Beyond the Fractional CTO

The five accountabilities your company needs delivered, and how to get reliable execution without the hire.

A YOU-SOURCE BOOK

CONTENTS

18. Appendix C — Sources & Further Reading

Beyond the Fractional CTO

Why a fractional CTO isn't the answer: the five accountabilities a growing company actually needs delivered.

You're about to hire a CTO. Full-time, fractional, you haven't decided. Something in the engineering side of your company isn't holding, and the org chart has a hole shaped like a title, so you're filling it.

This book is going to argue that you've misread the gap.

A fractional CTO sells you advice by the hour. Advice doesn't ship software. What a growing company is actually missing is reliable execution: the five accountabilities a CTO owns, delivered consistently. You can buy that outcome without buying the headcount.

That's the whole thesis. The rest of the book earns it. We take the job apart into the five things a CTO is accountable for, software delivery, team performance, operational reliability, technical quality, and roadmap and business alignment. We show what each one looks like when it's working and how it fails when nobody owns it. Then we show how the work actually runs, in tasks instead of a thousand-page spec, on a cadence you can see. By the end you'll have ten numbers that tell you whether any of it is true.

What you won't find here is a pitch to stop hiring CTOs. Some companies need one. The question this book answers is narrower and more useful: what does your company actually need delivered, and what's the cheapest honest way to get it.

You run a company. You can read a roadmap and a KPI table without anyone running a standup for you. So we'll talk to you that way, the way you'd want a sharp CFO to explain engineering, with no jargon walls and no magic.

Outcomes, not org charts.

The Title Trap

The fractional CTO boom is a symptom, not a cure. You feel an engineering gap and reach for a title to fill it, when what's actually missing is shipped software.

A founder calls. Revenue is climbing, the product backlog is a swamp, and the two contractors who built the thing have started disappearing for days at a time. Something is wrong with engineering and nobody inside the company can name it precisely. So the founder does the reasonable thing, the thing every advisor and every LinkedIn post recommends. They go looking for a fractional CTO.

It feels like progress. There's a search, a few interviews, a name to put in the box on the org chart. The box gets filled. And then a quarter goes by, and the product backlog is still a swamp, and the founder is paying a senior rate for a person who shows up to two meetings a week with thoughtful opinions about the roadmap. The opinions are good. Nothing has shipped.

This is the title trap. You went shopping for a job title and came home without any working software.

The gap nobody named

Here's what the founder actually felt, before they translated it into a hiring problem: work wasn't getting done. Not "we lack technical vision." Not "we have no one senior in the room." Those might be true too, but they aren't what was burning. What was burning is that features were promised and didn't appear, the dates kept moving, and there was no one the founder could point to and say *that person is accountable for this getting built*.

Hiring is the default response because hiring is what you do when an org chart has a hole. But an org chart hole and a delivery hole are different animals, and the title trap is mistaking one for the other. You can fill the seat and still ship nothing. Plenty of companies have done exactly that, at considerable cost.

The pressure to fill the seat fast is real, and it's getting worse. Across all roles, the average time to fill a job rose from 43.6 days in 2022 to 59.7 days in 2025, a jump of about 37% (Greenhouse 2025). That's two months of searching before anyone starts, and that figure isn't even engineering-specific, where the pool is tighter. So a fractional arrangement looks like relief.

Someone senior, in the room, this week. The relief is genuine. It just doesn't fix the thing that was actually broken.

Advice by the hour

Look closely at what a fractional CTO sells, and the shape of the problem comes into focus. The model is engineered around seniority, not delivery. You buy a slice of an experienced person's week, and that person spends it on guidance.

None of this knocks the model. The model is doing exactly what it was built to do. A vendor in the space describes the fractional CTO as someone who "leans more heavily toward strategic guidance, oversight, and experience while eschewing the leadership role of managing teams" (gofractional). Read that line twice. *Eschewing the leadership role of managing teams*. That isn't a defect in one provider's offering; it's the definition of the category. The person you brought in to fix delivery is, by their own category's description, the person who does not run the team that delivers.

What you've bought is advice by the hour. What you needed was hands on keyboard. Between those two things sits the execution gap, the distance between a smart opinion in a meeting and a working feature in production. Nothing a retainer buys you will cross that gap, no matter how good the advice or how polished the roadmap deck. The only thing that crosses the execution gap is execution: someone owning a piece of work and shipping it, then owning the next piece and shipping that.

WHAT YOU'RE BUYING	WHAT IT PRODUCES	CROSSES THE EXECUTION GAP?
Advice by the hour (fractional guidance)	Roadmaps, opinions, architecture sketches	No
Hands on keyboard (delivery)	Shipped, working software	Yes

The cost framing reinforces the wrong instinct. Fractional is sold as the budget-conscious choice, often pitched at roughly 40–60% less than a full-time CTO (vendor estimate, unverified). Cheaper than a full hire, sure. But cheaper advice is still advice. You can halve the price of the thing you didn't need and you've still not bought the thing you did.

Define the seat before you fill it

None of this means a fractional CTO is a bad hire. For the right company at the right moment, a seasoned technologist setting direction is exactly right. The mistake isn't the option. The mistake is reaching for any title before you've defined what you actually need delivered.

So put the hiring instinct down for a chapter. The boom in fractional engineering leadership isn't spreading because it works. It spreads because a lot of growing companies feel the same gap and keep reaching for the same familiar lever. The lever is a title. The gap is execution.

That gap has a name we'll use for the rest of this book: the absence of **reliable execution**, the right things shipped predictably, week after week. It's the outcome the title was supposed to deliver and so rarely does. The bet of this book is that you can buy that outcome directly, without buying the headcount, and measure it when it arrives.

> *Outcomes, not org charts.*

Before you decide who fills the seat, define the seat. What is a CTO actually accountable for? That's where we go next, and the answer is more concrete than a title suggests.

Next: five accountabilities hide behind three letters. Name them, and the hiring question changes shape.

What a CTO Is Actually Accountable For

Before you decide who fills the seat, define the seat. Strip the title down and a CTO is five things that either get delivered or don't.

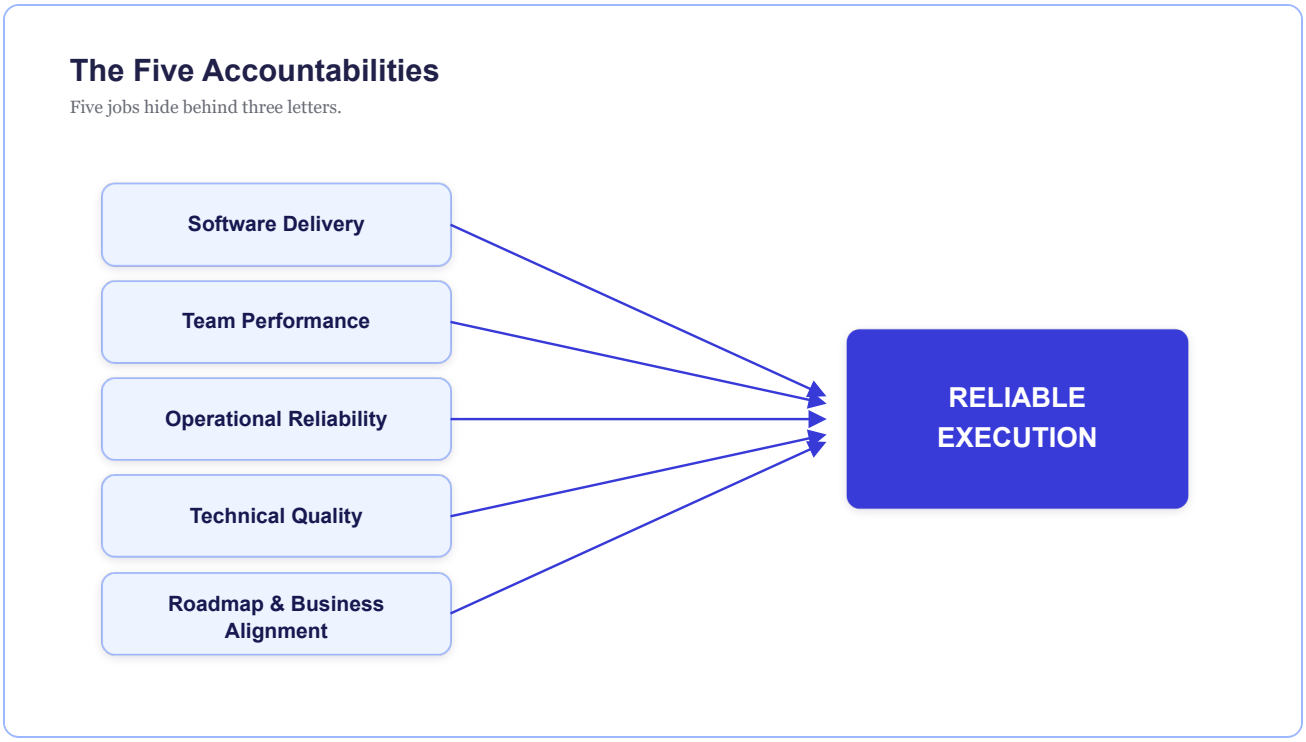
The last chapter left you with a question: you have a hole in the org chart, but what's actually missing? A job title doesn't answer that. "CTO" is a label, and labels are exactly what makes the hiring decision so hard. You can't interview for "technical leadership" the way you can interview for "ships a working checkout flow by the end of the month." One is a vibe. The other is a thing that happens or doesn't.

So let's do something the job ads rarely do. Let's take the role apart and look at what's inside it.

The role, decomposed

Underneath the title, a CTO is accountable for five outcomes. These are the five accountabilities, and they are the map for the rest of this book.

#	ACCOUNTABILITY	THE QUESTION IT ANSWERS
1	**Software delivery**	Does the right software ship, on a cadence you can count on?
2	**Team performance**	Does the work flow without heroes, bottlenecks, or a single point of failure?
3	**Operational reliability**	Does it stay up, and when it breaks, does it come back fast?
4	**Technical quality**	Does the codebase stay cheap to change instead of rotting?
5	**Roadmap & business alignment**	Is the work tied to a commercial outcome, or are you building features nobody asked for?



Read that table again and notice what each row has in common. Every one is phrased as a result, not a trait. None of them reads "is a strategic thinker" or "has gravitas" or "communicates a compelling technical vision." Each is instead a state of the world you can walk up to and check. The software shipped, or it sat in a branch. The system stayed up, or customers saw an error page at 9am. There's no personality test that tells you any of this in advance. There's only the work, after the fact.

That distinction is the whole game, so I'll say it plainly. **Each of the five is an execution outcome, not a personality you hire.** You don't need someone who *seems* like they could own reliability. You need reliability owned. The two are not the same purchase, and confusing them is how founders end up with an impressive hire and an empty production environment.

Why the public definitions mislead you

Go and read the authoritative descriptions of the CTO role. Splunk's, edstellar's, the Wikipedia entry. They cluster the job into roughly the same five or six areas you see above, which is reassuring. The decomposition isn't something I made up. It's how the people who study the role already carve it.

But watch where the emphasis lands. These descriptions lead with strategy, vision, and roadmap. They talk about "aligning technology investment with business strategy" and "translating technology capabilities into measurable business outcomes." Every bit of that is real and worth doing, and every bit of it sits upstream of the one thing they barely mention: software actually getting shipped.

> *The conventional definition over-weights advice and under-weights delivery. It tells you what a CTO should think about and goes quiet on what a CTO should make happen.*

This isn't an accident of one bad article. It's the gravitational pull of the title itself. "Chief" sounds like a thinking role, so the descriptions fill up with thinking, and the dull, decisive question slides to the bottom of the page. Does anything ship? For a 200-person company with a mature engineering org, the strategic framing is fine. For a growing company that hasn't put working software in front of customers in a quarter, it's a category error. You're being sold vision when what you're short on is output. That is the execution gap, named in the last chapter, showing up again as a flaw baked right into the job description.

The fractional CTO model inherits the same skew, only sharper. A fractional CTO leans toward strategic guidance and oversight while, in one vendor's own words, "eschewing the leadership role of managing teams." That's advice by the hour. It can be genuinely useful advice. It does not, on its own, move any of the five outcomes from "should happen" to "happened."

Each one ships, holds, or fails

Here's the test I want you to apply to all five for the rest of the book. Take any accountability and ask: what does it look like when nobody owns it?

Software delivery with no owner is stop-start. Dates slip without warning, a quarter passes, and nothing reaches production. Team performance with no owner is the bus factor of one, the bottleneck who can't take a holiday, the hire that never closes. Operational reliability with no owner is permanent firefighting and an outage nobody was watching for. Technical quality with no owner is a system the team is quietly afraid to touch. Alignment with no owner is a roadmap full of features that don't trace to a single dollar of revenue.

None of those failures are a leadership-philosophy problem. They're delivery problems. You don't fix a quarter of slipped dates with a better technology vision. You fix it with work, shipped, in a sequence you can see. Which means every accountability you just read is something you can hand to whoever, or whatever, can demonstrably move it. The title is one option. It is not the only one, and it is rarely the cheapest path to the outcome.

And because each is an outcome, each is measurable. That's not a throwaway line. Part IV of this book ends with a scorecard the business can read: two KPIs per accountability, ten numbers on one page. Lead time and delivery predictability for software delivery. Cycle time and key-person concentration for team performance. Change-failure rate and time-to-restore for reliability. Defect escape and rework ratio for quality. Percentage of work tied to a business outcome, and roadmap accuracy, for alignment. Hold that thought. It's the proof that none of this is soft.

The map for what's coming

Part II takes the five accountabilities one chapter at a time. Each chapter follows the same shape: what the accountability really means inside a growing company, what breaks when nobody owns it, and what reliable execution of it actually looks like in practice. By the end you'll have a working definition of the seat that has nothing to do with whose name goes in it.

There's one thing the public lists include that this book leaves out on purpose: innovation and R&D, the "explore emerging tech" mandate. It's a real part of a big-company CTO's job. It is also the wrong thing to optimise for when you haven't shipped reliably yet. A growing company needs a steady drumbeat of delivered software, not a research lab. Where strategy and forward-looking bets belong, they fold into roadmap and alignment, the fifth accountability, rather than standing as a sixth.

So that's the seat, defined as five outcomes you can measure rather than one title you can't evaluate. Now we go where the pain is loudest, and ask the most basic question of all.

