 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 things and give the team better diagnostic information to work from, we'd eliminate ninety percent of the effort without touching the part that required human judgment.

So that is what we built. End to end on a demo first — n8n for the workflow because it is drag and drop and 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 automation system that handles ninety percent of complex support work.

The path from that point 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.

What this looks like when it goes wrong

I'll be honest. I fail at this regularly. Not in the dramatic ways — I don't 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 didn't stop to ask the most basic question: do you actually understand what this produces and what you'd do with it? That question takes thirty seconds. Not asking it cost weeks. Thirty seconds.

The other version of this is subtler. Sometimes I'll be in a conversation asking the same question ten different ways. At some point I've learned to just say it: I've either not had enough coffee or I'm missing something, but I genuinely don't know what you're describing. Can we start from scratch?

WHAT I'M STILL FIGURING OUT

I should be honest about something before you take the context and communication chapter at face value. I know this framework well. I have seen it work hundreds of times. I also violate it regularly, for reasons that are entirely my own fault.

My biggest challenge with context and communication is my own ego and impatience. I find myself slipping 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 at the end of the day after 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 through it 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 just 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 five percent 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, fail, 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. The five percent was

what they could afford. It 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 also get home to their kids, see their friends, keep their job. Sometimes remembering that is the only context work that actually matters.

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 requirement or 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?

Keep going

You don't have to run all six levers alone. Some people want to build the engine themselves with people who've done it before in the room. Others would rather hand the whole thing over and get back a system that works. Pick the door that fits where you are right now.

Build it with us → {{COMMUNITY_URL}}

Have us build it for you → {{ONWARDS_LABS_CALL_URL}}

> **Curiosity over credentials. Always.**

04

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 determines whether any of them actually work. Because the reason effective work structure is so difficult in technical teams is not that the frameworks are complicated. It is that the role of the leader inside those frameworks is genuinely hard to get right — and when it goes wrong, it goes wrong in ways the leader almost never sees.

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

When you hire a technical team you are, by definition, hiring people who know more than you about their specific domain. That is the point. 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 and 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 happening in the business. The second is application. You are the one who takes technical information and translates it into business decisions. When a developer tells you that reducing the load time by fifty percent would require 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 CTO. 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.

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.

Effective technical teams run 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 the leader realising it is happening. Every time you weigh in on a technical decision that should sit with the team, you are sending a signal that their judgment is subject to your override. Over time that signal accumulates into a team that stops exercising judgment independently — because they have learned, through experience, that it will be second-guessed anyway.

The honest difficulty

Getting the balance right between involvement and delegation is genuinely hard in technical teams, and it is harder than it is in other domains. The reason is that the work is less visible. In a sales team you can see the pipeline. In an operations team you can see the throughput. In a technical team, the most important work — 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 leaders is to step in — to review, to check, to ask for updates — because the alternative feels like flying blind. This is something I bring up 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 requires active effort and deliberate attention. Am I involved in this because my involvement is genuinely necessary, or because it makes me feel less exposed?

THE HOW — 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. 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 via Loom 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 produces a transcript that can go straight into a GPT for further brainstorming. Slack for coordination. WhatsApp or 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 domain leads generate 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 the person has 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.

WHAT I PERSONALLY DO

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 people 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.

CASE STUDY — 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.

What this looks like when it goes wrong

I don't 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's been brewing for weeks. Not double-checking context on a handoff. Getting looped out of a client conversation because someone handled it directly and didn't think to flag it — not out of bad intent, just because the escalation path wasn't clear.

That last one is the most common. I don't want to be the bottleneck. My team doesn't 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 don't get far enough wrong to feel that way. Short, regular, low-stakes. Not glamorous. Just the thing that works.

WHAT I'M STILL FIGURING OUT

The chapter on work structure talks about staying out of the way. My honest problem is that I lean too far in that direction. I get hesitant to challenge. I hold back when I should push. And the situations where this costs me most are the ones involving extremely competent, experienced people — because the doubt that creeps in when you are about to challenge someone who is clearly skilled is genuinely hard to override.

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 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.

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?

Keep going

You don't have to run all six levers alone. Some people want to build the engine themselves with people who've done it before in the room. Others would rather hand the whole thing over and get back a system that works. Pick the door that fits where you are right now.

Build it with us → {{COMMUNITY_URL}}

Have us build it for you → {{ONWARDS_LABS_CALL_URL}}

> **Access. Not just rate.**

05

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 hiring approach 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. I mention that not to undermine what follows but because honesty about the failure rate of any hiring approach is more useful than presenting it as a solved problem. It is not.

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.

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.

Minimise the downside on both sides

One thing I am deliberate about, particularly for offshore hires: I do my best to avoid requiring major life changes before we have established that the fit is real. Asking someone to resign from a stable job to join something new, and then discovering after two weeks that it is not the right match, is a much bigger ask than structuring the initial engagement so that they can work with us on the side while keeping their existing role. If it does not work out, nobody is worse off.

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 alignment 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.

On specialists — and why you probably don't 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.

THE HOW

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 don't 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 sequentially. I ask my offshore leads if they know anyone. I post a LinkedIn ad. I test a specific capability on Fiverr with a small paid engagement. The LinkedIn 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.

I then offer a short contract structured so they do not need to change their situation. They keep their existing job or commitments. 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 extra money and learned something. My maximum exposure is two weeks of part-time pay and some time.

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.

CASE STUDY — 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 that sits 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 didn't understand the work. Wasn't 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.

What this 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 didn't work out? There was usually a moment in the first conversation where something didn't 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've changed is the structure. The two-week trial means backing someone costs me two weeks if it doesn't work, not six months. What I haven't 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'M 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.

CONTRARIAN TAKE

The credential is not the capability

I have more formal education than most people in my industry. BSc in psychology. Master's in marketing. 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.

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?

Keep going

You don't have to run all six levers alone. Some people want to build the engine themselves with people who've done it before in the room. Others would rather hand the whole thing over and get back a system that works. Pick the door that fits where you are right now.

Build it with us → {{COMMUNITY_URL}}

Have us build it for you → {{ONWARDS_LABS_CALL_URL}}

> **Eliminate tasks. Don't just improve them.**

06

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 arbitrage 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. I love working with offshore talent. Some of the best people I have ever worked with are offshore. I have also had some genuinely terrible experiences. Both things are true and both are worth understanding before you decide how much weight to put on this lever in your own situation.

Offshore is not always cheaper by rate

What you are actually 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 locally, and a flexibility in engagement structure that is nearly 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 genuinely 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 model is the senior specialist for whom this is a second engagement — someone already operating at a high level who runs hard for you in a contained, well-defined scope because the work is interesting and the arrangement works for their life.

Understand the legal and commercial reality before anything else

The first thing to understand about building an offshore team is that the legal and payroll infrastructure is the hard part — and most people who write about offshore hiring skip it entirely because it is unglamorous and complicated and varies significantly by country.

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

The liability piece is particularly important and rarely discussed honestly. In most offshore arrangements, the legal owner of the employment contract is your local lead. If you stop paying, they typically have limited recourse against you but significant exposure to the people they have contracted on your behalf. Three to six months of someone's salary is real money. Understanding this explains why good leads are hard to find and why the ones who exist are cautious about who they work with.

The lead is everything

If there is one thing I would want you to take from this chapter it is this. Your offshore lead is the single most important factor in

whether the model works. Not the rate. Not the platform you use to find people. Not the contract structure. The lead.

Finding the right lead has taken me years and multiple painful experiences to get right. What I am looking for is someone who acts as a genuine partner in that location — who takes real pride in running the team well, who is accountable for outcomes rather than just for activity, and who has enough standing in their local market to be genuinely useful when things go wrong. Because things will go wrong.

The model I have landed on: 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 just financially, but in terms of 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 are first in line when something goes wrong.

I am also transparent about rates. On fixed-cost client work the general structure is around forty percent going to the team, covering their rate, taxes, and any overhead the lead carries. On some projects team members have earned significantly more — one hundred and fifty dollars an hour in cases where the work warranted it. Transparent ownership of the economics, shared wins when they happen.

The real risks — and they are real

The bad experiences I have had and heard about from others tend to cluster around a few specific failure modes. 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 controls the relationship. Agencies that squeeze margin at every point. And the one that I find genuinely troubling — the use of unpaid or severely underpaid interns to do paid client work. The direct hiring model through a trusted lead is the most reliable protection against all of these.

THE HOW — Current Setup

My current setup has three major locations. The Philippines — direct hire with very low tax overhead, seriously good people and lots of specialists to be found. Some incredibly bad practices exist in offshoring here, which means there is a real opportunity to recruit exceptional people by simply treating them properly. My technical team is split between Serbia and Bangladesh. Serbia has an amazing talent pool but very high tax rates and strict regulation that requires your local lead to take on significant risk. Bangladesh has dedicated tax incentives for offshore technical development work and is more internationally friendly.

WHAT I PERSONALLY DO

I look after my people and make sure they know it. Not in a performative way — in a practical way. 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 that clearly rather than letting people assume continuity that is not guaranteed.

One practical offer: if you want introductions to leads in any of the locations I operate in, or want to understand the tax and legal structures in more detail, reach out. If it means my team

gets more good work from good people, that is a net positive for everyone.

CASE STUDY — The Bangladesh SaaS Build

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 SaaS 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 approximately five to ten percent of what an equivalent onshore team would have charged for the same scope. 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 who were genuinely running hard for a contained number of hours because the engagement was structured to make that possible. The end result was a production-ready enterprise product built by a team I would put against any onshore equivalent for this category of work.