Does anything actually ship?

Software Delivery

The first accountability is the one founders feel first: does anything actually ship? Not "are we busy." Not "is the team smart." Is working software reaching customers on a rhythm you can count on?

What it means

Software delivery is the plainest of the five accountabilities, which is exactly why it gets skipped in the conversation about who to hire. When a founder pictures a CTO, they picture strategy: architecture diagrams, a technology vision, a seat in the board meeting. Look at how the role is described in public, by the vendors who staff it and the consultancies who define it, and delivery is barely there. The descriptions skew to vision and oversight. Shipping is assumed.

That assumption is where companies get hurt. Delivery is the accountability for turning intent into something a customer can use, this week, and again the week after. Not a prototype. Not a demo that works on the founder's laptop. Software in production, doing the job it was built to do, on a cadence the business can plan around.

The cadence matters as much as the shipping. A team that delivers a big release every nine months, on a date that slips twice, is not delivering reliably even if the release is good. A founder can't sell against it, can't staff against it, can't promise a customer a date. What you want from this accountability is a drumbeat: small things, landing often, predictably enough that you stop wondering whether the next one will arrive.

The failure mode

Here is what it looks like when nobody owns delivery. The work is stop-start. A sprint produces something; the next two produce a refactor nobody asked for and a fortnight lost to an integration that "should have been simple." Dates are given with confidence and missed without explanation. You ask for a status and get a paragraph about complexity. A quarter ends and you realise that nothing a customer can touch went out the door.

This is not usually a story about lazy engineers. It is a story about size. The Standish Group's CHAOS data is blunt on this point: across modern software projects, just 29% land as outright successes, while 52% come in challenged (late, over budget, or stripped of scope) and 19% fail

outright (Standish CHAOS 2015). Those are sobering odds before you've written a line of code. The instinct, when delivery is shaky, is to plan harder and scope bigger so that "this time we get it right." That instinct is the trap.

Because the dominant predictor of whether a project ships is not the team's talent or the quality of the plan. It's the size of the bet. CHAOS found small projects succeed 61% of the time; "grand" projects succeed 6% of the time (Standish CHAOS 2015). Same firms, same engineers, wildly different outcomes. The variable that moved was how much you tried to ship at once. A company with a delivery problem has often, without noticing, talked itself into a grand project.

The other half of the failure is the tail. Look at large IT projects and the average cost overrun runs to 27% (Flyvbjerg & Budzier). An overrun you could plan for. What you can't plan for is the part the average hides: roughly one project in six becomes a "black swan," overrunning its budget by 200% and its schedule by some 70% (Flyvbjerg & Budzier). That's the one that doesn't make you late. That's the one that takes the runway with it. When a founder says they got burned by an engineering project, this is usually the project they mean.

> *The average overrun is the bill you can survive. The one-in-six catastrophe is the one that ends the company.*

What reliable execution looks like

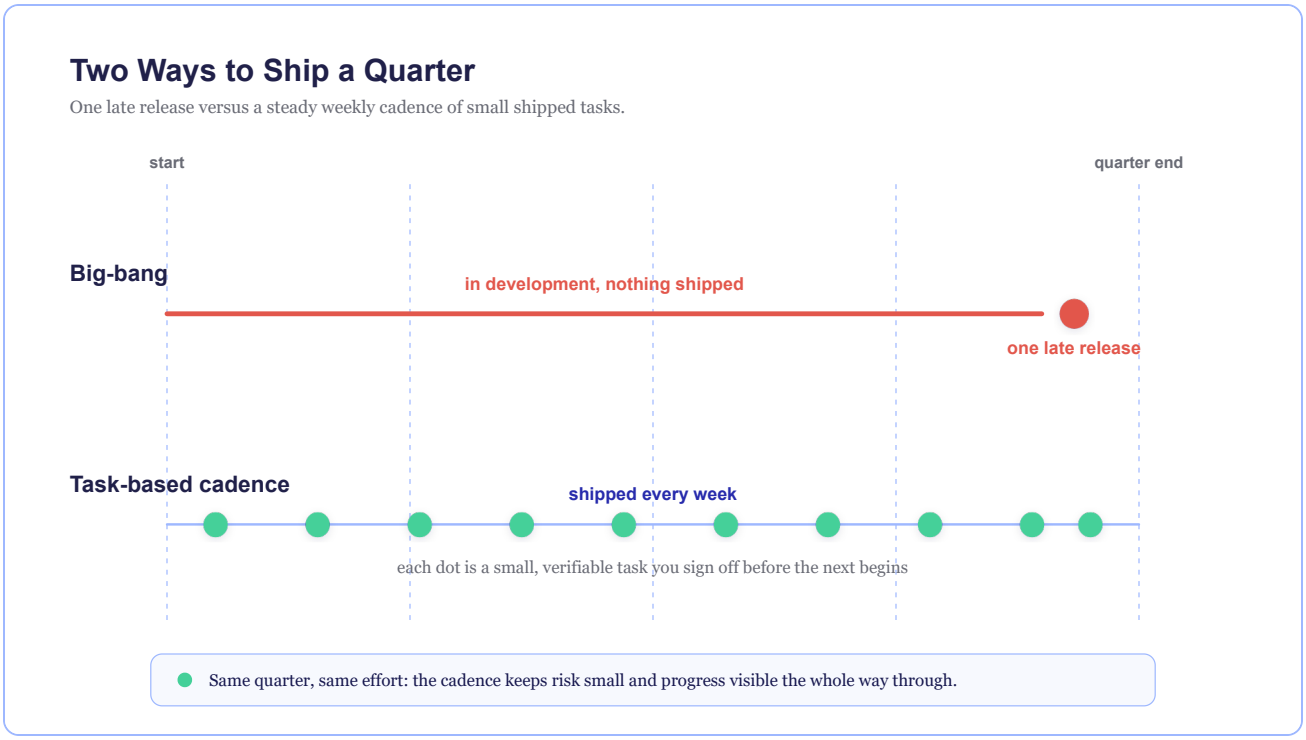
Reliable execution of software delivery is unglamorous, and that's the point. It looks like a steady stream of small, finished things. Each piece of work is scoped so it can ship on its own. Each has a clear definition of done, so "finished" is a fact you can check rather than a feeling someone reports. Nothing in the pipeline is so large that its failure would take a quarter with it.

Concretely, the unit of work is the task: small enough to ship in days, scoped tightly enough that you can verify it landed. A founder doesn't need to read the code to know a task is done. They need to see the thing it was supposed to do, working. That's the whole bargain. Make the work small, make "done" checkable, and the CHAOS odds start moving in your favour, because you have stopped betting the company on one grand release and started making a series of small bets you can win.

What does that look like on the ground? A shared view of what's moving, like this:

TASK	STATUS	DEFINITION OF DONE	SHIPPED
Add CSV export to the reporting page	In review	User can download a filtered report as CSV; matches on-screen totals	—
Email receipt after checkout	Shipped	Customer receives a branded receipt within 60s of payment	Tue
Fix duplicate-charge edge case	In progress	No customer is charged twice on a retried payment; covered by a regression check	—
Search by customer reference	Backlog	Support can find an order by reference in under 5s	—

You don't have to manage that board to read it. You can see what's moving, what's stuck, and what shipped this week without sitting in a single standup. Predictability comes from the same place reliability does: small units, visible state, a definition of done you can hold someone to.



How Dev on Demand covers it

This is the shape Dev on Demand is built around. Work comes in as a task, one engineer ships it, and you approve it before the next one begins. The cycle is short by design, a few days per task, with the approval gate sitting between every task and the next. You're never staring at a quarter-long black box wondering what's inside it; you're looking at the last thing that shipped and deciding what ships next.

That's not a claim that delivery becomes effortless. It's a claim about where the risk goes. When the unit of delivery is a small, verifiable task and you sign off on each one, the grand-project failure mode has nowhere to form. You can't accidentally drift into a 200% overrun when the longest thing you've committed to is a few days of work you'll inspect before continuing. The odds, the ones CHAOS measured, are quietly on your side again.

Delivery runs on people. So the next question is the obvious one: who's actually shipping, and what happens the day they leave?

Team Performance

A team that ships fast is worth nothing if the shipping stops the day one person takes a holiday. This chapter is about throughput you can rely on, and the quiet risk that hides underneath it.

What it actually means

Team performance sounds like a question about speed. How much does the team get done, and how quickly? That is half of it. The other half is whether the speed survives contact with reality: someone resigns, someone burns out, someone is on a flight with no Wi-Fi when the thing they alone understand breaks.

So the accountability is two things at once. Throughput, which is how much working software the team turns out over a given week. And resilience, which is whether that throughput holds when a person leaves the room. A growing company needs both. Plenty of founders only discover they were missing the second one on the morning it costs them.

Here is the trap. The faster a small team moves, the more tempting it is to let one strong engineer carry everything. They are good, they are quick, they say yes, and the work flows through them. For a while it looks like excellent performance. What you are actually watching is a single point of failure being loaded up, one task at a time, until it is holding the whole product on its back.

The failure mode: bus factor of one

The phrase is grim on purpose. **Bus factor of one** means exactly one person can keep a critical thing running, and if that person is hit by a bus, the thing stops. You do not need an actual bus. A resignation, a sabbatical, a competing offer, two weeks of flu, a baby. Any of them will do.

You can recognise the pattern without reading a line of code. Every awkward question routes to the same name. Deploys wait for one calendar. Nobody else can explain how billing works, or why the nightly job runs at 2am, or what that one environment variable does. The team has a hero, and the hero is tired.

Hero culture feels like strength from the outside. Inside, it is fragility wearing a cape. The hero becomes a bottleneck because everything needs their sign-off, then a flight risk because they are doing the work of three people, then, eventually, an outage, because they finally took the holiday they were owed and the thing they alone understood fell over while they were gone.

The other half of this failure mode is hiring that never closes. The founder's instinct, when the hero is drowning, is to hire more people. Reasonable. The problem is timing. Hiring is slow, and it has been getting slower. Greenhouse's benchmark data puts the average time-to-fill at 59.7 days in 2025, up from 43.6 days in 2022, a rise of about 37% (Greenhouse). That figure is all roles combined, not engineering specifically, and senior technical roles tend to sit at the long end of any such range. So call it two months at a minimum to even sign someone, before they have written anything you can ship, and before anyone has shown them where the bodies are buried.

Two months is a long time to keep a tired person upright. And a wrong hire under that pressure does not save you. You pay the first-year salary, the ramp time, and then, if it does not work, the cost of unwinding it and starting the search again. Nobody has a clean, citable dollar figure for that, and you should distrust anyone who quotes you one to two decimal places. The point stands without the number: a rushed senior hire to relieve a bus-factor problem can deepen the problem instead of fixing it.

What reliable execution looks like

Good team performance is boring to watch, which is the whole idea. Work moves through more than one pair of hands. Knowledge lives in places other than someone's head. The team's output this week looks a lot like its output last week, and you can plan around that.

The mechanism that gets you there is the same one this book keeps returning to: **the task**. When work is broken into small, scoped, verifiable units, no single task is large enough to belong to only one person forever. Each task carries its own context. Each one is reviewed by someone who did not write it. That review is not bureaucracy. It is the cheapest insurance you will ever buy against bus factor, because the reviewer learns the thing too, just by checking the work.

Look at the difference in a single ticket. A vague instruction concentrates knowledge in whoever happens to pick it up. A scoped task spreads it, because the context is written down where the next person can read it.

```
TASK-318  ·  Add per-seat billing to the admin dashboard

Context:    Customers on the Team plan are billed per active seat.
            The current dashboard shows a flat plan price only.
            Seat counts already exist in the `subscriptions` service;
            this task surfaces them, it does not change billing logic.

Scope:      - Show current seat count and per-seat price on the billing page
            - Recalculate the displayed total when seats change
            - Out of scope: invoicing, proration, anything that writes to Stripe

Definition  - Billing page shows seats x price = total, matching the
of done:      subscriptions service for three test accounts
            - Reviewed by someone who did not write it
            - Notes added to the billing section of the team wiki

Acceptance: Founder/PM can open the billing page for a Team-plan
            account and read the correct per-seat total without asking
            an engineer what it means.
```

Notice what that ticket does to key-person risk. The context is on the page, not in a head. The "reviewed by someone who did not write it" line forces a second person to understand it. The wiki note means a third can find it in six months. None of that is heroics. It is just work shaped so that knowledge has somewhere to go other than one overloaded brain.

Throughput follows from the same discipline. You measure it in finished tasks, not heroic weeks, so a fast fortnight from one exhausted person no longer reads as health. You watch how concentrated the work is: if every task in a fortnight has the same name on it, that is a number trending the wrong way, and you can see it before it becomes an outage. Part IV turns both of these into KPIs you can read on one page, cycle time per task and a measure of key-person concentration. For now, the point is that performance you can rely on is performance that does not depend on any one person staying.

	HERO TEAM	TEAM THAT SHARES
Who can ship a given change	One named person	Anyone who picks up the task
What happens on a two-week absence	Delivery stalls	Floor holds, pace dips slightly
Where knowledge lives	In someone's head	In the ticket, the review, the wiki
Founder's view of progress	"Ask Sam"	A board and a cycle-time number
Reaction to overload	Hire fast, hope	Add a stream, work already flows

Where this lands

This is the gap a title rarely closes on its own. Putting "Head of Engineering" on an org chart does not by itself spread knowledge across the team, balance the load, or close a two-month hire. Those are execution outcomes. They come from how the work is shaped and run, day to day, not from the seniority of the person nominally in charge of it.

Dev on Demand is built around the unit that does the spreading. Work arrives as tasks, each one scoped, delivered, and approved before the next begins, with the context written down rather than carried in one person's memory. Add a second stream when you need more throughput and the work keeps flowing, because it was never bottlenecked through a single hero to start with. We will come back to the full case in Part IV. The mechanism is the point here: throughput you can count on, without betting the company on one person's continued good health.

A team that ships fast and never stops still has to keep what it shipped running, which is the next accountability: operational reliability.

Operational Reliability

A team that ships fast still has to keep what it shipped running. Reliability is the accountability nobody talks about in the interview and everybody feels at 2 a.m.

You can ship the right things, on cadence, and still lose customers. The software went out, the feature worked in the demo, and then a payment endpoint started timing out on a Tuesday afternoon and nobody noticed until a customer emailed. By the time someone looked, it had been down for three hours. That is the gap this chapter is about. Software delivery gets the work into production. Operational reliability is whether it stays there, behaves, and recovers when it doesn't.

For a growing company this matters more than it did a year ago, because more is riding on the system staying up. When you had ten customers, an outage was an apology. With a few hundred paying users, an outage is churn, refunds, and a support inbox you can't clear. Reliability is the accountability that scales its consequences quietly while you're busy looking at the roadmap.

What it actually means

Strip the jargon and operational reliability is three plain questions. Does it stay up? When something breaks, do you find out before your customers do? And when it breaks, how fast are you back to normal? A CTO is meant to own all three. The edstellar role description puts it bluntly: the role is "accountable for system uptime and reliability… scale without buckling." That is an outcome, not a personality trait. You can measure whether it's being delivered.

Most founders only encounter reliability as a string of bad days. The site is slow. A background job silently stopped running last week and no one knew until the monthly report came out wrong. A deploy on Friday afternoon took the login page down over the weekend. Each one feels like bad luck. Strung together, they are a signal that nobody owns the question of whether the thing keeps working.

The failure mode: permanent firefighting

Here is what it looks like when the accountability has no owner. Things break at random. Each break is a scramble, found by a customer, fixed by whoever happens to be awake, with no record of what went wrong or whether it can happen again. There is no on-call rotation, so "on call"

means the one engineer who knows the system, all the time, forever. That's the **bus factor of one** from the last chapter, except now it's load-bearing at midnight.

Nothing gets written down because everyone is too busy putting out the current fire to prevent the next one. The same outage happens twice because the first one was patched, not understood. Over a quarter this hardens into a culture: the team stops shipping new work because it's spending its days nursing the old work back to health. Reliability didn't fail loudly. It bled the delivery cadence dry one incident at a time.

> *When you learn about an outage from a customer email, you didn't handle the incident. The incident handled you.*

The tell is the absence of two things: no one knows when the system is unhealthy until a human complains, and no one can say how long the last outage lasted because nobody was counting.

What reliable execution looks like

Good reliability is mostly boring, which is the point. It rests on a few practices that have nothing to do with heroics.

You watch the system, not the inbox. Something automated checks that the important paths still work, on a schedule, and tells you the moment one fails. Detection moves from "a customer noticed" to "a check noticed," and that shift is most of the battle. The faster you know, the smaller the blast radius.

You have a plan for who responds. An on-call rotation means the answer to "the payment service is down" is a name and a runbook, not a frantic group chat at 11 p.m. hoping someone sees it. None of this is ceremony for its own sake. The whole point is that recovery never depends on luck about who happens to be looking at their phone.

You learn from every incident. After something breaks you write down what happened, why, and what stops it recurring. This is the difference between a system that gets more reliable over time and one that breaks the same way every few weeks. A short, blameless write-up is enough.

Here's a sample of the artifact that closes the loop. After an incident, the follow-up work becomes a task like any other, scoped and shippable.

FIELD	ENTRY
Task	Add health check + alert on payment webhook endpoint
Context	On 4 June the payment webhook silently failed for ~3 hrs; found via customer email. No automated check covered this path.
Scope	Add an uptime check that exercises the webhook every 60s; alert the on-call channel on two consecutive failures.
Definition of done	Check is live; a forced failure pages on-call within 2 minutes; runbook entry added for "webhook down."
Acceptance check	Trigger a test failure in staging; confirm alert fires and the runbook link resolves.
Not in scope	Rewriting the webhook handler (separate task if root cause warrants).

Notice what that ticket does. It turns a 2 a.m. scramble into a scoped piece of work with a clear finish line. Reliability here is no separate discipline bolted on the side. It's the same task-based execution you already run, pointed at keeping things up.

Two numbers worth knowing

You don't need to run the system to read whether it's reliable. The industry has converged on a small set of measures through the DORA research program (the four key metrics behind the *Accelerate* work), and two of them speak directly to this accountability.

Change-failure rate is the share of your deployments that cause a problem in production: a rollback, a hotfix, an outage. It answers a question every founder should ask. When we ship, how often do we break something? A low rate means change is safe and the team can move without flinching. A high one means every release is a coin toss, and people start shipping less to avoid the risk, which quietly kills your delivery cadence.

Time to restore is how long it takes to get back to normal once something does break. Failures happen to everyone. What separates a reliable operation from a fragile one is whether recovery takes four minutes or four hours. A system that fails often but restores in minutes can feel rock-solid to a customer. A system that fails rarely but stays down for an afternoon does not.

Read together, these two tell you the whole story. One asks how often you break things, the other asks how badly it hurts when you do. You want both low, and you want someone

accountable for moving them. We'll formalise these alongside the rest of the scorecard later in the book. For now, hold the concepts: how often, and for how long.

METRIC	THE QUESTION IT ANSWERS	WHY A FOUNDER CARES
Change-failure rate	When we ship, how often do we break something?	High rate means the team ships less to avoid risk — the cadence dies
Time to restore	When we break something, how fast are we back?	Fast recovery is the difference between a blip and a churned customer

How this gets covered without a title

The thing to see is that reliability is execution, not advice. A fractional CTO can tell you that you ought to have monitoring and an on-call process. You knew that. No recommendation meets the accountability. What meets it is the health checks being live, the alerts being wired, the runbook being written, and the follow-up task being shipped. That is **hands on keyboard**, not **advice by the hour**.

Run as task-shaped work, reliability stops being a heroic property of one exhausted person and becomes a set of small, verifiable tasks: add this check, wire that alert, write this runbook, fix the root cause that caused last week's incident. Each one ships, each one you can see, and the system gets steadier as they land. You don't buy a title that's supposed to keep the lights on. You buy the work that actually does.

Reliability today is paid for by quality decisions made months ago, which is where we go next.

Technical Quality

The fourth accountability is the one you can't see from the outside: whether the system you've already built stays cheap to change, or quietly turns into a thing nobody dares touch.

You can ship on a steady cadence, keep a team that doesn't burn out, and stay up around the clock, and still be in trouble. The trouble lives under the surface. It shows up the day a small feature that should have taken two days takes two weeks, and nobody can quite explain why. The explanation is technical quality, or the lack of it. The codebase has started to fight back.

This is the accountability founders understand least, because it's the only one with no obvious customer-facing symptom until it's expensive. Delivery, you can watch. Reliability, your users report. Quality rots silently. By the time you feel it, you're paying interest on a debt you didn't know you'd taken on.

What it actually means

Technical quality is a measure of one thing: how much it costs to change the system tomorrow. A high-quality codebase is one where a sensible change is a small change. A low-quality one is where every change ripples, breaks something three modules away, and demands a day of testing to prove you didn't make it worse.

Two forces eat quality over time. The first is **technical debt**: shortcuts taken to ship faster, which is often the right call, until the bill comes due and nobody's been paying it down. The second is **security**, the slow accumulation of unpatched dependencies, weak handling of credentials, and access nobody's reviewed since the company was a third its current size. Both are invisible right up to the moment they're catastrophic. Debt becomes the rewrite. Security becomes the breach notification.

Here's the distinction that matters for you as a non-engineer: neither of these is a one-time event you can buy your way out of. They're not a project. They're a *practice*: a set of decisions made, or skipped, on every single piece of work. That's why this is an execution accountability, not an advice one. A consultant can tell you your debt is high. Only someone hands on keyboard pays it down.

The failure mode: the system everyone's afraid to touch

Nobody decides to let quality rot. It happens by accumulation, one reasonable shortcut at a time, each defensible on the day it was taken. Then one morning you ask for a change that sounds trivial and your best engineer winces.

The wince is the tell. It means the change touches a part of the system that's tangled, undertested, or understood by exactly one person who may or may not still work for you. You've heard the name for that last risk already in this book: the **bus factor of one**. In a low-quality codebase, the bus factor isn't a property of the team. It's baked into the code itself. The system has memorised who can safely change it, and the answer is "almost no one."

The numbers on what happens next are not kind. Flyvbjerg and Budzier, looking across 1,471 IT projects, found an average cost overrun of 27%. The figure that should worry you is the tail: **one in six runs 200% over budget with a roughly 70% schedule overrun** (Flyvbjerg & Budzier, HBR 2011). The average is bad. The tail is where companies die. And the projects that fall into the tail are rarely the ones that were badly planned on day one. They're the ones built on a foundation that had already quietly rotted, so every estimate was wrong before anyone wrote it down.

When quality goes unmanaged, the symptoms compound. Estimates stop meaning anything, because no one can predict what a change will disturb. Good engineers leave, because nobody enjoys working in a codebase that punishes them. The remaining ones grow conservative, refusing changes that should be routine, and the roadmap slows to protect a system that's too fragile to evolve. You end up running a company that can no longer change its own software. That is the quiet death this chapter is about.

What reliable execution looks like

The fix is not heroic. It's the opposite. Quality holds when it's maintained deliberately, in small increments, as a normal part of doing the work, rather than deferred until it demands a rescue.

Standish found that small projects succeed 61% of the time against just 6% for the largest ones (Standish CHAOS 2015). The same logic that governs how you *ship* governs how you *maintain*. Small, scoped, verifiable changes are the ones you can reason about, test, and reverse if they go wrong. A quality practice is just that discipline pointed at the existing system: you pay debt down in the same small units you use to add features, and you never let any single change grow so large that nobody can see the bottom of it.

What does that look like in practice? A definition of done that includes more than "it works on my screen." Security treated as a checklist item on routine work, not a panicked audit after an incident. And a small, visible budget for paying down debt, a slice of every cycle spent leaving the system a little cleaner than you found it, so the cost-to-change line stays flat instead of climbing.

Below is what a debt-paydown task looks like as a ticket. It's the same shape as any other task in this book, which is the point: maintenance isn't a separate, scary category of work. It's just work, scoped the same way.

FIELD	ENTRY
Title	Replace ad-hoc auth checks in the billing module with the shared permission layer
Context	Billing has three hand-rolled permission checks that drifted from the rest of the app. Each new billing feature has to re-implement them, slowing delivery and creating two known access bugs.
Scope	The billing module only. Migrate its three checks to the shared layer. No new features. No changes outside billing.
Definition of done	All three checks use the shared layer; the two known access bugs are gone; existing billing tests pass; one new test covers the previously unguarded path.
Acceptance check	A non-billing user is correctly denied on all three paths; a billing user passes; reviewer confirms no duplicated permission logic remains in the module.

Notice there's no application code on the page, and you don't need any to evaluate it. You can read the context, you can see the scope is bounded, and you can check the acceptance criteria were met. That's the founder's whole job on technical quality: not to assess the code, but to insist that the *practice* exists and is visibly running on every cycle.

How this gets covered

When you buy reliable execution rather than a title, quality stops being something you hope the senior hire cares about and becomes something the operating model enforces. Task-based delivery already works in small, scoped, verifiable units, with a definition of done and an approval gate before the next task begins. That's the same machinery that keeps quality from rotting, applied by default. Debt gets paid down as tasks, not as an annual emergency. Security shows up in the acceptance check, not in a breach notification. The codebase stays cheap to change because cheap-to-change is built into how every change is made, and you can see it ticket by ticket without ever reading a line of the code itself.

Quality only earns its keep if you're building the right thing in the first place, and that's not an engineering question at all.

Roadmap & Business Alignment

The fifth accountability is the one that decides whether the other four were worth anything: are you building the thing the business actually needs? Ship fast, ship clean, ship reliably, and still ship the wrong product, and you've executed beautifully into a wall.

A team can clear every other bar in this book and still fail here. The code is sound. The cadence holds. Nothing's on fire. And six months in, the founder looks at what got built and realises half of it serves no customer, no deal, and no number on the board. Engineering was busy. The business didn't move.

That's the alignment failure, and it's quieter than the others. An outage announces itself. A slipped launch announces itself. Building the wrong thing announces nothing. It feels like progress right up until you ask what the work was for.

What alignment actually means

Roadmap and business alignment is the discipline of making sure every piece of engineering work traces back to a commercial outcome. Not a vibe, not a stakeholder's hunch, but a specific thing the business is trying to do: win a segment, cut a support cost, unblock a renewal, hit a compliance deadline before it costs you a contract.

Public descriptions of the CTO role call this "translating technology capabilities into measurable business outcomes" (Splunk). It sounds like strategy, and the strategy part is real. But the accountability isn't having the roadmap. It's that the work flowing through the team this week actually maps to it. Plenty of companies have a roadmap. Far fewer can tell you, task by task, why each thing in flight is being built and what it's worth.

The reason this lands on engineering at all is that the gap between "what the business wants" and "what gets built" is where value leaks out. A founder says "we need to reduce churn." That sentence becomes a feature, the feature becomes a backlog of work, the backlog gets handed to whoever's free, and four steps later someone's building a settings page nobody asked for because it seemed related. Each handoff loses a little intent. Alignment is the work of not losing it.