The real question is not offshore or local. It is where is the right person.

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 the starting point.

We had a client recently who just 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 matters.

The framing most people bring into this conversation is wrong from the start. Offshore is not the budget option for people who cannot afford to do it properly. In most cases, taking budget out of the equation entirely, you can find the right people faster, get the right model set up, and deliver faster using global markets than local ones. The reason is simple: 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 to find one specific person and the signal-to-noise ratio is harder to manage at distance.

The decision framework is not complicated. 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 reframe nobody talks about

Offshore carries a stigma that is almost entirely the product of people doing it badly. The typical understanding 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 to hiring someone from a lower-cost part of your own country — it is just the same thing at scale.

People use cheap offshore work as proof that offshore does not work and then complain about the quality. If you paid a Fiverr 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: 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 genuinely never found a good answer to that question. The work is the same. The output is what matters.

What actually is different — work culture

One thing that is genuinely real and worth understanding before you start: work culture varies significantly and it affects 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 is less confronting than being seen to step out of place. I have also seen a version of this that manifests 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 generally — 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 your environment is different. You have to demonstrate it consistently over time before the behaviour changes.

What I have found is that this pattern does not map cleanly to geography in the way people assume. I have seen it more frequently in certain US contexts than in Bangladesh. Serbia tends to run on stricter hierarchies that take time to shift — but even then it applies maybe half the time and less so with younger people. The younger generation of technical workers has grown up with a more global culture around work. Their reference points are international. Their values around collaboration and communication are closer to what you are trying to build than many people expect.

The practical implication: do not assume the cultural work is done because someone said yes in the interview. Build an environment where disagreement is explicitly welcomed, where mistakes are discussed openly, and where the measure of a good team member is output and honesty rather than deference. Then wait. The people worth keeping will find their footing in it.

What this looks like when it goes wrong

I've played the offshore model conservatively and haven't had a catastrophic failure. The worst outcome is just nothing — no meaningful work. Containable if the structure is right.

What I struggle with is the handoff problem. I find it hard to hand off key client engagements to other people. I tell myself it's about quality. Mostly it's about control.

The honest truth about offshore work is that it reduces to the same problem as all hiring: finding good people and treating them well. The geography changes the logistics. It doesn't change the fundamentals.

WHAT I'M STILL FIGURING OUT

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

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

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

I have done my best to get it right. 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 distinction between running a distributed team and managing offshore resources. They sound like the same thing 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 team culture and operating rhythm across time zones, languages, and different working contexts? Both are solvable but they require different thinking and I have not fully separated them in practice as cleanly as I would like.

CONTRARIAN TAKE

Most agencies are running a business model you do not understand

The standard advice when you need software built, a strategy delivered, or a technical problem solved 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, maximise revenue from that client, reduce the cost of delivery as far as possible, and move on. A long-term view — do exceptional work, become the trusted partner rather than the vendor — is genuinely 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 gets expanded in ways that feel reasonable at the time but were always the plan. The team presented in the pitch is not the team doing the work — the seniors sell, the juniors deliver, and the gap in capability is absorbed quietly. The deliverable is built to look complete rather than to work completely. The quote is not built from first principles of what the work actually costs. It is back-calculated from what they think you will pay.

The reason this 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 quite sure whether the problem is the vendor or whether this is just how hard the

work is. That uncertainty is not accidental. It is the operating environment the model depends on. Someone comes in with an idea they cannot convey in technical terms. The agency quotes a number based not on what it will take to build but on what the client appears willing to spend. The gap between those two numbers is margin.

I have seen this at every price point. The fifty thousand dollar agency and the five million dollar consulting firm are running versions of the same model. The sophistication of the presentation scales with the fee. The underlying dynamic does not.

The agencies and firms that are genuinely exceptional exist. I have worked with a small number of them. The difference is not capability — it is whether the incentive structure of the engagement aligns 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 actually matters.

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 you were to build one offshore relationship in the next six months, what specific capability gap would you be filling — and what would the right person need to look like?
- Do you have a clear enough brief that someone working asynchronously in another timezone could execute it without a daily check-in? If not, what is missing?

Keep going

You don't have to run all six levers alone. Some people want to build the engine themselves with people who've done it before in the room. Others would rather hand the whole thing over and get back a system that works. Pick the door that fits where you are right now.

Build it with us → {{COMMUNITY_URL}}

Have us build it for you → {{ONWARDS_LABS_CALL_URL}}

> **Your team is what makes you good.**

07

Tools & Automation

> **The best tool decision I ever made was to stop using a tool entirely. The second best was a five-minute trial.**

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 quickly and has not let me down yet.

AI is estimation. That is it. 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 judgment about context that the model does not have — AI is not the answer yet, and pretending otherwise will cost you.

Where it actually delivers

The two categories where I consistently get real value from AI tools are removal and challenge.

Removal is the more valuable of the two. The best AI tools I have adopted are the ones that have completely eliminated something I used to have to think about. Automated meeting note recording 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 follow up trying to remember what was agreed. That entire category of cognitive overhead is gone. Not reduced — gone. And gone is a fundamentally different outcome to reduced.

The second category is challenge — brainstorming, assumption testing, general research, first-pass thinking on problems I have not fully formed yet. This is where volume and imperfection is 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 extraordinarily good at this.

Code reviews sit in the same category. A quick function, a first draft, a structural check — perfect use case. Complex architectural decisions, nuanced 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 super confident first year university student who has just finished their first semester and genuinely believes they know everything there is to know about software development. Impressive for the level. Dangerous if you forget the level.

Alignment over optimisation

The single most useful principle for tooling in any team environment is alignment. Not with best practice, not with whatever the most sophisticated option is — with what already exists and what people already use.

Every tool change has a cost that rarely appears in the evaluation. The cognitive overhead of learning something new. The period of reduced 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 factored into the business case for the switch.

Using a slightly worse tool that requires no change is a better outcome than using an optimised 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 PM tools.

THE HOW

For your personal stack: keep it consistent and avoid swapping too much. Switching costs are real even at the individual level. 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 productivity gain.

That said — every so often something genuinely changes the game. When that happens the switching cost is worth paying immediately and completely. The challenge is developing the judgment to distinguish between the two categories quickly, because the people selling you both will use identical language.

WHAT I PERSONALLY DO

I test constantly and I accept that I will sometimes pay for things I do not end up using. That is the cost of staying across what is actually available. Most of it is noise. Occasionally something lands that is genuinely transformative and I would rather find it early than late.

Three recent tools that genuinely changed how I work: Heyreach for LinkedIn automation — set it up, forget about it,

never think about it again. Claude skills for PowerPoint generation — my most hated task is now ninety-five percent faster. Notion meeting transcription with actions automatically collected and added to my task list — the cognitive overhead of capturing what was said and what was agreed is completely gone.

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 to them.

CASE STUDY — Scaling The Testing Culture

The honest admission first. I miss things constantly. Tools that are game changers for someone on my team are invisible to me because I am not doing that work every day. A Claude-powered pull request review tool that transforms the code review process is genuinely extraordinary — for a developer who does code reviews. I was not doing code reviews so I missed it entirely. A new design tool that changes how fast someone can produce visual work will never surface through my testing because I am terrible at design and do not spend time in that space.

This is the real limitation of a leader-driven approach to tooling. Your testing is bounded by your own work. The multiplier is enabling the same testing instinct across the whole team. If your developers are scanning for tools that make development faster, your designers are tracking what is changing in design tooling, your analysts are across what is emerging in data — you have distributed the search across every domain of work.

The structure that makes this work is removing the friction from acting on what people find. The model I have seen work

well — and have started applying myself — is a dedicated discretionary budget that people can spend without asking. A set amount on a card, no questions asked, if they think there is value in testing something. What it communicates is that their judgment is trusted and their curiosity is sanctioned. The result is a team that is collectively staying ahead of what is possible rather than waiting for the leader to notice it first.

How I actually evaluate tools

I want to be honest about this because most advice on tool evaluation is aspirational nonsense. Pilot programs, scoring matrices, formal review periods. Nobody does that. 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 quickly — sometimes five minutes, sometimes a day — I cancel it and move on. I have learned the hard way that 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 completely forgot I signed up for. Learn from that.

The five-minute test sounds shallow. It is actually quite 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 of those are real signals. A tool that requires three hours of setup before it does anything useful is a tool that requires three hours of setup every time someone new joins your team. That cost compounds.

Sometimes there is a longer overlap — running two tools simultaneously for a period before switching. That is not by design. It happens when something is important enough that I

want to be really sure before I cut over. I do not plan for it. It emerges when the stakes feel high enough to warrant it.

For the team I have no formal process either. The only bar is that it gives some kind of value. In practice they are pickier than I am and I have never had to push back on a tool choice they made. Technical people evaluate tools constantly — it is part of how they stay current. Trusting that instinct is usually the right call.

The other reason I test so many tools

There is an honest admission here. Testing tools is partly a hobby. I build a lot of technology and the space moves fast enough that half of what I think I know is wrong by the time I act on it. Even when a tool does not fit what I need right now, the process of testing it teaches me something. That knowledge clicks into new ideas in ways I did not plan for. It also means I have strong opinions about a lot of software nobody asked me about.