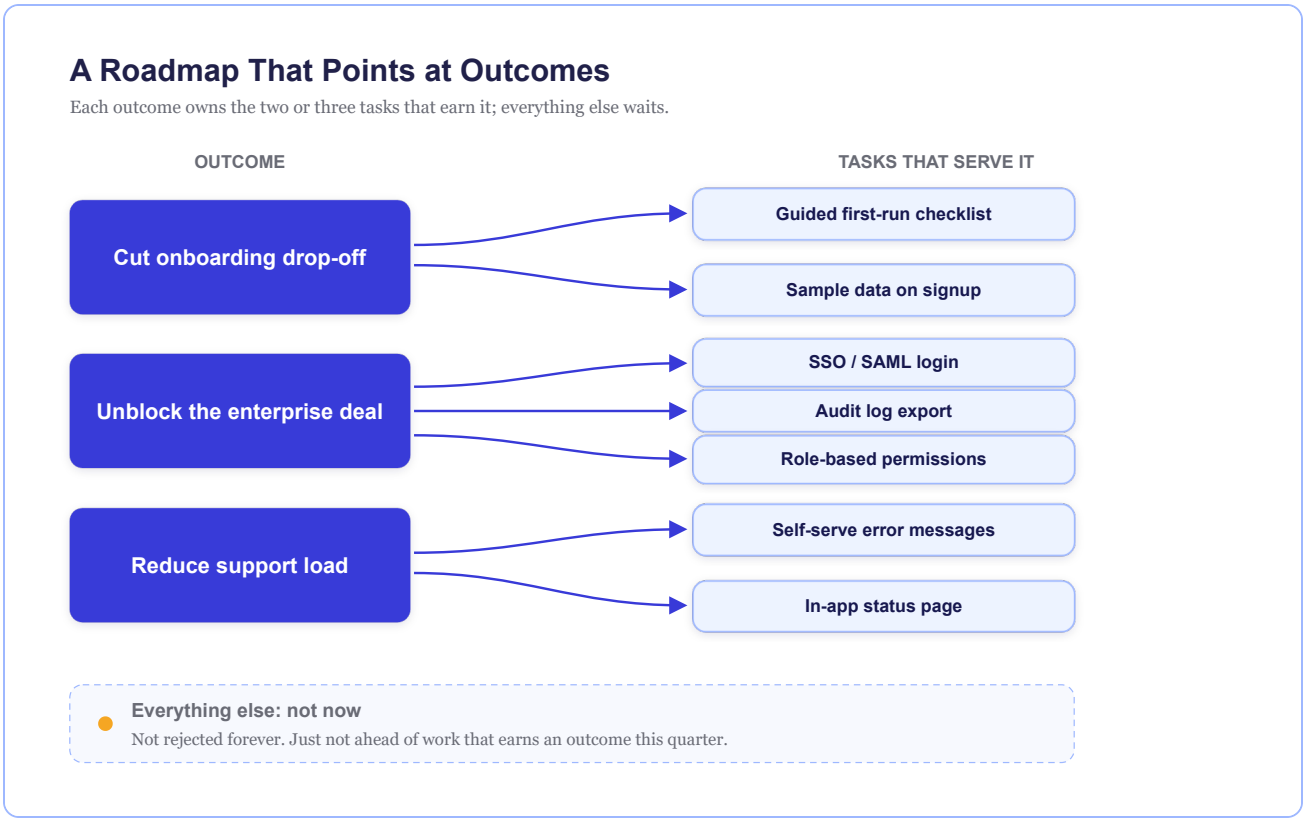
The failure mode: building things nobody uses

Here's the number that should worry any founder funding an engineering team. Pendo, which instruments how people actually use software, found that **80% of features are rarely or never used, and roughly 12% of features drive 80% of usage** (Pendo 2019). Four-fifths of what gets built barely gets touched.

Sit with that as a spend, not a statistic. Every one of those unused features was scoped, designed, built, tested, and shipped. Someone's salary paid for it. It occupies code that now has to be maintained, secured, and worked around forever. The company paid full price to build something and is now paying rent to keep it alive, and customers don't care that it exists.

This is the **spec graveyard** seen from the business side. A thousand-page specification doesn't just go obsolete and unread. It commits the team, up front, to building a pile of features that sounded essential in a planning room and turn out to be the 80% nobody opens. The bigger the upfront commitment, the more of it you discover, too late, you didn't need.

When nobody owns alignment, the symptoms are specific. Engineering ships steadily and revenue doesn't notice. The roadmap is a list of features rather than a list of outcomes. Sales asks for something and can't get an answer on whether it's coming. The founder can't look at the current sprint and say, in one sentence, what each item is supposed to earn. The team is executing. It just isn't executing toward anything the business can bank.



What reliable execution looks like

Aligned execution starts before a line of work is scoped. It starts with the question every task has to answer: *what business outcome does this serve, and how will we know it worked?* If a piece of work can't answer that, it doesn't get built yet. Not rejected forever. Just not now, not ahead of work that can.

The mechanism is plain. Each unit of work carries its business case on its face. The founder, who is not an engineer and shouldn't have to be, can read why a thing is being built without anyone translating. Here's that discipline as a task ticket:

FIELD	ENTRY
Task	Add saved-search to the customer portal
Business outcome	Cut support tickets from enterprise accounts re-running the same queries (top-3 ticket category, ~40 tickets/mo)
Who asked / evidence	Support lead; ticket tags pulled from last quarter
Definition of done	Logged-in users can save, name, and re-run a search; saved searches persist across sessions
How we'll know it worked	That ticket category drops month over month after release
If we skip it	Support load stays; renewal risk on two named accounts cites "clunky reporting"

Notice what the ticket forces. It names the outcome before the build. It points at evidence instead of opinion. It states how you'll measure whether the bet paid off. And it makes the cost of *not* doing it explicit, which is how you decide what's actually next. A founder can read that ticket in fifteen seconds and know exactly what their money is buying.

Run every task through that filter and the roadmap stops being a feature list and becomes a sequence of bets, each tied to a number. Some bets miss. That's fine. The point isn't a perfect hit rate. The point is that you find out fast, because every task was small enough to ship and measure on its own, and the misses get cut before they become the 80% that just sits there.

> *A roadmap reads like a promise to build a list. Treat it instead as a running argument about what's worth building next, settled one shipped task at a time.*

This is also where small, **task-shaped work** quietly does the alignment job for you. When the business changes its mind in month three, and it will, a team mid-way through a monolithic spec has to either plough on building the now-wrong thing or eat the cost of tearing it up. A team working task by task just points the next task somewhere else. The world moved; the next

bet moves with it. So alignment stops being a document you write once and becomes a decision you re-make every time you pick up the next piece of work.

Where this lands in practice

For a founder, the test of alignment is uncomfortable and simple. Pull up whatever's being built right now and, for each item, say out loud what business outcome it serves and how you'll know. If you can do it for everything, alignment is owned. If you stall on a few, those are your candidates for the 80%: work in flight that nobody can connect to a result.

Dev on Demand is built so the connection can't go missing. Work arrives as discrete tasks, each one approved by you before the next begins. That approval gate is the alignment check made routine: nothing proceeds until the person who owns the commercial picture has looked at the next task and agreed it's worth doing. You don't need a thousand-page spec or a standing meeting to keep engineering pointed at the business. You need each task to earn its place, one at a time, where you can see it.

Five accountabilities, one map. The next question is how you actually run the work so all five get delivered, and it doesn't start with a spec.

The Spec Graveyard

The thousand-page specification promises certainty and delivers a tombstone. Here's why it fails, and what replaces it.

Somewhere in your shared drive there is a document nobody opens. It has a version number in double digits, a table of contents four levels deep, and a sign-off page with three signatures on it. It cost a quarter of someone's salary to produce. When the project finally shipped, the team built from memory and hallway conversations, not from the document. The spec sat there the whole time, authoritative and ignored. That's the spec graveyard, and most growing companies have one.

The instinct behind it is reasonable. You're about to spend real money building software, you can't write the code yourself, and you want to know what you're getting before you commit. So you ask for everything to be written down first. Every screen, every rule, every edge case, agreed and signed. It feels like control. It feels like the responsible thing a founder does with other people's money and their own runway.

The problem isn't that the spec is too detailed. The problem is timing. A big upfront specification asks you to make your most consequential decisions at the moment you know the least. You haven't shipped anything, no customer has touched it, and your understanding of the problem is at its thinnest. You commit it all to paper anyway, because the document demands answers in boxes that can't be left blank. Then reality arrives.

The document is obsolete before it ships

Markets move. A competitor launches the feature you scoped for Q3. A regulation changes. Your best customer asks for something nobody anticipated, and it turns out to matter more than half of what's in chapters four through nine. The world keeps editing itself while your specification stays frozen at the date it was signed. By the time the build catches up to page 600, the early pages describe a company that no longer exists.

This is the quiet violence of the big spec: it converts learning into rework. Every genuine thing you discover while building, the discoveries that should be your most valuable asset, now contradicts a document somebody approved. So a discovery doesn't feel like progress. It feels like a problem. It triggers a change request, a re-estimate, an awkward conversation about

scope. The spec turns the act of getting smarter into a cost centre. A process that should reward learning starts punishing it.

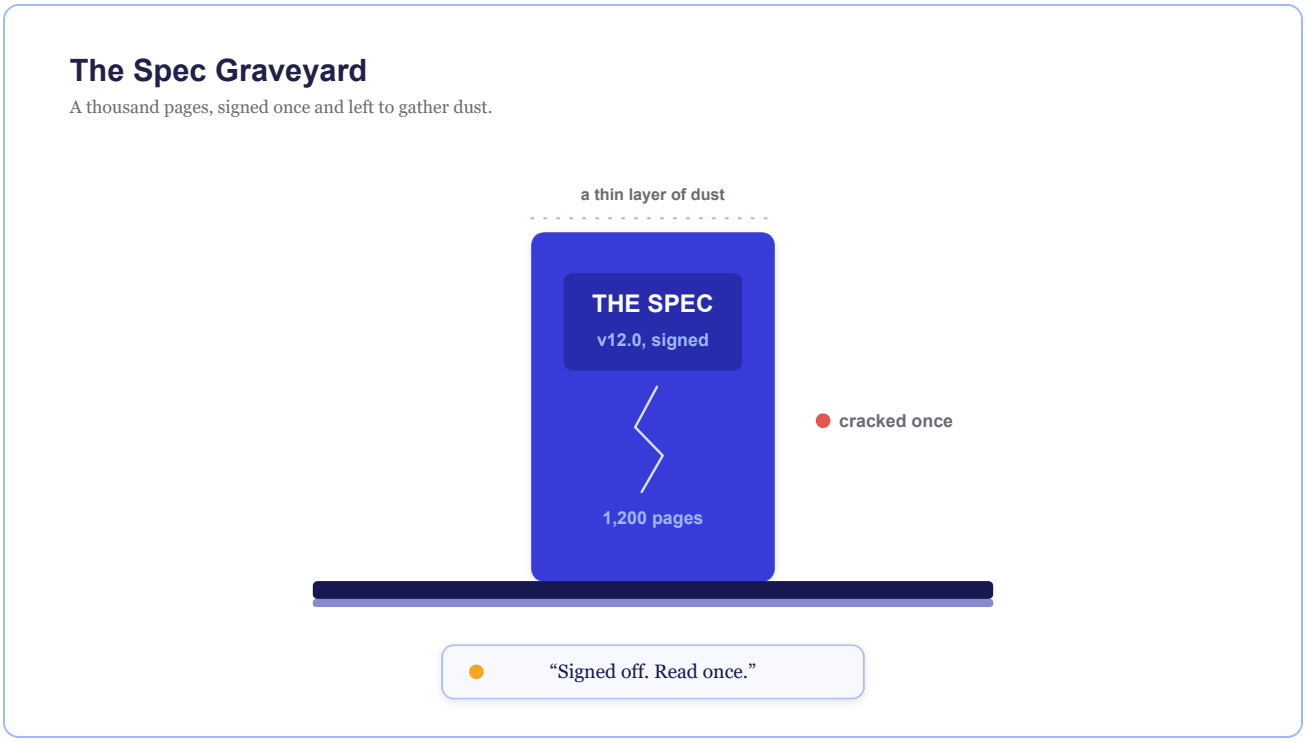
> *A specification frozen at the start of a project is a bet that you understood the problem perfectly before you'd seen it work. You didn't. Nobody does.*

And the rework isn't free. Across 1,471 IT projects, the average cost overrun ran to 27%, but the average hides the real danger: one in six projects blew through budget by 200% and ran roughly 70% over schedule (Flyvbjerg & Budzier, HBR 2011). Those aren't projects that were managed carelessly. They were projects that committed hard and early to a plan, then discovered the plan was wrong somewhere around the point of no return. The fat tail is where the spec graveyard buries you. Not the average miss. The catastrophe nobody could course-correct out of, because the whole apparatus was built to resist change.

Nobody reads it anyway

Here is the part that should bother you most. Even when the spec is good, the people building from it largely don't use it. A developer facing an ambiguous screen will ask the person next to them before she scrolls to section 7.3. Engineers build from the conversation in the room and the ticket in front of them. The document is a reference of last resort, consulted mostly to settle arguments about what was agreed, which means it functions less as a blueprint and more as a contract you reach for when something has already gone wrong.

You can see the same waste from the customer's side. Pendo studied how people actually use the software they pay for and found that 80% of features are rarely or never used, while around 12% drive most of the usage (Pendo 2019 Feature Adoption Report). Read that against a thousand-page spec and the maths is grim. You paid to specify everything. You paid to build most of it. And most of what you built, nobody touches. The spec's great promise was completeness. Completeness turns out to be the expensive way to deliver a pile of features your customers will never open.



None of this means planning is the enemy. You should think hard about what you're building and why. The failure isn't thought. It's the format: a single monolithic document, decided once, defended forever, that mistakes the volume of detail for the reduction of risk. The risk it was meant to kill, the risk of building the wrong thing, is exactly the risk it makes worse, because it locks the wrong thing in before you could possibly know it was wrong.

Smaller is the whole game

So if the monolith is the problem, what's the fix? The data points in one direction, and it's not subtle. The Standish Group's CHAOS research, looking across thousands of projects, found that small projects succeed 61% of the time. Grand projects, the big-bang efforts the thousand-page spec is built to govern, succeed just 6% of the time (Standish CHAOS 2015).

Sit with that gap. It isn't 61 versus 50. It's 61 versus six. The single strongest predictor of whether a software project lands isn't the talent on it, the methodology, or the budget. It's size. The smaller you cut the work, the more often it actually ships.

PROJECT SIZE	SUCCESSFUL	CHALLENGED	FAILED
Small	61%	32%	7%
Grand	6%	51%	43%

(Standish CHAOS 2015. "Challenged" means delivered late, over budget, or short of scope.)

That table is the argument of this entire part of the book, in two rows. The thousand-page spec is a machine for producing the bottom row. It takes work that could have been cut into small, shippable pieces and welds it into one grand project that the numbers say has a 6% chance of going well and a 43% chance of outright failure. You don't fix that by writing a better spec. You fix it by refusing to build a grand project at all.

The alternative isn't planning less. It's planning in a different shape. Instead of one document that decides everything before anything ships, you break the work into small units you can scope, build, and verify one at a time. Each piece is small enough to succeed at the 61% end of the table. Each one ships, gets used, and teaches you something real before you commit to the next. Learning stops being rework. It becomes the input to what you do next.

That unit has a name, and the whole operating system is built around it.

Next: the task. What a unit of work looks like when it's small enough to actually finish.

Task-Shaped Work

If the thousand-page spec is where projects go to die, the task is where work actually gets done. Small, scoped, verifiable, and shipped on its own: the unit you can hold someone to.

The unit is the task

A spec describes a destination. A task is a single step you can take, watch land, and check off before you take the next one. That difference is the whole of task-based development. You stop trying to design the entire system on paper and start moving it forward one finished piece at a time.

Call that piece the task. It is the unit of delivery: small enough to ship in days, scoped tightly enough that "done" is a fact rather than an opinion, and self-contained enough that finishing it doesn't depend on six other things landing first. A good task reads like a promise someone can keep this week. A bad one reads like a chapter of the spec graveyard, broken off and renamed.

The reason to work this way is not neatness. It is the odds. The Standish Group's CHAOS data found small projects succeed 61% of the time while "grand" projects succeed 6% (Standish CHAOS 2015). Nothing else explains the gap. The same firms staffed both kinds of project with the same engineers on the same calendars, and the only variable that moved was how much they tried to do at once. Task-shaped work is what taking that finding seriously looks like in practice. You decompose the grand project into a stream of small ones, each of which you can actually win.

What makes a task a task

Three things turn a vague request into something an engineer can ship and you can verify. Miss any one and you're back to managing a feeling instead of checking a fact.

It's scoped. A task names exactly what's in and, just as usefully, what's out. "Improve the checkout" is not a task; it's a wish. "Let a customer apply a discount code at checkout, one code per order, percentage codes only" is a task. The boundary is the point. A scoped task can be estimated, started, and finished without quietly swallowing three other pieces of work on its way.

It has a definition of done. This is the part founders underrate and the part that does the heavy lifting. The definition of done is the plain-English condition that makes "finished" checkable by someone who can't read the code. Not "the discount feature is done" but "a customer can enter a valid percentage code, the order total drops by that percentage, an invalid code shows a clear error, and no code can be used twice." You don't need to know how it was built. You need to be able to look at the running thing and say yes or no.

It ships on its own. A task that can only go out as part of a bundle isn't really a unit; it's a fragment pretending to be one. The discipline is to scope each task so it can reach production by itself, behind a flag if it has to be, without waiting on the rest of a release. That independence is what gives you the steady drumbeat instead of the quarterly big bang. It's also what keeps the failure small. When something does go wrong, it went wrong inside one task you can see, not inside a nine-month release nobody can untangle.

> *A task you can't verify isn't small. It's just a spec that hasn't admitted it yet.*

The sample task ticket

Here is the artifact the whole method runs on. It's deliberately plain. A founder should be able to read one in under a minute and know what's being built, how they'll check it, and roughly when it lands. Reuse this format; it shows up again in the cadence and on the board.

The Shape of a Task

One card a founder can read in a minute: what gets built, and how you'll know it's done.

TASK

Apply a discount code at checkout

CONTEXT

Sales wants a launch promotion next month, but no code can be honoured today.

SCOPE

In: one percentage code per order. Out: stacking, fixed-amount codes, per-customer limits.

DEFINITION OF DONE

- ✓ A valid code drops the order total by the stated percentage.
- ✓ An invalid or expired code shows a clear inline error.
- ✓ A code cannot be reused once an order is placed.

ACCEPTANCE CHECK

Reviewer orders with a valid code, an expired code, then the same code twice.

Owner: one named engineer **Estimate: ~3 days**

FIELD	ENTRY
Task	Apply a discount code at checkout
Why	Sales wants to run a launch promotion next month; today there's no way to honour a code.
Scope (in)	One code per order; percentage-off codes only; codes set by an admin in the existing settings page.
Scope (out)	Stacking multiple codes; fixed-amount codes; per-customer usage limits. (Separate tasks if needed.)
Definition of done	A customer enters a valid code and the order total drops by the stated percentage; an invalid or expired code shows a clear inline error; a code can't be reused once an order is placed.
Acceptance check	Reviewer places a test order with a valid code (total drops), an expired code (error shown), and the same code twice (second use rejected).
Owner	One engineer, named.
Estimate	~3 days.

Read down that ticket and notice what isn't there. No architecture. No database design. No instruction on how to build it. Those are the engineer's job, and putting them in the ticket is how you slide back toward the spec graveyard. What the ticket pins down is the contract: what "done" means, and how you'll know it's true. The "Why" line earns its place too. It's the thread back to a business goal, so the task doesn't become motion for its own sake.

The acceptance check is the line that protects you. Without it, "done" is whatever the person who built it says it is. With it, anyone can sit down, follow three steps, and reach the same verdict. That's the difference between reliable execution and taking someone's word for it.

Why small wins

There's an instinct, when delivery has been shaky, to scope bigger. Plan harder, specify more, get it right this time. The CHAOS numbers say that instinct is backwards. Size is the thing most correlated with failure, so the move is to make the bets smaller, not the plan thicker.

The fat tail makes the case sharper. Across large IT projects the average cost overrun is 27%, but roughly one in six becomes a "black swan" that overruns its budget by 200% and its schedule by some 70% (Flyvbjerg & Budzier). You cannot have a 200% overrun on a three-day task. There isn't room for the catastrophe to grow. By keeping the unit small you don't just improve your average outcome; you cut off the disaster that ends companies. Each task is a bet sized so that losing it costs you days, not a quarter and the runway with it.

Small units also change what you can learn. A task ships, a customer touches it, and what you find out shapes the next task instead of being buried in a spec written before anyone knew anything. The discount-code work goes live, sales discover they actually need fixed-amount codes more than percentages, and that becomes the next ticket. You couldn't have known that up front. Task-shaped work is built to absorb that the world keeps changing while you build, which is the exact thing the thousand-page spec couldn't survive.

One task, scoped and shipped, is the easy part. A stream of them, week after week, needs a rhythm, and that's what running the cadence is for.

Running the Cadence

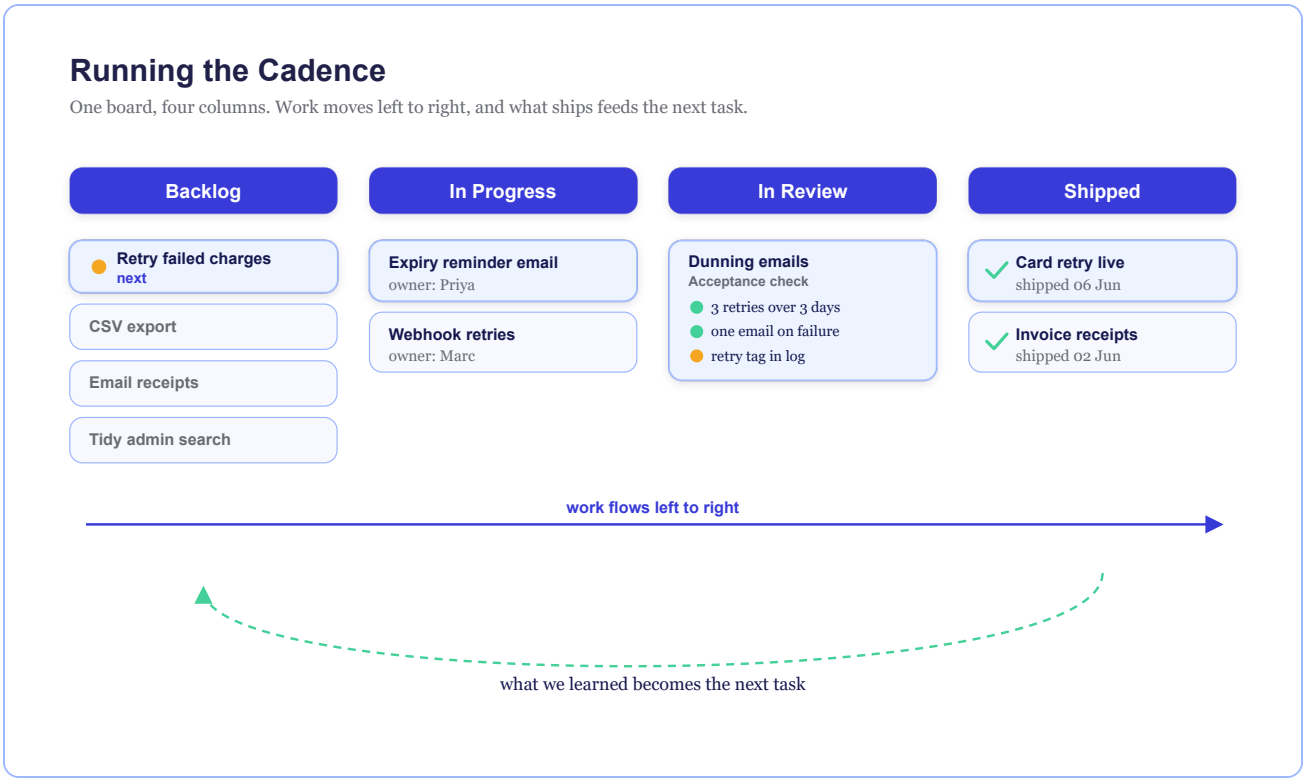
A single task is a one-off. A stream of tasks shipped week after week is a delivery engine, and an engine needs a rhythm. This chapter shows the rhythm.

You have the task. It is small, scoped, and verifiable, with a definition of done you can actually check. One of those is a nice afternoon. The accountability the business is asking for is not a nice afternoon. It is a steady drumbeat of shipped work, the kind that lets you tell a customer "yes, that lands next week" and be right. That drumbeat has a name in this book: reliable execution. This chapter is about how you run it.

The mechanism is a cadence. A cadence is just a repeating cycle with fixed shapes: what gets picked up, how it moves, who checks it, and what happens to the result. Once the shape is fixed, the work flows through it without anyone reinventing the process each week. That predictability is the whole point. The Standish CHAOS data found that small, scoped projects succeed 61% of the time against 6% for "grand" ones (Standish CHAOS 2015). A cadence is how you keep your work small on purpose, forever, instead of letting it swell back into a grand project by accident.

The board

Everything visible, in one place. The board is where the cadence lives, and it has four columns.



COLUMN	WHAT SITS HERE	WHAT IT MEANS
Backlog	Candidate tasks, ordered top to bottom	The top of this list is the most valuable thing not yet started
In Progress	The tasks actively being built right now	Deliberately short. A long In Progress column is a queue pretending to be work
In Review	Built, waiting on acceptance against its definition of done	The gate. Nothing crosses it on trust alone
Shipped	Accepted and in production this cycle	The drumbeat, made countable

A founder can read this board in ten seconds without knowing a line of code. You see what is moving, what is stuck, and what landed. You are not running the work. You are watching it run, which is exactly the position you want to be in.

Prioritisation: the top of the backlog earns its place

The backlog is ordered, not piled. Ordering is the part founders should care about most, because it is where engineering meets the business, and it is the one decision a board makes for you every single week: the task at the top is the next thing that gets built.

So what climbs to the top? The work tied to a business goal you can name. Not "refactor the billing module" but "stop the failed-payment emails that are churning customers this month." Every task on the board should trace to an outcome a commercial person recognises. If it cannot, it is either disguised maintenance, which is fine but should be labelled honestly, or it is someone's pet idea, which is the seed of the feature nobody asked for. Pendo found that 80% of features are rarely or never used (Pendo 2019). A great deal of that waste is decided at the moment of prioritisation, by what you let climb the list.

Reordering is cheap, and that is the quiet advantage of running task-shaped work. You can change your mind every cycle. The market moves, a customer escalates, a competitor ships something, and the response is to drag a new task to the top of the backlog, not to renegotiate a thousand-page contract. The spec graveyard is full of plans that were correct on the day they were signed and wrong by the time anyone built them. A reordered backlog is a plan that gets to stay correct.

Here is a slice of one, with the trace written in:

#	TASK	BUSINESS GOAL IT SERVES
1	Retry failed card charges automatically for 3 days	Cut involuntary churn (currently ~$4,000/mo lost)
2	Add CSV export to the reporting screen	Unblock two enterprise deals stalled on "we need our data out"
3	Email receipts on every successful payment	Reduce "where's my invoice" support load
4	Tidy the internal admin search	Maintenance — speeds up the support team, no revenue tie

You do not need to understand how a card retry works. You need to agree that cutting a $4,000-a-month leak beats tidying an admin screen this week. That judgement is yours, and the board

is built so you can make it.

Review and acceptance: the gate that protects the drumbeat

In Review is the most important column on the board and the one most teams quietly skip. A task is not done because an engineer says it is done. It is done when it has been checked against the definition of done written when the task was scoped, the one with the acceptance check at the bottom. That check is the contract. It is why the work was specified in plain terms before anyone started, so that "done" is a fact you can verify rather than an opinion you have to trust.

Acceptance is also where the task-by-task approval gate sits. One task is reviewed and accepted, and only then does the next begin. That sounds slow. It is the opposite. It means a problem surfaces while it is one task wide, not after it has been built on top of for a month. The cost of catching a defect in review is small. The cost of catching it three tasks later, woven into everything since, is the rework that quietly eats a quarter.

> *A definition of done you check in review is the cheapest insurance you will ever buy. It costs one conversation now and saves a rebuild later.*

A worked acceptance, kept to the same ticket format used throughout the book:

```
TASK #1 — Retry failed card charges automatically for 3 days

Definition of done:
  - A declined charge is retried once a day for 3 days, then marked failed
  - The customer is emailed on final failure, not on each retry
  - Successful retries appear in the payments log with a "retry" tag

Acceptance check (run in review):
  - Force a test card to decline → confirm 3 retries over 3 days, then "failed"
  - Confirm exactly one customer email, sent on final failure
  - Confirm the payments log shows the retry tag

Status: ACCEPTED — checks passed, shipped to production 06 June.
```

Notice what acceptance is not. It is not a code review, which is an engineering concern that happens earlier and inside the team. It is the business-readable confirmation that the thing you asked for is the thing that shipped. You can read every line of that acceptance check without writing software. That is by design, and it is the difference between advice by the hour and hands on keyboard you can verify.

The feedback loop: shipped work writes the next task

The dashed arrow on the board, the one looping back from Shipped to Backlog, is where reliable execution compounds. Every shipped task teaches you something. The card-retry feature goes live and you learn that most declines are expired cards, not insufficient funds, which means the highest-value next task is a "your card is about to expire" reminder rather than anything you had planned. The work tells you what to build next. The board is built to catch that signal and turn it into the next top-of-backlog task while it is still fresh.

This is what a thousand-page spec cannot do. A spec is a guess made at the moment of least knowledge, the very start, and then defended against everything you learn afterward. A cadence inverts it. You ship the smallest useful thing, watch what happens, and let the result reorder the

backlog. Each loop you know more, and the next task is better aimed than the last. The plan improves as the product ships, instead of rotting while it waits.