I do not think this is inefficient. I think it is one of the ways I stay close to what is actually possible rather than what was possible eighteen months ago. The danger is the line between genuine exploration and procrastination — the elaborate system-building that feels productive but is really just avoidance. I wrote about this earlier and I will not repeat myself. Just know that I sit on both sides of that line depending on the week.

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 then six weeks later you realise the tool does not actually do the thing you needed it to do — or it 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 cost 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.

What this looks like when it goes wrong

I should tell you that I am currently failing at exactly what this chapter describes.

I'm trying to scale my content. The bottleneck is not a tool problem. The solution is simple: pick up the camera and film something.

Instead I have mapped workflows, evaluated scheduling platforms, and started and restarted the system three times. I have filmed almost nothing.

The honest reason is that optimising the process is comfortable. Filming is uncomfortable. The elaborate system is not a productivity tool. It is a very convincing way to avoid the hard part. I know this. I am writing it in a book. And I still do it.

WHAT I'M STILL FIGURING OUT

The line between genuine exploration and productive procrastination is something I have not solved.

I test a lot of tools. Some of that is legitimate — staying current, understanding what is possible, finding things that create real leverage. Some of it is the same avoidance I described earlier in a different costume. The elaborate system that never quite gets finished. The tool that would solve everything if only I could get it set up properly.

What I have not built is a reliable way to tell the difference in the moment. Looking back I can usually see it. In the moment it genuinely feels like the same thing — curiosity, exploration, building toward something useful. The tell is usually whether anything ships at the end of it. But by the time that is clear I have already lost the time.

The AI layer on top of this makes it harder. The pace of change is fast enough that something I evaluated three months ago may genuinely be worth re-evaluating now. That is a real reason to keep testing. It is also a very convincing excuse to keep testing things that do not need testing. I have not found a clean rule that separates the two.

CONTRARIAN TAKE

Everyone is chasing the wrong five percent of AI

AI is a blanket term for a collection of tools, some of which have existed for decades and some of which genuinely are 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 actually is. If you want to know how tall someone is, you collect related data — weight, shoe size, arm span — plot it all, draw a line through it, and make a guess. Machine learning is about getting fancier with how the line gets drawn — clustering groups, identifying patterns across more dimensions, updating as new data comes in. But it is still, fundamentally, an educated guess. The current generation of large language models does this at a scale and sophistication that produces outputs which are genuinely extraordinary. It does not change what they are.

The reason this matters is confidence. When I run a traditional machine learning model in a high-stakes environment, 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 an LLM for a detailed legal document once. It produced a detailed legal document. It looked exactly like a legal document. I had zero ability to evaluate whether it was actually 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 distinguish from the good ones.

The instinct most people have here is almost universally wrong. The pull is to point your AI investment at the most critical and most sensitive use case — because that is what feels important and that is what gets funding approved. These are also the use cases that require the highest certainty and the most careful risk management. Which means 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 LLM platform. By the time it was finished the underlying technology had moved on and the platform did not work properly with the current models.

In my own startup I spent months building a custom back-and-forth reasoning loop for an LLM-powered feature. By the time we had it working the model provider had released a native reasoning capability that did the same thing a hundred times better out of the box. The lesson: build for change, not for correctness. The goal is not to build the right thing. It is to build something you can swap out quickly 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 judgment, the relationship, the decision that requires real accountability — is surrounded by a vast amount of work that is necessary but not valuable. AI is extraordinarily 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 significantly outperform any single critical AI project.

The reason this does not happen is governance. Not security — governance. The decision requires 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 rather than impact. The people genuinely pulling ahead with AI are not doing it because they found a better use case. They are doing it because someone 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?
- What is a tool you are currently using out of habit rather than because it is genuinely 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?

Keep going

You don't have to run all six levers alone. Some people want to build the engine themselves with people who've done it before in the room. Others would rather hand the whole thing over and get back a system that works. Pick the door that fits where you are right now.

Build it with us → {{COMMUNITY_URL}}

Have us build it for you → {{ONWARDS_LABS_CALL_URL}}

> **Six places the gap opens. Six fixes.**

08

Retention & Culture

> **Your job is to get the most out of your team, not to make yourself feel good.**

I said earlier that this lever could be argued as a result of the other levers rather than a lever in its own right. That is partially true. If you get context, structure, the right people, and a working model that makes sense — a lot of what falls under retention and culture emerges naturally. But not all of it. And the part that does not emerge naturally is the part that requires the most discipline, because it is the part that costs you something personally every time you do it right.

Your people are what make you good. Full stop.

There is a specific failure of perception that 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 the outcomes the team produces. They are much less aware of what the team is doing in the background to make those outcomes possible. And over time that asymmetry of awareness creates a distorted picture — one where the leader's contribution looms large and the team's contribution 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 to 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 determine 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 it is, 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 is going to 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.

Accountability is what most people lack

Before you assess whether someone on your team 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, when you trace them 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 narrative 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 HOW — 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: do not respond if I message you and you are not working. Do not work outside of 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 feel comfortable with everything else. The accountability runs both ways or it does not work at all.

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 that 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, here is what I am changing. That last part is what most leaders skip. The acknowledgment without the specific personal accountability is just an apology. The accountability is what makes it mean something.

CASE STUDY — The Rude Client

A client came to me needing help with a website. Small engagement, more of a favour than a commercial relationship. I connected them with my designer — genuinely one of the best I have ever worked with, someone whose English is frankly better than mine. The client's behaviour toward him was contemptuous from the start. The specific words were: your designer is useless, he doesn't speak English. When I spoke to my designer he just shook his head. He had been absorbing it without saying anything to me.

I called them directly. Stopped all work immediately. Handed over everything that had been completed to that point. There were apologies — the kind that come when someone realises the relationship is over — but it did not change the decision. The effect on the team was immediate and visible. They knew the rules we had set at the beginning were real. That knowledge changes the working relationship in a way that is hard to overstate.

One more thing worth saying directly. You may be reading this and nodding along, thinking yes, this is how I already operate with my local team. But this is the part that falls off the radar with contractors and largely does not exist at all with offshore. The check-ins do not happen. The personal investment does not happen. The protection from bad client 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 say something when something is wrong — you have to create the same conditions. You cannot build a high-performing team culture that applies to some of the team and not others.

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 much simpler. I reach out in whatever way that person prefers — some people want a call, some want a Slack message, some want to talk face to face. The medium matters less than the directness. And I am direct.

Something like this: hey, I'm struggling to figure out how to say this but honest version — your communication style with your juniors is hurting more than helping. I get it. But it's burning them. You're doing great work and I get that they need to step up, but that approach isn't 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 immediately 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.

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.

The money follows-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 that 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 that this is working — the one I pay attention to — 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 figure 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 just reality. Your job when that 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's the one where nothing went wrong.

I had a young guy on the team. Capable, motivated, did good work. Just didn't like software development. Wanted to be outside. You can't fix that. What you can do is make the exit good — be supportive, help them land somewhere better.

I worked at a consultancy 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 figure out why.

How you treat people on the way out is one of the clearest signals your culture sends. Everyone who stays is watching.

WHAT I'M STILL FIGURING OUT

I have not figured out how to scale culture without diluting it.

The things that make a team feel like a good place to work — the directness, the follow-through on money, the space to disagree, the sense that the person running it actually gives a damn — all of those depend on me being close enough to the relationships to hold them. They work because they are personal. The question I keep coming back to is what happens

when the team is large enough that I cannot be personal with everyone.

I have seen the answer that most growing businesses land on: hire managers, delegate culture, build processes. It works in the sense that the business continues to function. It does not work in the sense that the culture you had at twenty people is not the culture you have at a hundred people. Something gets lost and most leaders call it scaling. I am not sure that is the right word for it.

I am not sure I accept it yet. I have not been at the scale where I have had to find out. What I know is that the things I value most about the teams I have built are the things most at risk when the team gets bigger. I do not have a clean answer to how you protect them. I am watching closely and I will let you know.

BEFORE YOU MOVE ON

- Think about the last time a team member 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 on your team — explicit or implied — that you have not followed through on? What is that doing to the relationship?
- Who on your team receives the least personal investment from you right now? What would change if you reversed that for thirty days?

Hard Lessons — What This Actually Cost Me

Most business books are written from the position of having figured it out. The author has made the mistakes, absorbed the lessons, and now presents the framework that emerges from the other side. The mistakes themselves are cleaned up, reframed as learning moments, given narrative arcs that end with insight.

I want to be more honest than that. Not because honesty is a brand position — because the cleaned-up version of these stories is genuinely less useful than the real one. So here is what building seven businesses across fifteen years actually cost me, in the categories that still sting.

The partner who blew the cash

I had a business partner. Things went wrong in the way things go wrong when there is money involved and trust breaks down. I ended up taking a payout to exit. I lost three software products I had built and the consulting business that went with them. The details are not ones I will put in print — there are NDAs and there are people involved who deserve privacy even if I am not sure they deserve much else.

What I will say is this. The most expensive thing I have ever done in business is fail to define what happens when things go wrong before they go wrong. Not what happens when the business succeeds — everyone wants to have that conversation. What happens when it does not. Who owns what. What the exit looks like. What the decision-making process is when the two of

you fundamentally disagree. Those conversations are uncomfortable to have when things are going well and they feel unnecessary. They are the only conversations that matter when things are not.

I lost years of work. I started again. The framework I use now for any partnership or significant relationship has more legal structure in it than most people think is necessary. Most people think it is excessive. Those people have not lost years of work yet.

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, not the client's. For six months I had a person on payroll who was not working, who knew the legal and contractual situation gave them leverage, and who used that leverage. It cost me six months of salary, a significant amount of legal time, and the kind of ongoing stress that makes everything else harder.

The lesson I took from it is structural rather than personal. Salaried employment creates obligations that do not scale cleanly with performance. The probation period that is supposed to protect you rarely does in practice — by the time you have enough evidence to act on, you are usually past the point where acting is clean. The two-week trial model I describe in the hiring chapter exists directly because of situations like this one. It is not a preference. It is a scar.

The sob story hires

I have a genuine soft spot for people who just need a shot. I have hired based on that instinct more times than I should have. The pattern is consistent enough that I can describe it precisely. Someone comes in with a compelling story — difficult circumstances, real talent that just needs the right environment, a history that explains why the CV does not reflect the capability. I give them the shot. The performance does not materialise. The story expands to explain why. I invest more. The performance still does not materialise. By the time I accept what is happening I have spent significantly more than the cost of the initial mistake.

I do not think giving people shots is wrong. I still do it. What I have learned is the difference between giving someone a shot in a contained structure — a small engagement, a clear deliverable, a defined period — and giving someone a shot in an open-ended one. The first costs you two weeks if it does not work. The second costs you considerably more. I know which one I kept choosing. That is also in my graveyard.

The legal threats

I have had legal action threatened over an NDA that was vague enough that neither party actually knew what it covered. I have had an offshore staff member threaten legal action and demand six months pay after being let go for genuine performance issues. In both cases the legal exposure was less than it felt like in the moment — but the cost in time, energy, and distraction was real regardless of the eventual outcome.

The pattern in both cases was the same. Vague agreements made when the relationship was good that became weapons

when the relationship went bad. Specificity in agreements is not bureaucracy. It is the thing that means a bad outcome costs you a defined amount rather than an undefined one.

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, the timing was wrong, the original assumption did not survive contact with reality. Those I am at peace with. What I am less at peace with is the projects that ran on past the point where I knew they were not working, because stopping felt like failure and continuing felt like persistence. The distinction between those two things is one I still find genuinely hard to make in real time.

I have spent significant money on app projects that are still not generating revenue. I have watched the same pattern play out in both directions — as the founder spending money and as the advisor watching a founder spend money. The sunk cost trap is not a cognitive bias that other people fall into. It is something I fall into, have fallen into, and will probably fall into again. Knowing it exists does not protect you from it. What helps is having someone external who has no emotional investment in the project and is willing to say the thing that is obvious from the outside.

The vendors

Nine in ten vendors I have used have been somewhere between disappointing and genuinely terrible. I have had contractors hold deliverables and demand fifty percent more cash than was agreed. I have had agencies produce work that required complete remediation before it could be used. I have spent more money than I should have testing tools that did not work, on

vendor promises that did not survive the first month of production use.

The honest version of why this keeps happening is not that I have bad judgment about vendors. It is that the vendor selection process is structurally broken in ways that favour the vendor. You are evaluating a sales presentation, not the actual work. The work only becomes visible after you have signed and paid. By then the leverage has shifted completely. What I do now — and what I would do from the beginning if I were starting again — is pay a small amount for a small thing before committing to a large thing, and treat the result of that test as the only signal that actually matters.

The work I should not have taken

I have taken consulting projects I hated because I needed the revenue. I have been pushed into selling work I was not comfortable selling because a client wanted it and the commercial pressure was real. I have stayed in engagements past the point where they were good for me or my team because ending them felt harder than continuing.

The cost of taking work that is wrong for you is not 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 consumes that could have gone toward something better. It is the signal it sends to your team about the standards you are willing to hold. And it is the slow erosion of the positioning you are trying to build — because every time you say yes to the wrong thing you make it slightly harder to say yes to the right thing.

I am more disciplined about this now than I have ever been. I am not perfectly disciplined. But I have enough scar tissue from

the alternative to know that the revenue from the wrong engagement is almost never worth what it costs.

Keep going

You don't have to run all six levers alone. Some people want to build the engine themselves with people who've done it before in the room. Others would rather hand the whole thing over and get back a system that works. Pick the door that fits where you are right now.

Build it with us → {{COMMUNITY_URL}}

Have us build it for you → {{ONWARDS_LABS_CALL_URL}}

Closing Summary

> **Most people will use one or two of these levers and see real results. A few will use all six. You will know which one you are by whether you close this book or open a notebook.**

You have just read six levers. Most people will use one or two and see meaningful results. Some will work through all of them and build something genuinely different. A few will read this, nod along, and change nothing. If that is you, I genuinely hope the next book works out better.

The gap between those outcomes is not intelligence or resources or luck. It is whether you do the work of actually closing the gap — the uncomfortable conversations, the honest context mapping, the discipline to lead the system instead of firefighting inside it.

The technical side of your world is not going to slow down. The tools will keep changing. The talent market will keep shifting. The distance between the business and the technical will keep widening if you let it — and it will keep costing you if it does.

What does not change: the six places the gap opens. The way trust gets built across that divide. The difference between a team that owns the outcome and a team that executes instructions. Those things were true ten years ago and they will be true in ten more.

If one thing in this book made you think differently about a problem you are currently sitting in — act on that first. Not the whole framework. The one thing. This week. The rest follows from starting.

And if something in here is wrong, or you have a better version of it from your own experience — tell me. I mean that. The best version of this conversation is not one direction.

Real Questions. Straight Answers.

> **The questions people actually ask — rephrased exactly how they ask them.**

This section is different from everything that came before it. No framework, no principles, no methodology. Just the questions I get asked most often — by founders at different stages, by people who are stuck — and the most useful answer I can give to each one.

The questions are rephrased as people actually ask them, not as a textbook would present them. If something sounds blunt it is because the honest answer usually is.

(If you are inside a larger organisation rather than building a startup, there is a bonus set of questions for you at the very back of the book — "For Readers Inside Larger Organisations.")

The Planning Founder

Someone with an idea, trying to pick a direction before committing significant time or money.

What would you do differently if you were starting this from scratch?

I would define done before starting. Not the features, the outcome. What does the business look like when this is working? What decisions will be made differently? Who will use it and

how? I would get a technical person who has no stake in the outcome to review any significant vendor proposal before I signed it. And I would build nothing until I had spent real time with the people who would actually use it — not their manager, them — and understood what they needed in their own words.

What is the biggest mistake you see founders make when building their first technical team?

Hiring for technical skill before they understand what they need built. The first technical hire needs to be someone who can help you figure out what to build, not just someone who can build it. That requires communication, curiosity about the business problem, and the judgment to push back when the brief is wrong. Pure technical skill with none of those things will produce technically impressive work that misses the point. Hire for the conversation as much as the capability.

Should I hire a CTO or just find a good developer to start?

Find a good developer first. A CTO without a working product is expensive overhead. You need someone who can build before you need someone who can lead. The exception is if you are raising significant funding and investors are asking — in which case the title matters for optics. But for building, start with a developer who can ship and is honest about what they do not know. And remember: until you can afford or attract that leadership hire, the technical leadership seat is yours. That is what this entire book is about.

What does a senior developer actually cost — locally, offshore, and everything in between?

Senior developer locally in Australia: $120–180K salary, or $120–180 per hour as a contractor. The good ones are at the top of that range and worth it. Offshore senior developer from a reputable agency in Serbia, Philippines, or Eastern Europe: $40–80 per hour depending on the market and the person. Quality varies enormously — the best offshore developers are as good as anyone local, but the worst are not worth any price. The hidden cost people forget: management time, communication overhead, and the cost of getting it wrong. Cheap offshore with poor communication can cost more than local with good communication.

Agency versus freelancer versus hiring someone — what are the real tradeoffs?

Agency: faster to start, more accountability, higher cost, variable quality, and you are one client among many. Good for defined scopes. Bad for anything requiring deep context about your business. Freelancer: more flexible, lower cost, higher personal accountability, but single point of failure. If they get sick, go quiet, or get a better offer, you are stuck. Good for contained work. Bad for critical ongoing development. Hire: highest cost to start, highest loyalty, builds internal capability over time, and you own the relationship. Good for core product. Bad if you are not sure what you need yet. Most early-stage founders start with a freelancer or agency, then hire once they understand what they actually need.

I have been quoted wildly different prices for the same thing. Why is there such a big range?

Because software scoping is genuinely hard and most developers are quoting different things. One developer quotes the minimum viable version. Another quotes everything they think you might want based on the conversation. A third adds contingency for all the things that usually go wrong. A fourth is doing fixed-price and building the risk premium into the number. Ask each of them to describe in plain language what their quote includes and excludes. The differences in those descriptions will tell you more than the numbers.

Should I build this myself or buy something off the shelf?

Default to buy. Build only when you have exhausted the off-the-shelf options and can clearly articulate what they cannot do that you need. Custom software is expensive to build, expensive to maintain, and takes longer than quoted. Off-the-shelf software is imperfect, requires compromise, and usually does ninety percent of what you need. Ninety percent off the shelf today beats custom in twelve months in most cases. The exception: if the thing you are building is your actual competitive advantage — the thing that makes your product different — then build it. For everything else, buy.

What tech stack should I use — everyone gives me a different answer?

The honest answer is that stack matters less than the team. A great developer in any mainstream language will outperform a mediocre developer in the theoretically optimal one. That said: if you are hiring in Australia, JavaScript and Python have the largest talent pools, which means more options and more

competition keeping quality up. If you are building a mobile app, React Native gives you one codebase for both platforms. If you are building data infrastructure, Python is the default. Ignore anyone who tells you there is one right answer. There are good and bad choices, not one correct one.