The rhythm itself stays boring, and boring is the goal. A typical cycle runs a few days per task. Daily, the board updates so a stuck task is visible the day it sticks, not the week it slips. Each cycle, a task is accepted and shipped, the loop feeds the backlog, and the top of the list gets re-checked against the goals that matter now. Nobody is heroic. Nothing is dramatic. Things just ship, on a beat you can set your calendar by, which is the only kind of delivery a founder can actually plan a business around.

That is the engine. The next question is the one a non-technical founder always asks, and should: how do you trust it is running without standing over it every day?

Next: how you see the cadence working without managing it, and who, exactly, is on the hook for each task.

Visibility & Accountability

A standup tells you what people did yesterday. It doesn't tell you whether the work is shipping, who owns it, or whether you're allowed to stop paying. This chapter is about getting all three without sitting in the meeting.

You don't want to run engineering. You shouldn't have to. The whole reason "CTO" looked like the answer was that the title was supposed to come with accountability baked in: one person who owns whether software ships, so you don't have to watch it. The title trap is that the seat fills and the accountability doesn't. You still can't tell, from where you sit, whether this week produced anything you can use.

So let's separate the two things the title was bundling. There's **ownership** (who is responsible for a given piece of work being done right), and there's **visibility** (how you, the non-technical founder, can see that it's happening). A good operating system gives you both as a side effect of running the work, not as a separate reporting layer you have to chase.

Ownership: one task, one name

Task-based development makes ownership trivial to assign, because the unit is small enough that one person can own it end to end. The task has a name on it: not a committee, not a "team" or a Slack channel where responsibility quietly evaporates between everyone who could have done something about it. One developer takes the task, ships it, and their name stays attached through review and acceptance.

This is the quiet fix for the **bus factor of one**. When work arrives as a thousand-page spec handed to a team, ownership smears across everyone and lands on no one, and the only person who really understands the system is the one you can't afford to lose. When work arrives as a stream of scoped tasks, each one is a small, documented, finished thing. The knowledge lives in the tickets and the shipped work, not in one person's head. You can read who owned what, going back months, on a single board.

Here's the ownership map for a typical week, the kind you can read in ten seconds without knowing what any of the code does:

TASK	OWNER	STATE	ACCEPTANCE
Add CSV export to the orders report	Priya	In review	Awaiting your approval
Fix duplicate-charge edge case at checkout	Marco	Shipped	Approved Tue
Wire up the new pricing tier to billing	Priya	In progress	—
Email receipts: include VAT line	Sam	Backlog	Not started

Four rows, each with a name against it and a state you can act on. You're not reading code here. You're reading whether the thing you asked for is moving, and who to ask if it isn't.

Visibility: progress you can read, not a meeting you attend

The standup exists because nobody trusts the board. People gather to say out loud what should already be visible, and a non-technical founder sits there nodding at words they can't evaluate, learning nothing they could act on. That's not accountability. That's theatre with a calendar invite.

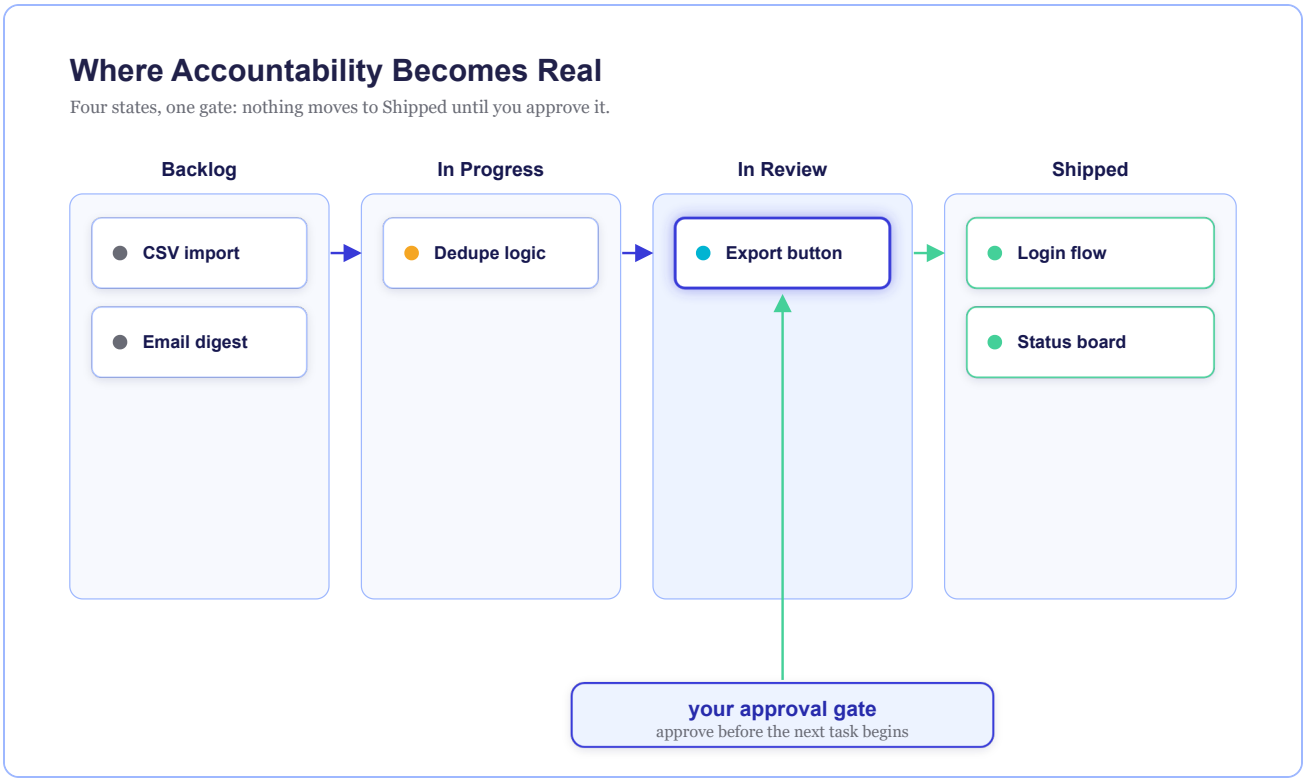
Real visibility is a board you can open on your own time and a definition of done you didn't have to write in code. The board moves left to right (backlog, in progress, in review, shipped), and a card only crosses into "shipped" when the work meets an acceptance check written in plain English. You read the check, not the diff.

> *The test of visibility isn't whether you can attend the standup. It's whether you'd learn anything by skipping it. With task-shaped work and a written definition of done, you wouldn't.*

A sample task ticket is what makes this work for a non-technical reader. Notice that everything load-bearing is in language you can evaluate:

```
TASK-218 — CSV export for the orders report
Owner:    Priya
Context:  Finance is copying orders out by hand every Friday.
          They want a one-click export.
Scope:    Add an "Export CSV" button to the existing orders
          report. Columns: order ID, date, customer, total, status.
          This task does NOT touch the report's filters or layout.
Done when:
  - A button on the orders report downloads a CSV.
  - The file opens cleanly in Excel and Google Sheets.
  - Columns match the five above, in that order.
  - Totals in the CSV match the totals shown on screen.
Acceptance: You (or Finance) export one real Friday's orders
          and confirm the numbers match.
```

You can check every line of that. You don't need to know how the button was built. You need to know that Finance stopped copying orders by hand on Friday, and the "Acceptance" line tells you exactly how to confirm it. That is the difference between **advice by the hour** and **hands on keyboard**: one gives you a recommendation about exports, the other gives you a working button and a way to prove it works.



The approval gate is where accountability becomes real

A board you can read is visibility. Accountability needs one more thing: a point where *you* decide whether the work was good enough, and your decision has teeth.

This is the mechanism in Dev on Demand. Work runs one task at a time, and you approve each completed task before the next one begins. Nothing rolls forward on momentum. If TASK-218 didn't actually let Finance export their Friday orders, it doesn't get marked done because the calendar says it's Friday. It gets marked done when you say the acceptance check passed. The model is built around it, in its own words: *one task, one developer, done right, no surprises.*

That gate does something a standup never can. It moves the burden of proof onto the work. In a status meeting, the question is "what did you do?" and the answer is a story. At an approval gate, the question is "does this pass the check we agreed on?" and the answer is yes or no, demonstrated. You're not evaluating effort or eloquence. You're evaluating a finished thing against a definition of done you both signed off before the work started.

It also means you're never holding a quarter of unverified work. The cost of a task that missed the point is one task, caught at the gate, not three months of build discovered at a demo. Small units, checked one at a time, is the same principle that makes small projects succeed far more often than big ones (Standish CHAOS 2015) applied to how you *govern* the work, not just how it's sized.

How this becomes the scorecard

Everything in this chapter is the raw material for the KPIs coming in the next chapter. You don't have to do the counting by hand. The board already records it.

Watch what each artifact quietly measures:

WHAT YOU SEE ON THE BOARD	WHAT IT BECOMES ON THE SCORECARD
Time a task spent from "in progress" to "shipped"	Cycle time per task
Tasks that came back at the approval gate	Defect escape rate, rework ratio
Whether one name owns most of the board	Bus-factor / key-person concentration
The "Context" line tying each task to a business reason	% of work tied to a business outcome
Tasks you approved vs. tasks you planned	Delivery predictability

A founder can read all ten of those numbers. The harder question, the one the title was supposed to answer and rarely did, is who is accountable for *moving* them in the right direction. Hold that thought.

Seeing progress becomes real when it's ten numbers on one page. So here's the page.

The Scorecard: The Top 10 KPIs

Ten numbers on one page. Each one tells you whether one of the five accountabilities is being delivered, or quietly failing while everyone looks busy.

You've read the five accountabilities and the way the work runs. Now you need the part a founder can actually use: a single page that tells you, at a glance, whether your engineering spend is producing what you paid for.

This isn't a wall of charts, and it isn't a tool only an engineer can read. It's ten numbers, grouped the way the book is grouped, two per accountability. Read top to bottom and you've taken the temperature of the whole operation in about ninety seconds.

The point of the scorecard isn't to turn you into an engineering manager. It's the opposite. You shouldn't have to sit in a standup to know whether things are on track. The numbers do that for you, and they do it in business language: how fast, how predictable, how often it breaks, how much of the work mattered.

A note before the metrics. Four of these come straight from DORA, the industry-standard framework for measuring software delivery (deployment frequency, lead time for changes, change-failure rate, and time to restore). The other six are the operational numbers a non-technical owner needs to see the rest of the picture. Where a benchmark exists I'll name it. Where one doesn't, I'll tell you what "good" looks like in plain terms rather than invent a figure.

Software delivery: is the drumbeat steady?

This is Chapter 3, made measurable. Two numbers tell you whether the right things are shipping on a cadence you can plan around.

1. Lead time for changes. The clock from "we decided to do this" to "it's live for customers." Short lead time means a request becomes working software in days, not quarters. Long lead time is the stop-start pattern you felt in Chapter 3, where a decision sits for weeks before anyone touches it. This is one of the four DORA metrics, so you can compare yourself against a recognised standard rather than your own gut.

2. Delivery predictability. Of the work you committed to this cycle, how much actually shipped? Plan ten tasks, deliver eight, and your predictability is 80%. The absolute speed

matters less than whether the number you're told is the number you get. A team that reliably delivers eight of ten is more useful to a business than one that promises twelve and lands five. Predictability is what lets you make commitments to your own customers without crossing your fingers.

Team performance: does the work flow, or does it depend on one hero?

Chapter 4, on the dashboard. Throughput you can count, and the key-person risk the title was supposed to remove.

3. Cycle time per task. Lead time measures the whole journey; cycle time measures the active part, from the moment someone starts a task to the moment it's done. It's your read on flow. When cycle time creeps up, work is queuing somewhere, or tasks are too big to move cleanly. Small, scoped tasks keep this number low and stable, which is exactly why the work is shaped the way Chapter 9 describes.

4. Bus-factor / key-person concentration. How many people could keep a given system running if one person were out for a fortnight? A bus factor of one is the quietest risk in your company. Everything works right up until the day it doesn't, and then nobody else can fix it. You measure this by asking, per critical area, how many engineers have shipped to it recently. One name everywhere is a flashing light, however fast that person is.

Operational reliability: when it breaks, how bad and how long?

Chapter 5. Nothing stays up forever. These two tell you whether failure is rare and recovery is fast.

5. Change-failure rate. Of everything you ship, what fraction breaks something and needs a fix, a rollback, or a patch? A DORA metric, and a blunt honesty check. A team shipping fast with a high failure rate isn't fast, it's accruing a bill. A low rate means the speed is real and customers aren't paying for it in outages.

6. Time to restore (MTTR). When something does break, how long until service is back? The fourth DORA metric, and the one your customers feel most directly. A short restore time means

a bad deploy is a blip; a long one means a bad deploy is a bad week. Notice that reliability isn't about never failing. It's about failing rarely and recovering quickly, which is something you can actually measure and hold someone to.

Technical quality: is the codebase getting cheaper or more expensive to change?

Chapter 6. Quality is invisible until it isn't. These surface the rot before it becomes a system everyone's afraid to touch.

7. Defect escape rate. Of the bugs that exist, how many were caught before customers hit them versus after? A high escape rate means your customers are doing your testing for you, which is the most expensive way to find a defect and the worst for trust. Drive this down and reliability and reputation move with it.

8. Rework / tech-debt ratio. What share of effort goes to redoing or repairing past work rather than building new things? Some rework is normal. A ratio that climbs quarter over quarter is the sound of a codebase rotting, the early warning before change gets slow and brittle. This is the number that explains why a team that shipped fast last year is crawling this year, and it's the one founders almost never get shown.

Roadmap & business alignment: did the work matter?

Chapter 7. The two most commercial numbers on the page, and the ones closest to your job.

9. Percent of delivered work tied to a business outcome. For each shipped task, can someone name the goal it serves: a revenue lever, a retention risk, a cost saved, a customer commitment? Work that can't be traced to an outcome is work you paid for on faith. Pendo found that 80% of features are rarely or never used (Pendo 2019). This is the number that keeps your team off that list.

10. Roadmap commitment accuracy. When you tell the board, or a customer, that something lands by a date, how often does it? Predictability (number two) is about the cycle in front of you; this is about the promises you make further out. It's the difference between a roadmap you can stake your reputation on and a wish list that quietly slips.