How do I find good offshore developers without getting burned?

Start small. Pay for a small piece of work with a clear deliverable before committing to anything significant. The quality of that output is the only reliable signal you have. Check references from people who have used them for at least six months, not just a successful project. Ask specifically about communication when things went wrong. Use platforms like Turing, Toptal, or established agencies in established markets rather than individual freelancers from job boards. The vetting has been done. The margin you pay is worth it early on.

I have a co-founder who is technical. How do I know if they are actually good?

Ask them to explain a recent technical decision they made — what the options were, why they chose what they chose, and what the tradeoff was. Good technical people can do this clearly to a non-technical audience. They have done it before. Then ask what they would change about the current codebase and why. If they have no opinion, or if they only have positive opinions, that is a signal. Good developers always see things they would do differently. And ship something together. A small thing, quickly. How they work under pressure and communicate problems is more useful than any interview.

How long should something like this actually take to build?

Simple informational website: one to two weeks. Basic web app with user accounts and a few core features: six to twelve weeks. Complex product with integrations, payments, and multiple user types: four to eight months. Any of these can double if the brief changes, the data is messier than expected, or the team is part-time. Most quotes assume nothing goes wrong. Something always goes wrong.

What broke on your early builds that you wish you had caught earlier?

Not validating assumptions early enough. Building for months before showing it to real users and finding out they wanted something different. Not having monitoring in place from day one. Finding out about a bug when a user reports it rather than before they notice. Underestimating the data problem. The application code is often straightforward. The data it needs to work with is almost always messier and more complex than it looks.

If you were me with this idea and this budget, what would you actually do first?

Talk to ten potential users before writing a line of code. Not to validate the idea — to understand the problem deeply enough that you know exactly what matters and what does not. Then build the smallest possible version that tests the most important assumption. Not the full vision. The one thing that, if people use it and value it, tells you that the rest is worth building. Most early builds are too big. The discipline of building less is harder than it sounds and more important than almost anything else.

The Stuck Founder

Someone who has done significant work, spent real money, and things are just not progressing.

We have been building this for eight months and it still isn't done. Is that normal?

For a genuinely complex product with a small team, yes. For most things, no. Eight months with nothing to show is a problem. Eight months of steady progress with a working product that is still being refined is different. The question is not how long it has taken — it is whether you have something working that real users have touched. If nobody has used any version of this yet, that is the problem, not the timeline.

My developer just went quiet. What do I do now?

Give it forty-eight hours then send one direct message: "I have not heard from you in a few days. Can you let me know where things stand by end of tomorrow?" If you get nothing after that, call. Not email — call. If you still cannot reach them, you have a problem and you need to start thinking about contingency. Do not catastrophise before you have tried to reach them directly. Developers go quiet for many reasons, most of which are not malicious. But also do not wait two weeks hoping it resolves itself.

We keep getting told it's almost done. 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 remaining. Not "a few things" — every item, with an estimate for each. If they cannot produce that list, they do not know where they are. If they can produce it, you can

have an honest conversation about timeline. The pattern of "almost done" extending indefinitely usually means scope was underestimated and nobody wants to have the conversation about what that means for time and cost. You need to have that conversation.

I feel like every time I ask for one thing I get told it needs five other things first.

Because software has dependencies and most people quoting you are being accurate, not difficult. The question is whether those dependencies are real or whether the scope is being expanded. Ask them to separate the two: what do we actually need to do the specific thing I asked for, and what are you recommending we also do and why? If they cannot separate those two things, they are not thinking clearly about the problem. If they can, you can decide what is necessary and what can wait.

We have spent way more than we budgeted and I can't tell if we are close or not.

Get an independent technical person — not your current developer — to assess where you are. Someone who has no stake in the outcome, who can look at what has been built and tell you plainly: what works, what does not, how much is left, and whether the current approach is the right one. This costs money and your current developer will not love it. Do it anyway. You are making decisions without information. That assessment gives you information.

My developer built something that works but nobody can change anything without breaking it.

This is called tightly coupled code and it is a common outcome when something has been built quickly by one person without thinking about maintenance. It is not necessarily a disaster — it can be refactored over time. The question is how much it is costing you right now. If every small change requires days of work and regularly breaks other things, the refactoring cost is lower than the ongoing cost of the current state. Get an independent assessment before deciding to rebuild. Sometimes the problem is smaller than it looks.

I brought in a new developer to look at the code and they said it needs to be rebuilt from scratch. Is that true or are they just trying to get more work?

Sometimes true, sometimes not. The way to tell the difference: ask them to be specific about what the problems are, how long they would take to fix incrementally, and what rebuilding would cost and take. Then ask what they would do if it were their own money. A developer who is generating work will give you a vague answer about fundamental architectural problems. A developer who is being straight with you will give you a specific list and an honest opinion about which approach makes more sense. If they cannot articulate the problems specifically, get a second opinion.

How do I know if my developer is actually working or just 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." A developer who is working will either show you

the thing or tell you in advance why it is not ready and when it will be. A developer who is not working will find reasons why that specific thing is not the right thing to show you right now. The pattern of output is the signal. Not the communication, not the updates — what actually gets built and when.

We have two developers who keep blaming each other when things go wrong. How do I sort that out?

Put them in the same room — physical or virtual — and ask them to walk you through the problem together, out loud, in front of you. Blame disappears quickly when both people have to explain their version of events simultaneously. Either the problem becomes clear, or it becomes clear that one of them is not being straight with you. Do not adjudicate between their separate accounts. That just entrenches the dynamic. Make the problem visible together.

I gave my developer really detailed requirements and they still built the wrong thing. What went wrong?

Because requirements on paper are not the same as shared understanding of the problem. The requirements told them what to build. They did not tell them why, who for, what the person using it actually needs, or what good looks like. So they built what was written down, literally, and it was correct to the spec and wrong to the need. The fix going forward is to review working software early and often — not requirements documents. Show them what you mean, not describe it. And involve the end user in the review, not just yourself.

I need to cut the scope down but I don't know what to cut. How do I decide?

Cut anything that is not required for the core loop to work. The core loop is the one thing a user needs to be able to do for the product to have any value at all. Write down every feature currently in scope. Ask for each one: if we removed this, would the product still do the one thing it needs to do? If yes, cut it. You can add it back later. You cannot get back the time you spent building things nobody needed yet.

My developer says we need to rewrite everything in a different language. Do we?

Rarely. This is a common recommendation that is sometimes right and often self-serving. The question is whether the current language is actually causing a problem or whether it is just not the developer's preferred tool. If the language is causing performance issues that cannot be solved another way, or if it is impossible to hire anyone who knows it, those are real reasons. "It would be cleaner in X" is not. Get a second opinion from someone who has no stake in the rewrite before agreeing to it.

I just want something that works. How do I stop this from being so complicated?

Because software is complicated and most of the people building it have not been taught to protect you from that complexity. The ones who can keep it simple are usually the most experienced — not the youngest, not the ones with the most impressive technical vocabulary. Find someone who has built things that worked and been maintained for years. Ask them what they would cut. Experience makes things simpler. Inexperience makes them more complex.

Bonus — AI and the Six-Lever Engine

> **Why the people already doing this well are pulling further ahead, and what everyone else needs to understand before they spend another dollar.**

I want to be honest about something before this chapter starts. Most of what gets written about AI in a business context is either hype dressed up as insight or fear dressed up as caution. Neither is very useful. What I am going to try to give you instead is the most accurate picture I can of what is actually happening — what works, what does not, why the conventional approach keeps failing, and what the people getting real results are doing differently.

I should also say: this is the fastest-moving area I have ever worked in. Something I write today may be partially 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 much more slowly than the technology does.

Part 1 — What AI Actually Is Right Now

AI is a blanket term for a collection of tools, some of which have existed for decades and some of which genuinely are new. What has changed is not the underlying logic. It is the availability, the

barrier to entry, and the quality of the output you can get without significant technical setup.

The simplest honest description: AI is estimation at scale. If you want to know how tall someone is, you collect related data — weight, shoe size, arm span — plot it, draw a line, and make a guess. Machine learning is about getting more sophisticated with how that line gets drawn. The current generation of large language models does this across language, at a scale and sophistication that produces outputs which are genuinely extraordinary. It does not change what they fundamentally are: very good guesses, made very quickly, across a very wide range of inputs.

The reason this matters practically is confidence calibration. When you run a traditional machine learning model you get a validation process — backtesting, accuracy scores, explicit understanding of where it works and where it does not. The current AI tools do not come with that calibration built in. They produce outputs that look authoritative whether they are right or not. That is both what makes them extraordinarily useful and what makes them genuinely dangerous in high-stakes contexts.

The other thing worth understanding: the gap between what these models could do twelve months ago and what they can do now is enormous. The gap between now and twelve months from now will be equally large. Anyone trying to build for a fixed version of this technology is building on sand. The only sensible architecture — for software, for process, for strategy — is one you can update.

Part 2 — What Is Actually Working

I want to be straight with you: I have not seen much custom AI deployment that genuinely works yet. The out-of-the-box tools

are not great for a lot of use cases either. What I have seen work is when people learn what these tools actually do well — which is different from what they are marketed to do — and then reframe their work around that.

What AI does well: drafting, summarising, reformatting, first-pass reviewing, generating options, handling repetitive language tasks at volume. What it does not do well: anything requiring genuine judgment, institutional context, accountability, or decisions where being wrong has serious consequences.

The use cases that are actually delivering value are almost embarrassingly mundane. Auto-drafting emails. Meeting transcriptions with automatic action logging. Simple internal assistants for room bookings and calendar management. First-pass document drafts that a human reviews and refines. Pulling together a few slides to convey what someone needs — work that used to take days now taking a few hours. Reviewing documents for completeness or inconsistency before they go out.

None of those are impressive. None of them will get funding approved in most large organisations. All of them are actually being used, which is more than can be said for most of the impressive ones.

The pattern across all of it is the same: low-hanging, repetitive, language-based work that has a clear enough brief that AI can do a useful first pass without needing deep context. The value is not the output. The value is the time reclaimed from the task — time that a person can now spend on the five percent of their work that actually requires them.

Part 2b — The Developer Problem Nobody Is Talking About

There is a version of the AI productivity argument that sounds compelling and is mostly wrong. It goes like this: AI makes developers faster, therefore you need fewer developers, therefore your costs go down. I have watched founders and executives build entire hiring strategies around this logic. It does not survive contact with reality.

A developer still costs the same. Their cost of living has not changed. They are not suddenly willing to work one hour a week instead of full time because they have access to better tools. Your cash outlay for a good developer is identical to what it was two years ago. What has changed is the output that is possible from that same developer with the right AI stack. A good developer with the right tools is genuinely fifty times faster than a good developer without them. That is not hype — I have watched it happen.

So the opportunity is not cost reduction. The opportunity is output multiplication. The developer you are already paying for can now build what used to require a team. The question is whether you are set up to capture that.

Most are not. And here is why.

Most development has always started with a vague list of requirements and proceeded through iterative exploration. Try something, see what it produces, adjust, try again. That process works. It is how most software gets built. AI does not fix that process. It accelerates it. Which means if the process is inefficient — and most of the time it is — 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.

The compounding effect — the one that is genuinely extraordinary — only happens when three things are true simultaneously. The right developer, paid well enough that they are not distracted or resentful or already looking elsewhere. The right tools, set up properly and actually integrated into how they work rather than bolted on as an afterthought. And a crystal clear scope — not a vague direction, not an iterative exploration brief, but a specific and well-defined problem with explicit success criteria that both sides have agreed on.

When those three things are true, you are not looking at fifty times faster. You are looking at a hundred to a thousand times faster. I have seen this. It is not theoretical. A single developer with genuine AI fluency, working from a clear brief, can produce in a week what used to take a team a quarter. That is the real opportunity. It has nothing to do with cutting headcount and everything to do with the quality of the setup.

The implication for how you hire changes significantly. You are no longer looking for volume. You are looking for one exceptional person who understands both the technical landscape and how to use AI as a genuine force multiplier — not as a shortcut around thinking, but as a tool that removes the ceiling on what careful thinking can produce. That person exists. They are not cheap. They are worth considerably more than their rate, because the output gap between them and an average developer is now so large that the comparison barely makes sense.

AI is a tool. It requires the same things every other tool requires: the right person using it, the right problem to apply it to, and enough clarity about what success looks like that the tool

is actually solving the right thing. The people who treat it as a strategy rather than a tool — who expect the technology itself to produce the clarity and the direction — are going to spend a great deal of money finding out why that does not work.

Part 3 — The Contradiction at the Heart of Enterprise AI

Here is the central problem nobody is being honest about. The standard enterprise approach to making AI safe is to strip out all the variability — lock down the inputs, constrain the outputs, build governance layers, require sign-off at every stage. This is also the approach that makes AI useless. Variability is not the risk. Variability is the capability. The moment you remove it you have an expensive autocomplete.

Rapid, small, and highly flexible is what works. The thought process around AI deployment needs to change from finding the single targeted measurable use case to enabling much smaller changes much more widely. Not one project that promises to save ten million dollars. A hundred small changes that each save an hour a week per person.

The measurement problem is real and worth naming directly. If you made everyone ten percent more efficient with AI tools today, it would take months for that efficiency to actually surface in your outcomes. People need time to change their patterns. The efficiency needs to compound before it becomes visible. What you will likely see first is not cash flooding in — it is cost growth slowing down. That new hire you were about to make turns out not to be needed. The project that was going to require a vendor engagement gets done internally. The money just quietly stops getting spent in places it used to get spent.

That is not a story that gets executive sponsorship. It is not a story you can build a business case around. It is the honest story of how productivity improvements actually flow through organisations — gradually, diffusely, in ways that are nearly impossible to attribute. Most will not invest in it for exactly that reason. The ones that do will find themselves with a structural cost advantage in three years that their competitors cannot easily close.

Part 4 — How the Six Levers Apply to AI Projects

Every AI project I have seen fail has failed on one of the six levers. Not on the technology. The technology is usually fine. The failure is in the system around it.

Context and communication failures are the most common. The brief was wrong. The business side described what they wanted without being able to convey what they actually needed. The technical side built to the brief. Nobody closed the gap. The output does not get used because it solves the stated problem rather than the real one. An AI model that answers the question you asked is not useful if you asked the wrong question.

Work structure failures are the second most common. Someone builds a proof of concept that works. Nobody builds the production version because there is no clear owner, no clear process for moving from demo to deployed, no clarity on who maintains it when the model provider updates their API and everything breaks. This happens constantly. Impressive demos that live on someone's laptop forever.

People failures on AI projects tend to look like this: you hire data scientists when you need engineers, or engineers when you need people who understand the business problem, or people who understand the business problem when you need someone

who can translate between all three. The rarest person in any AI project is the one who can hold the technical side and the business side simultaneously. That person is worth finding before you start.

Tools failures are particularly expensive in AI because the space moves so fast. Teams that built deep process around specific models or tools eighteen months ago are now locked into something that underperforms what they could get for free from a current general model. The lesson is the same as in the tools chapter: do not build deep process around anything until you know it is staying. In AI, assume nothing is staying.

The lever that gets ignored most often on AI projects is retention — specifically, the retention of the people who built the thing. AI systems require ongoing maintenance, context, and institutional knowledge that is almost impossible to document fully. When the person who built it leaves, the system degrades in ways that are slow, invisible, and expensive. If you are investing seriously in AI capability, invest equally seriously in keeping the people who built it.

Part 4b — Two Case Studies That Bracket the Whole Argument

I want to give you two concrete examples that sit on opposite sides of this. One is a cautionary tale about where AI creates a trap that is hard to see until you are already in it. The other is this book.

Case Study 1 — The 90% Problem

I have had several conversations recently that follow exactly the same pattern. Someone comes to me and says: I have been stuck at ninety percent for months. It was moving fast at the start.

Now every small change takes forever, breaks something else, and I cannot see a path through.

In almost every case, the same thing has happened underneath.

AI tools are genuinely lowering the barrier to entry for technical work. A designer who has never written code can now take a beautiful mockup, run it through the right tools, and produce a working web application that looks exactly like what they designed. That is remarkable. It is also where the trap opens.

What gets built using this approach is a collection of single features that do not fit together. Each one was built in isolation, each one does what it was asked to do, and none of them were built with the others in mind. The underlying architecture is not an architecture — it is a series of individual responses to individual prompts, stitched together until something that looks like a product appears.

The models get lost. They duplicate logic. They do not consider the context that exists outside the specific request they are answering. The codebase becomes impossible to reason about, even for the AI tools themselves, because there is no coherent structure for them to reason about.

The useful analogy is a super smart junior developer who does exactly what you ask. Give a junior a simple task with a clear brief and they will execute it perfectly. Start adding complexity — systems that need to talk to each other, state that needs to be managed across the application, edge cases that require understanding of the whole rather than the part — and a junior gets stuck. That is not a criticism of juniors. It is a description of where experience matters. AI tools are currently somewhere between a capable junior and a mid-level developer

depending on what you are asking them to do. They are moving up that scale fast. But right now, when your project crosses a certain complexity threshold, you need a human who can hold the whole picture in their head.

The practical implication: use AI tools aggressively for simple, well-scoped problems. Be honest about the complexity threshold of your specific project. When you feel the resistance starting — when the changes are getting slower, when fixes are creating new problems, when you cannot quite explain why it is not working — that is the signal that you have crossed the line where AI alone is not enough. Get a senior developer involved before you are fully tangled, not after. The earlier you bring them in, the cheaper the untangling.

This threshold is rising constantly. What used to require a senior developer six months ago can now be done reliably with AI tools. What requires a senior developer today will probably be handleable by AI tools in another six months. The landscape shifts fast enough that any specific advice I give you here will be partially wrong by the time you read it. What will not change is the principle: know where the complexity line is, and be honest about which side of it you are on.

Case Study 2 — This Book

I want to be transparent about how this book was made, because it is a better illustration of what AI is actually good for than anything I could construct as a hypothetical.

I used AI tools heavily throughout this process. What that looked like in practice: I spent a long time iterating on structure — what the book should cover, how to order the ideas, how to frame concepts for different audiences. I used AI to walk me through section by section and capture my thinking. When I had

a direction, I used it to help me develop rough notes into more complete ideas.

What did not work: using AI to expand my notes into finished written text. Every time I tried that, the output was technically competent and completely flat. It oversimplified. It lost whatever personality the original had. It read like a summary of what I wanted to say rather than me actually saying it. The words that ended up in this book are mine. All of them. That was not the plan at the start — I assumed I would be able to use AI more directly for the writing itself. I was wrong about that.