The dashboard

Here's the whole thing on one page. Read it the way you'd read a P&L: not every number every day, but the shape, and which line is moving the wrong way.

#	ACCOUNTABILITY	KPI	WHAT IT ANSWERS	"GOOD" LOOKS LIKE
1	Software delivery	Lead time for changes	How fast a decision becomes live software	Days, not weeks; trending down or flat
2	Software delivery	Delivery predictability	Did we ship what we committed to?	Committed ≈ delivered, cycle after cycle
3	Team performance	Cycle time per task	How cleanly work flows once started	Low and stable; no creeping queues
4	Team performance	Bus-factor concentration	Could we survive losing one person?	More than one name on every critical area
5	Operational reliability	Change-failure rate	How often a release breaks something	Low and falling
6	Operational reliability	Time to restore (MTTR)	How fast we recover when it breaks	Short; an outage is a blip, not a week
7	Technical quality	Defect escape rate	Are customers finding our bugs?	Most defects caught before release
8	Technical quality	Rework / tech-debt ratio	Is the codebase getting costlier to change?	Steady, not climbing
9	Roadmap & alignment	% work tied to an outcome	Did the work matter commercially?	Every task traces to a business goal
10	Roadmap & alignment	Roadmap commitment accuracy	Do our dated promises land?	Dates hit; slippage is rare and flagged early

Four of these (1, 5, 6, and lead time's close cousin) are DORA metrics, which means you're not grading on a curve you invented. The rest are the operational reality a founder needs and rarely gets handed.

You don't need to act on all ten constantly. Glance at the page weekly, watch for a line drifting red, and ask about that one. That's the whole discipline. The scorecard turns "I think engineering is going okay" into something you can defend to a board.

There's a catch, and it's the bridge to the rest of this part of the book. A founder can read all ten of these. That was always the easy half. The hard half is having someone whose job is to *move* them, week after week, and to answer for the line that's gone the wrong way.

You can read the scorecard. The real question is who owns it: who do you build, hire, or subscribe to make these ten numbers their problem?

Build, Hire, or Subscribe

Three doors lead out of the engineering gap. Only one of them puts shipped software in production next month, and the cost maths tells you why.

You've read twelve chapters arguing that what you're missing isn't a title, it's reliable execution: the five accountabilities, delivered week after week, measured by the ten KPIs. Fine. Now you have to actually do something. There are three doors, and you'll walk through one of them whether you choose deliberately or by default.

Door one: build an in-house team. Door two: hire a CTO, full-time or fractional. Door three: subscribe to the execution and skip the org-chart question entirely. Each one buys you something different, and each one costs differently. Let's price them honestly.

Door one — build the team

This is the instinct most founders trust, because it feels like control. You hire engineers, they sit on your payroll, the code is yours, the people are yours. The trouble starts with the bill, and it's bigger than the salary line you're picturing.

The average US software developer earns about $154,076 a year (US Census/BLS via DataUSA, 2024). That's the wage. It is not the cost. Benefits run to roughly 29.9% of total compensation for private-industry workers (BLS ECEC), which puts the fully-loaded figure at about 1.43 times base. Do that multiplication and one developer costs you in the region of $220,000 a year, or about $18,000 a month. One developer.

A single developer is not a team and it isn't an accountability owner either. You need at least two or three to cover delivery, plus someone to own quality and reliability, plus someone deciding what gets built. Stack three or four loaded engineers and you're past $700,000 a year before anyone has reviewed a line of code. And you carry that cost from the day each contract starts, not from the day the first feature ships.

Then there's the wait. Hiring is slow and getting slower: across all roles, time-to-fill rose from 43.6 days in 2022 to 59.7 days in 2025, a 37% jump (Greenhouse). That figure covers every kind of job, not engineering specifically, so read it as direction rather than a precise number. Senior engineers are not faster to land than the average. Two months to fill one seat, and you still have

to onboard, integrate, and hope the hire was a good one. If it wasn't, you eat the first-year salary plus the ramp time plus the cost of starting over.

Build is the right door when engineering *is* the business and you intend to run it at scale for years. For a growing company that mostly needs things to ship, it's a heavy, slow, expensive way to find out whether your hires can execute.

Door two — hire a CTO

So you skip the team and reach for a leader. Get the right person at the top, the thinking goes, and the team and the shipping follow.

A full-time technology leader is a serious line item. The official proxy, a US Computer and Information Systems Manager, averages about $155,405 a year (US Census/BLS via DataUSA, 2024), and the same 1.43 loading applies on top. At the senior end the market runs far higher. That buys you strategy, hiring, architecture decisions, a single accountable name. What it does not buy you, on its own, is a working team underneath that name. A CTO with no engineers to direct is an expensive way to produce slide decks.

The fractional version answers the cost objection. A **fractional CTO** gives you a fraction of a senior leader's time for a fraction of the price, and one vendor estimate puts it at 40 to 60% less than a full-time hire (vendor figure, treat as a range). The model is real and it has real uses. Read what fractional vendors say it is, though, in their own words: a fractional CTO "leans more heavily toward strategic guidance, oversight, and experience while eschewing the leadership role of managing teams" (gofractional). That sentence is the whole problem in one line.

Strategic guidance is advice by the hour. It is not hands on keyboard. A fractional leader can tell you the roadmap is wrong, the architecture won't scale, the hire was a mistake. All useful. None of it ships software, because the person giving the advice has explicitly stepped back from running the people who would. You've bought a diagnosis and left the treatment unfunded. That distance between sound advice and working code in production is the execution gap, and a part-time advisor sits on the wrong side of it by design.

Hire is the right door when you have a team that ships but lacks direction, or when the strategic call genuinely is the bottleneck. If nothing is reaching production, a CTO of either kind is a title filling a seat. You went shopping for an org-chart fix and you'll come home without any shipped software.

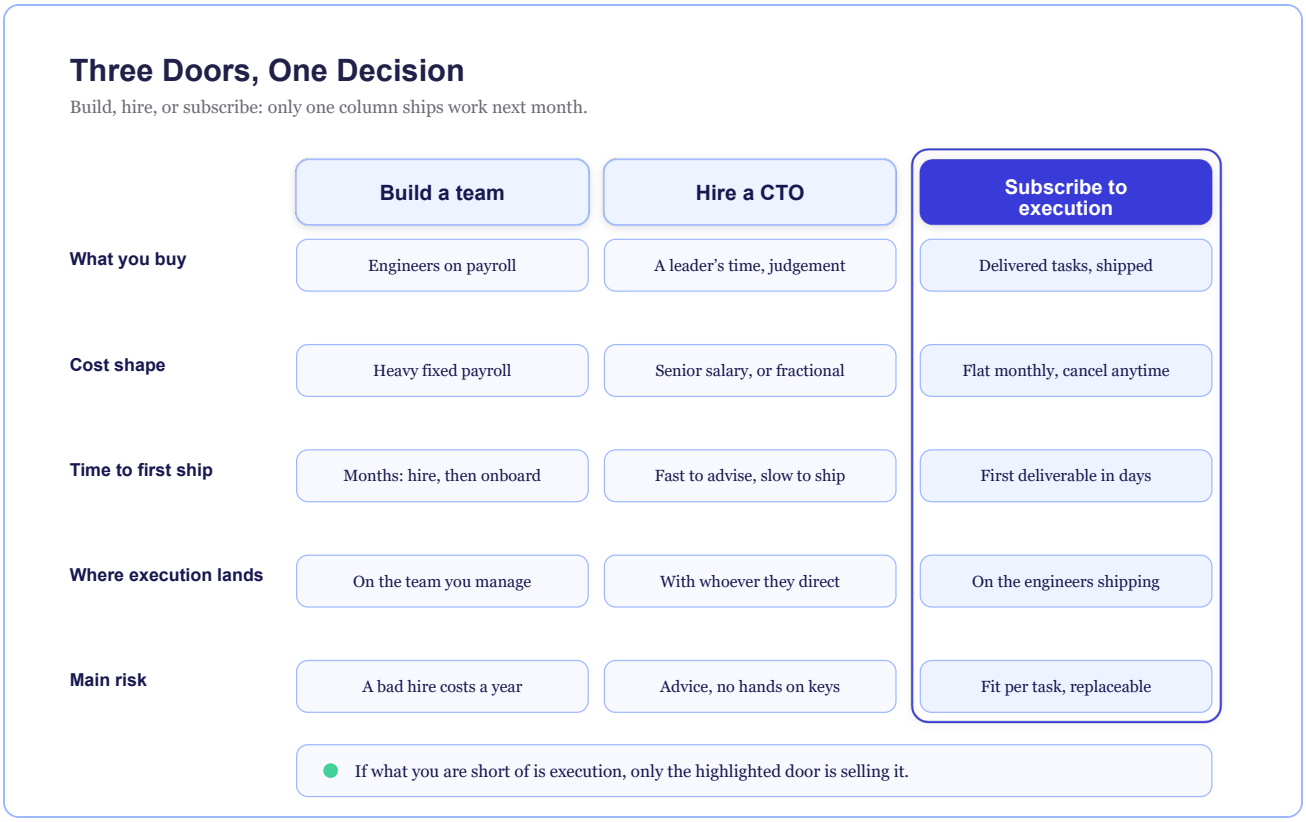
Door three — subscribe to the execution

The third door asks a different question. Not "who do I put in the seat," but "how do I get the five accountabilities delivered, on a cadence, without owning the headcount or the hiring risk."

That's subscription engineering. You pay a flat monthly fee for engineers who deliver task-based work, you approve each task before the next one starts, and you can stop when you no longer need it. There's no two-month hire to wait out and no severance to plan. The execution is the product, not a by-product you hope your hires produce.

This is Dev on Demand, and the chapters ahead spell out exactly what it covers. Here it earns one row in a table. The point of the comparison isn't to crown a winner for every company. It's to show you what each door actually delivers against the thing you came for, which is reliable execution.

The three doors, side by side



	BUILD A TEAM	HIRE A CTO	SUBSCRIBE TO EXECUTION
What you buy	Engineers on payroll	A leader's time and judgement	Delivered tasks, shipped
Indicative cost	~$18,000/mo per loaded developer; a real team runs well past $700k/yr	Full-time leader at loaded cost; fractional ~40–60% less (vendor estimate)	Flat monthly subscription, cancel anytime
Time to first shipped work	Two months to hire, then onboarding	Fast to advise; team and shipping still to be built	First deliverable within days
Where execution lands	On the team you assembled and manage	With whoever the leader directs (fractional steps back from managing teams)	On the engineers delivering each task
Who owns accountability	You, across every seat	One name, advisory if fractional	Task-by-task approval gate
Main risk	Slow, expensive, a bad hire costs a year	Advice without hands on keyboard	Fit per task, with a replacement guarantee

Read the table the way you'd read a board pack. With Build, you get people, along with the long and costly job of managing them. With Hire, you get a name at the top, and if it's fractional, advice that stops short of running anyone. Subscribe is the column that hands you the shipped work itself. So if the thing you're short of is execution, only one of these three is actually selling it.

None of this is a case against ever building a team or ever hiring a leader. Both are right for some companies at some stage. The question this book has pushed you toward is narrower and more useful than "which is best." It's this: which door puts working software in front of your customers, predictably, starting next month? Price the doors against that, and the choice gets a lot clearer.

Next: pulling the whole spine together. The title you never needed, and the execution you did.

Conclusion: Reliable Execution, Delivered

You never needed the title. You needed the five accountabilities delivered, week after week, and a way to know they were. Here's the whole argument in one breath, and the one small step that proves it.

Go back to the founder from the first chapter. Revenue climbing, backlog a swamp, contractors vanishing for days, and the instinct to fix it by filling a box on the org chart. That founder didn't need a fractional CTO. They needed work to ship. The seat was never the problem. The problem was that nothing was crossing the distance between a good idea in a meeting and a working feature in front of customers.

That distance is the execution gap, and it's where this whole book has been pointing. A title doesn't cross it. Neither does a retainer for two mornings a week, or advice by the hour, however sharp. All of that stays on the wrong side of the gap. The only thing that crosses it is execution: someone owning a piece of work, shipping it, then owning the next.

The spine, in one breath

Strip the book to its frame and it's four moves.

You don't hire a CTO. You hire five accountabilities wearing one job title: software delivery, team performance, operational reliability, technical quality, and roadmap and business alignment. Every one of them is an outcome you can watch land or watch slip. None of them is a personality you interview for.

Those five get delivered by how the work is run, not by who sits at the top of the chart. The thousand-page specification is the spec graveyard, obsolete before it ships and read by no one. The thing that actually works is task-based development: small, scoped, verifiable work, one task at a time, each one you can check, each one shippable on its own. The Standish CHAOS data is blunt about why. Small projects succeed 61% of the time; "grand" ones, just 6% (Standish CHAOS 2015). Size is the dominant driver of whether software ships at all. Task-shaped work is how you stay on the right side of that line on purpose.

And you know it's working because you can read it. Ten numbers on one page, two per accountability: lead time and delivery predictability; cycle time and bus factor; change-failure rate and time to restore; defect escape rate and rework ratio; percent of work tied to a business outcome and roadmap commitment accuracy. You don't run the standup. You read the scorecard, the way you read a P&L, and you ask about the line that's gone red.

WHAT YOU THOUGHT YOU NEEDED	WHAT YOU ACTUALLY NEEDED
A CTO on the org chart	Five accountabilities, delivered
A thousand-page spec	Task-shaped work, shipped one task at a time
To trust that engineering is "going okay"	Ten numbers you can read in ninety seconds
A title to hold accountable	Execution you can see, week after week

That's the argument. Define the seat, deliver the seat, measure the seat. Nowhere in those three steps did you have to fill it with a single expensive hire.

Where the execution actually lands

The scorecard chapter ended on a catch. A founder can read all ten numbers. That was always the easy half. The hard half is having someone whose job is to *move* them, and to answer for the one that drifted the wrong way.