What worked extremely well: editing. Checking for consistency across chapters. Identifying redundancy. Reviewing whether a section would make sense to someone who had not read the earlier chapters. Benchmarking against other books in the category. Generating fifty possible titles so I could see what territory I was in. Flagging where my argument had gaps. All of that was genuinely faster with AI involvement than without it.

The end result is that I wrote all the words but the process was massively faster and cheaper than the traditional route. It does not always feel like AI is doing any heavy lifting — when you are the one still doing the hard thinking, the hard writing, the hard decisions about what to include and what to cut, it can feel like you did all of it. When I step back and look at what the process would have looked like without AI assistance, the honest estimate is that it saved me around eighty percent of the time and cost I would otherwise have spent.

That is the real case for AI as a tool. Not that it does the work for you. Not that it replaces the judgment or the voice or the decision-making. It removes the friction around the work, compresses the loops, handles the tasks that do not require you — and leaves you with more capacity for the parts that do. The

quality of what comes out depends entirely on the quality of what you put in and the clarity of what you are trying to produce.

Strategic and intentional. That is the whole game.

Part 5 — The Honest Summary

AI is not going to transform your startup overnight. It is not going to replace most of the people on your team in the near term. It is not the silver bullet that the vendor presenting to you this quarter is selling.

It is also genuinely extraordinary. The things it can do at the low end — the draft, the summary, the first pass, the repetitive task — it does faster and at lower cost than any human can match. If you give your people access to it, train them on what it is good for, tell them what not to put into it, and let them find their own applications — you will get real value. Not dramatic value. Quiet, compounding, structural value that shows up in your cost base over time.

The people who are genuinely pulling ahead are not doing it because they found the killer AI use case. They are doing it because someone made a broad decision, gave people permission, accepted that some things would go wrong, and moved faster than their governance process wanted them to. That is a leadership decision. It has almost nothing to do with the technology.

The six levers in this book apply to AI projects exactly as they apply to everything else. The gap between the business and the technical is wider on AI projects than on most — because the technology is newer, the vocabulary is less shared, and the hype

makes honest conversation harder. That makes the levers more important, not less.

Close the gap. Do the boring work. Let the technology do what it is good at. The rest follows.

Keep going

You don't have to run all six levers alone. Some people want to build the engine themselves with people who've done it before in the room. Others would rather hand the whole thing over and get back a system that works. Pick the door that fits where you are right now.

Build it with us → {{COMMUNITY_URL}}

Have us build it for you → {{ONWARDS_LABS_CALL_URL}}

About the Author

> **I never set out to be the person who translates between technical and non-technical teams. It just kept happening.**

Ryan Richardson is the founder of Onwards, a capability deployment firm that does this work at the highest end of the market — including with tier-one mining and resources companies like BHP, Rio Tinto, and South32. He is also CTO of a funded startup and runs several other ventures simultaneously. He has spent fifteen years building at the intersection of business and technology — not from one side of it, but from both at once.

The background is genuinely unusual. Psychology. Marketing. An MBA. A year of medicine before leaving. A started PhD. Army reserve. Paramedic training. None of it was strategic. All of it turned out to be relevant.

Psychology taught him to read people and systems. Marketing taught him how communication actually works. Business taught him how organisations make decisions. Medicine taught him to think clearly under pressure. The army taught him that delivery is not optional. Paramedic training taught him to act decisively with incomplete information.

He also has ADHD. His brain runs multiple things simultaneously, finds patterns across unrelated domains, and gets deeply impatient with slow processes. For a long time that felt like a liability. Eventually he realised it was exactly the thing

that made him useful at the gap between business and technology — the place where neither side can fully see what the other is doing, and someone who can move between them is worth a great deal.

He learned technology not because he loved it but because he saw it as the only honest way out of a model he refused to accept — the model where output is measured by hours in a chair rather than value created. If the automation did not work he was still there at midnight. That changes how you think about technology permanently. It stops being about what is possible and starts being about what is necessary.

He built in the shadows of large organisations for years. If he asked permission the answer was always no. If he showed them what was already working they were suddenly very interested. Spend your own time. Take the risk. Prove the result. Not the right way. Just the only way that worked.

Right now he has two young kids, two dogs, three cats, a growing team, a funded startup, and more on his plate than any sensible person would take on. Every business decision he makes is designed to keep him at home with his kids. That is not a lifestyle choice. It is the whole point. Technology is what makes it possible — more leverage, more value created, less time spent proving yourself in a room.

He is not teaching this from a comfortable distance. He is still inside it. Today.

Both Sides of Tech is the community he built for people who live in this gap — most of all the founder who can't get their product built and is learning to lead the people who build it. If that is you, find him there.

Everything is on the table except client confidences and NDAs.

Resources & Further Reading

> Tools I actually use. People worth following. A short list, deliberately.

Tools I Use Every Day

This is not a sponsored list. These are the tools that have earned a permanent place in how I work — the ones that eliminated something rather than just improved it.

- **Claude (Anthropic)** — primary AI tool for writing, analysis, brainstorming, and code. The tool this book was largely assembled with.
- **Notion** — documentation, meeting transcription with auto-generated actions, project context. The closest thing I have to a second brain.
- **n8n** — workflow automation. Open source, self-hostable, and genuinely powerful once you understand it. The tool behind most of our client automation work.
- **Heyreach** — LinkedIn automation. Set and forget. The only outreach tool I have found that truly disappears into the background.
- **ClickUp** — project and task management for client work. Shared workspaces with clients as our transparency model in practice.
- **Loom** — async video communication. Replaces most of the meetings that would otherwise exist only to give someone context.

- **Power BI** — enterprise analytics and reporting. Still the right tool for most analytics work at scale.

People Worth Following

These are people whose thinking has shaped how I work. Not because they agree with everything I believe — often because they don't.

- **Alex Hormozi** — on offer structure, value creation, and the mechanics of building businesses that are not dependent on the founder's time.
- **Naval Ravikant** — on leverage, specific knowledge, and the compounding nature of working on the right things rather than harder on the wrong ones.
- **Paul Graham** — on startups, founders, and the specific ways that smart people fool themselves into building things that do not matter.
- **Simon Sinek** — on leadership and the difference between authority and trust. More relevant in technical teams than most people expect.

If You Want to Go Deeper

A short reading list for the ideas that most influenced this book's framework.

- **The E-Myth Revisited** — Michael Gerber. Why most people who start businesses are not actually running a business. The systems versus founder dependency distinction.
- **$100M Offers** — Alex Hormozi. How to think about value creation and why the quality of what you offer matters more than how hard you sell it.

- **Thinking in Systems** — Donella Meadows. The clearest explanation of why systems behave the way they do and how to intervene effectively rather than just symptomatically.
- **The Phoenix Project** — Gene Kim. A novel about IT delivery that captures the failure modes of technical teams better than any non-fiction book I have read.
- **An Elegant Puzzle** — Will Larson. Engineering management from someone who has actually done it at scale. Practical, honest, and not full of frameworks that sound good in theory.

Stay in Touch

The best outcome from this book is a conversation. If something resonated, if you disagreed with something and have a better version of it from your own experience, or if you just want to think through how this applies to your specific situation — reach out.

Ryan Richardson ryan@onwardsanalytics.com.au onwardsanalytics.com.au LinkedIn: Ryan Richardson — Onwards Analytics

Bonus Appendix — For Readers Inside Larger Organisations

> **Most of this book is written for the founder. But the gap opens inside big organisations too — and the questions get more specific. If you are accountable for technical outcomes inside an enterprise, being pushed toward AI, and you want straight answers without the jargon, this section is for you.**

These are the questions I get asked most often by people leading technical work inside larger organisations. The principles are the same six levers — but the vocabulary, the constraints, and the politics are different enough to warrant their own answers.

We are getting told we need to use AI. Where do we actually start?

Start with a problem, not a technology. Pick the thing your team does every day that is tedious, repetitive, and produces output that is close enough most of the time. Document processing, first-pass reporting, meeting notes, routine emails — that is where AI is ready right now. It is not where it sounds impressive. It is where it actually works. The mistake most organisations make is starting with the most ambitious use case because that is what gets executive sponsorship. Those projects require the highest certainty, the most validation, the most careful risk management. They are the hardest places to deploy

AI well. Start somewhere unglamorous and get a win. Then move.

We need to use AI to get more from our data — can you build it into our reports?

Depends what you mean. If you mean AI-generated narrative that summarises what the numbers show — that is available today and not complicated to build. If you mean AI that actually diagnoses what is wrong with your data, identifies anomalies, or tells you what to do about it — that is more complex and the quality varies enormously depending on your data quality underneath. The more useful question is: what does your team actually read and act on right now? Start there. Do not build AI on top of reports nobody opens.

What are other companies like us actually doing with this right now?

The honest answer and the press release answer are different things. The press release answer: AI-powered strategy tools, predictive maintenance platforms, autonomous decision systems. The honest answer: document search, meeting summarisation, first-pass report narrative, email drafting, maintenance scheduling in mining operations, and a lot of people quietly using general AI tools on their own because the official platform is not approved yet. The unglamorous applications are where the real productivity gains are happening. Most of the flagship projects are still in pilot.

Our IT team wants to move everything to the cloud. Do we actually need to do that?

Not necessarily. Cloud is more flexible and more expensive. It is not inherently better. The right question is: what problem are you solving? If your current infrastructure is limiting your ability to scale, breaking regularly, or costing more to maintain than to replace — then yes, moving to cloud probably makes sense. If it is working and your IT team wants to move because it is the modern thing to do — that is not a good enough reason. Cloud migration is disruptive and expensive. Do not do it because your vendor recommended it.

We have Power BI but nobody really uses it. Is that a tool problem or something else?

Almost always something else. Usually a context problem. The report was built for the person who requested it, not the person who needs to use it. The metrics it shows are not the metrics anyone actually makes decisions from. Nobody was consulted on what would be useful. It was built to the spec, delivered on time, and then sat there. The fix is not a better tool. It is going back and asking the people who are supposed to use it what they actually need to see. Then rebuilding. Probably takes a day. Should have happened before the build.

Our vendor keeps talking about a semantic layer and a data lake. What does any of that actually mean?

Semantic layer: a translation layer that sits between your raw data and your reports. It means "total revenue" always means the same thing whether you are looking at it in a dashboard or in an export. Without one, different reports can show different numbers for the same metric depending on how they were built.

You need it when you have multiple reports that need to agree on definitions. Data lake: a place to store large volumes of raw data before it has been cleaned or structured. Useful when you have data coming in from many sources that you want to keep for analysis later. Not useful if you just need clean operational reporting. Most organisations at your scale need a semantic layer before they need a data lake.

How do I know if what my data team is building is actually good?

Three questions that matter more than any technical review: Does it get used? If the people it was built for are not using it, it has failed regardless of how elegant it is technically. Does it change decisions? If people look at it and then do the same thing they would have done anyway, it is a reporting product, not a decision tool. Can someone else maintain it? If the person who built it left tomorrow and nobody could change anything without breaking it, you do not own the asset — you are dependent on the person. If all three answers are yes, it is good. If any are no, you know where to start.

We want to automate some of our reporting. Where do we even start?

Start with the report that takes the longest to produce and changes the least. Not the most important report. The most painful one. You want a win that is visible and contained. A report that takes someone three hours every Monday and could run automatically in ten minutes. That is your first automation. It builds confidence, it demonstrates value, and it teaches you where your data quality problems are — which you will need to fix before you automate anything more complex.

What is this actually going to cost and how long will it take?

Nobody can tell you accurately until they understand the problem properly. If someone gives you a number in the first meeting, treat it as a signal about their process not their expertise. What I can tell you: most things take longer than quoted and most quotes underestimate complexity. The gap between "it's almost done" and "it's actually done" is real and consistent. Budget a contingency of thirty percent over whatever you are quoted. If they hit it, great. If they do not, you have not been caught out. And get fixed-price milestones, not time-and-materials, wherever possible.

We keep starting things and never finishing them. Why does that keep happening?

Because finishing things is harder than starting them and most organisations do not have a system for the hard middle part. Starting a project generates enthusiasm, executive attention, and visible progress. The middle is where complexity surfaces, priorities shift, and the person who sponsored it gets pulled onto the next thing. Without a clear definition of done and someone accountable for getting there, projects drift. The fix is not better project management software. It is a shorter list of things you are actually committed to finishing, with one person accountable for each one, and a clear definition of what done looks like before you start.

I feel like I am always being told what we can't do. How do I get straight answers out of my tech team?

Tell them explicitly that you need to make a decision and you need to understand the constraints clearly enough to make it.

Not the technical detail — the actual options, the real tradeoffs, and what they would recommend and why. Most technical people default to explaining the complexity because they are worried about being held to something they cannot deliver. They are not trying to obstruct you. They are trying to be accurate. The way to change that is to make it clear that you are not asking for a guarantee — you are asking for their best assessment so you can move forward. If you have done that and still cannot get a straight answer, the problem is not communication — it is the person.

Someone sold us a dashboard six months ago. Nobody uses it. What went wrong?

Usually one of three things. The person who commissioned it and the person who was supposed to use it were different people and nobody checked if the right one was consulted. The data it relies on is unreliable enough that people stopped trusting the numbers. Or it was built to the requirements as stated rather than the problem as it actually exists. The fastest way to find out is to ask the people who do not use it why they do not use it. Not their manager. Them. The answer will be specific and it will tell you exactly what needs to change.

We have a lot of data sitting around. Should we be doing something with it?

Probably. But not in the way most vendors will suggest. The first step is not to build a data platform. The first step is to understand what decisions you are currently making badly because you do not have access to the right information. Start there. If you can name three decisions that would be better with

better data, you have a case for investment. If you cannot, the data sitting around is not yet a problem worth solving.

How do we know if a vendor is actually good or just good at selling?

Watch what they do when something goes wrong. Anyone can perform well when things are going well. The signal is how they communicate when there is a problem — do they tell you early and clearly, or do you find out late when the damage is done? Ask for references from projects that had problems, not just successful ones. Ask those references specifically how the vendor handled it when things went sideways. And pay small for something small first. The result of a small engagement is the only reliable indicator of how a large one will go. Sales presentations tell you nothing.

Chapter Workbooks

Each workbook corresponds to a chapter. Use them in order or go straight to the one that is most relevant to where you are right now. The questions in each workbook go deeper than the "Before You Move On" questions in the chapter — they are designed for a longer working session rather than a quick reflection.

You do not need to complete all of them. The most valuable thing you can do is pick the lever that is causing you the most pain right now and work through that workbook carefully. Then come back to the others when you are ready.

Workbook — Chapter 1: The Problem

Use this section to apply the ideas from the chapter to your actual situation. There are no right answers — only honest ones.

01 — Guiding Questions

- Think about your current technical environment. If you had to describe the baseline honestly — not the version you would say in a meeting, but the real version — what would you say?
- What is the work that is happening in your technical team right now that looks like productivity but probably isn't? Be specific.
- Where in your startup are smart, capable people achieving significantly less than they should — and what is the environmental reason for that, not the personal one?
- If your technical team's output multiplied by ten tomorrow, what would that actually look like? What would be different? What would you be able to do that you cannot do now?

02 — Identify Your Core Issue

Name the single biggest gap between the baseline you described and the ceiling you can imagine. Write it in one sentence.

03 — Simple Framework: The Baseline Audit

What is your team actually producing? For the last two weeks of work, list what was actually produced — not what was done, what was produced. Then run the 100x question: this is a thought experiment, not a planning exercise. Answer it honestly and see what it surfaces.

My Notes — Chapter 1

Workbook — Chapter 3: Context & Communication

Use this section to apply the ideas from the chapter to your actual situation. There are no right answers — only honest ones.

01 — Guiding Questions

- Think about a project that went sideways. At what point did the context gap actually open — and who was responsible for closing it?
- What is a requirement or brief you have recently given or received that felt complete but probably wasn't? What was the assumption hiding underneath it?
- When you are working through a problem with your technical team, what are the signals that tell you the picture is complete enough to start building?
- What is the equivalent of the 'systems versus features' question in your current environment? What keeps getting built as a feature that should have been designed as a system?

02 — Identify Your Core Issue

Name the assumption you are most likely carrying into your next brief without testing it.

03 — Simple Framework: The Context Conversation

The Two Circles. Use this before any new piece of work starts. Fill in each column honestly — the technical circle (what can be built, the constraints) and the user/business circle (the real problem, what good looks like). The overlap is what you are actually building.

The Context Completeness Test. Before you start building anything, answer these. If any feel uncertain, that is where to focus the next conversation.

My Notes — Chapter 3

> ***(The remaining chapter workbooks — Work Structure, The Right People, Location Arbitrage, Tools & Automation, Retention & Culture — follow the same three-part structure: Guiding Questions, Identify Your Core Issue, and a Simple Framework, each paired with space for your own notes. Use the "Before You Move On" questions at the end of each chapter as your starting prompts.)***

> **You're nodding along in a meeting you stopped understanding three minutes ago. Or you're the one talking, watching everyone nod, knowing they have no idea what you just said. Either way, you're on one side of a gap that is costing your startup more than anyone wants to admit.**

The Non-Technical CTO is for the founder who has an idea they can't get built, who is spending money on developers and unsure they're getting value, who is stuck at seventy percent and wants to finally ship — and who has decided to lead the technology side of their startup without becoming an engineer to do it.

Ryan Richardson breaks down the six places this gap reliably opens — and exactly what closing it looks like in practice. Not a methodology. Not a framework to hang on the wall. The specific things that change output, protect budget, and make technical work actually work — written by someone who does this for funded startups and tier-one enterprises alike, across seven businesses and over a hundred technical projects.

If you've ever approved an invoice you shouldn't have, hired someone who looked great on paper and delivered nothing, or watched a build go sideways in slow motion while everyone insisted it was fine — this book is for you.

The gap is predictable. So is the fix.

Before you close the book

You now know where the gap opens and how to close it. Knowing and doing are different problems. The levers work, but they take a steady hand and someone to tell you when you're about to make a mistake you can't see yet.

So here's the fork. If you want to build it yourself — with founders who are closing the same gap, templates that save you the trial and error, and people to pressure-test your decisions before they cost you — come build it with us.

If you'd rather skip the learning curve and get back a working system — the right people, the right structure, the right tools, built and handed over — have us build it for you. Book a call and we'll tell you straight whether it's a fit.

Build it with us → {{COMMUNITY_URL}}

Book a call, have us build it for you → {{ONWARDS_LABS_CALL_URL}}

bothsidesoftech.com