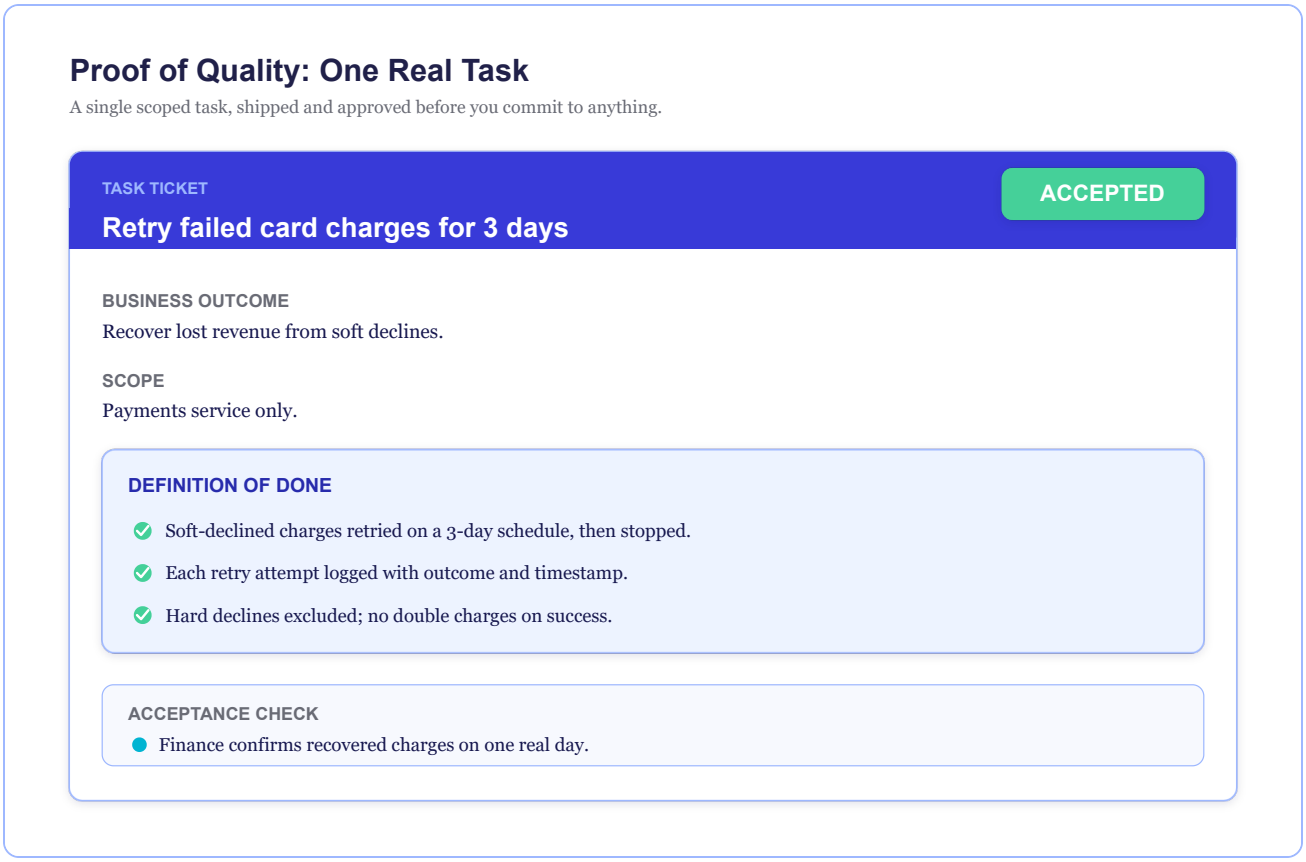
That's the part advice doesn't reach. A roadmap deck doesn't move your lead time. An architecture opinion doesn't lower your change-failure rate. Someone with hands on keyboard, shipping task after task and owning the result, is the only thing that moves a single line on that page. The accountabilities, the operating system, and the scorecard all converge on the same requirement: delivery you can count on and check.

This is **Dev on Demand**. Subscription engineering on a flat monthly fee, cancel anytime, built around the unit this book has called the task. You submit a task, an engineer ships it on roughly a three-day cycle, and you approve it before the next one begins. The approval gate is the accountability the title was supposed to provide and so rarely does. You see the work, you check the work, and only then does the meter move to the next piece. The proof points are on the live page: a 98.4% renewal rate, 60-plus projects delivered, a first deliverable inside five business days, a five-day replacement guarantee, and 100% IP ownership from the start.

The feature list isn't really the point. What matters is the shape of the model. You're not buying a slice of a senior person's week spent on opinions; you're buying a stream of shipped, approved, verifiable tasks. Hands on keyboard, on a cadence, against the ten numbers you already know how to read.

The next step: one real task

You don't have to take any of this on faith, and you shouldn't. The natural next step isn't a contract or a discovery phase. It's the **Proof of Quality**: one real task, scoped and shipped, evaluated by you before you commit to anything.



Pick something that's been stuck. Write it as a task: what it is, the outcome it serves, the scope, the definition of done, the check that tells you it's right. Hand it over. Watch it ship inside a few days. Then judge it the way you'd judge any piece of work you paid for. Did it land on time? Did it do what the ticket said? Could you read what happened without sitting in a single standup?

One task tells you more about whether you'll get reliable execution than a stack of interviews ever will.

That's the whole pitch, and it's a small one on purpose. No transformation, no year-long programme. Just one task, done right, that you approve before anything else begins.

You started this book about to fill a seat. You can close it knowing the seat was never the thing. The five accountabilities are the thing, task-based development delivers them, and the ten KPIs tell you it's true. What's left is to stop shopping for a title and start buying the outcome.

> *Buy the execution, not the title. Outcomes, not org charts.*

Your move: write one stuck piece of work as a task, and let the Proof of Quality show you what reliable execution feels like.

Appendix A — The Artifact Pack

The templates Part III runs on. Copy them, swap in your own fields, and put them to work this week. You don't need software to start: a shared doc and a spreadsheet will do.

These are the three artifacts the book keeps pointing at: the task ticket, the board and its cadence, and the roadmap slice. Each one is plain enough that a non-technical founder can read it and a developer can work from it. That dual readability is the point. The artifact is where the business and the keyboard meet.

Nothing here is proprietary, and nothing here is heavy. If a template makes the work harder to see rather than easier, cut a field.

---

1. The task ticket

The unit of delivery. One task, scoped small enough to ship on its own and check when it's done. A good ticket fits on one screen and answers four questions: what, why, what counts as finished, and how you'll know.

FIELD	WHAT GOES HERE	WHY IT EARNS ITS PLACE
Title	One line, plain English. "Add CSV export to the orders list."	A founder should grasp it without a translation.
Context	Two or three sentences. What's happening, who asked, what business need sits behind it.	Stops the developer guessing and stops scope creep later.
Scope	A short list of what this task *does* cover, and one line on what it explicitly *does not*.	The out-of-scope line is what keeps the task small.
Definition of done	Bullet checklist of the conditions that must all be true to call it finished.	This is the contract. No "done" without every box ticked.
Acceptance check	The single thing you'll do to verify it works. "Export an order list, open the file, confirm the columns."	Lets a non-technical owner accept the work without reading code.
Owner	One name. The person whose keyboard this lands on.	One owner per task. No shared accountability.
Estimate	A rough size, in days, not a precise forecast.	Sets expectation, feeds the cadence, stays honest.

SAMPLE TASK TICKET

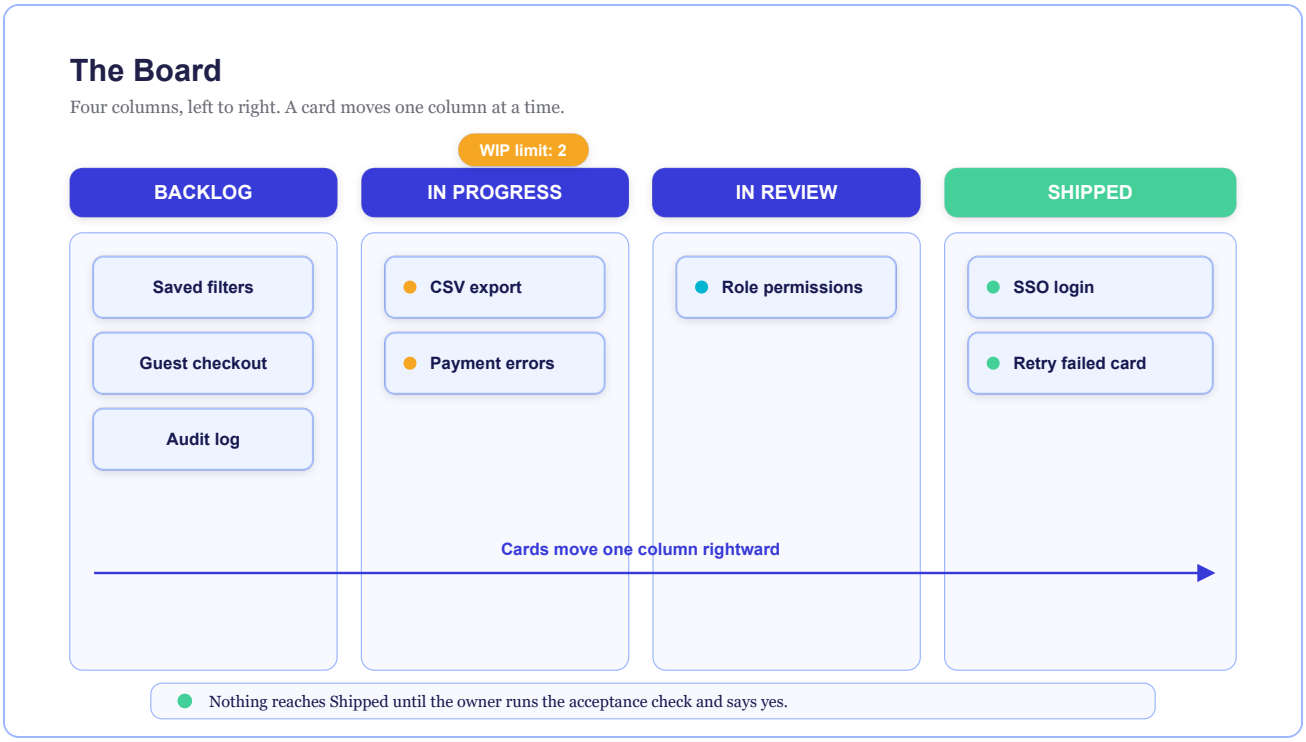
> ***Title:*** *Add CSV export to the orders list* ***Context:*** *Finance currently copies orders into a spreadsheet by hand every Friday. They've asked for a one-click export so the weekly reconciliation stops eating an afternoon. This is the first of three small finance-facing tasks.* ***Scope:*** *- In: an "Export CSV" button on the existing orders list that downloads the currently filtered orders. - Not in: scheduled or emailed exports, any new report formats, changes to the orders list itself.* ***Definition of done:*** *- Button visible on the orders list to users with the finance role. - Export respects the active filters (date range, status). - File opens cleanly in Excel and Google Sheets with correct column headers. - Works on a 10,000-row order list without timing out.* ***Acceptance check:*** *Filter the orders list to last month, click Export, open the file, confirm the row count and totals match what's on screen.* ***Owner:*** *Priya · **Estimate:** 2 days*

Reuse this shape for every task. The fields don't change; only the content does. When a ticket starts needing a fifth bullet under scope or a second owner, that's the signal to split it into two tasks.

---

2. The board and the cadence

The board is where the founder watches delivery without running it. Four columns, left to right, and a task moves one column at a time. You read the board by scanning where the cards sit, not by attending a meeting.



COLUMN	WHAT IT MEANS	WHO ACTS HERE
Backlog	Written, scoped tickets waiting their turn, ordered by priority.	Founder / owner sets the order.
In Progress	Being worked on right now. Keep this column short.	Developer.
In Review	Built, awaiting the acceptance check.	Owner runs the check and approves.
Shipped	Approved and live.	Nobody — it's done.

Two rules keep the board honest. First, a **work-in-progress limit**: cap how many cards sit in In Progress at once (one or two per developer). A long In Progress column means work is being started, not finished. Second, the **approval gate**: nothing moves from In Review to Shipped until the owner runs the acceptance check and says yes. That gate is where accountability lives.

THE WEEKLY CADENCE

The board doesn't run itself. A light rhythm keeps tasks flowing and gives the founder a fixed point each week to see progress and steer.

WHEN	EVENT	LENGTH	WHAT HAPPENS
Start of week	Prioritise	~30 min	Owner orders the backlog. Pull the top tasks into the week.
Daily	Async update	minutes	Each developer posts: shipped yesterday, working on today, anything blocked. Written, not a meeting.
On completion	Approval gate	~15 min	Owner runs the acceptance check, approves or sends it back with a note.
End of week	Review	~30 min	Look at what shipped vs. what was planned. That gap feeds next week's order.

You don't need to attend a standup or read a burndown chart. You read the board, you read the daily updates when you want to, and you run the approval gate. That's the whole job of seeing the work.

---

3. The roadmap slice

A roadmap slice ties a stream of tasks back to a business outcome, so every piece of work has a reason a non-technical reader can defend. This is the artifact that keeps delivery aligned with revenue instead of drifting toward whatever's technically interesting.

It's deliberately not a year-long Gantt chart. A slice covers one outcome and the handful of tasks that move it, with rough timing in weeks. You hold several slices at once, ordered by which outcome matters most.

BUSINESS OUTCOME	WHY IT MATTERS NOW	TASKS (IN ORDER)	ROUGH TIMING	DONE WHEN
Cut the weekly finance reconciliation	Finance loses an afternoon every Friday; it doesn't scale as orders grow.	CSV export · saved filter presets · scheduled Friday email	Weeks 1–3	Reconciliation takes under 30 minutes with no manual copying.
Reduce checkout drop-off	~1 in 5 carts abandons at payment; each point recovered is direct revenue.	Guest checkout · clearer payment errors · retry-failed-card flow	Weeks 3–6	Measured drop-off at the payment step falls and holds.
Onboard enterprise customers faster	Sales has three deals stalled on missing access controls.	Role-based permissions · audit log · SSO login	Weeks 5–9	A new enterprise account can be set up without engineering touching it.

Each row reads as a sentence a founder would say to a board: *we're spending the next three weeks getting finance off manual reconciliation, and we'll know it worked when Friday stops costing an afternoon*. The tasks in each row become tickets in the backlog. The outcome column is what you protect when someone proposes a feature nobody asked for.

Keep the slice shallow and current. When an outcome is met, drop the row and pull the next one up. A roadmap that lives in this shape never becomes the spec graveyard, because it was never trying to predict the whole year in the first place.

Appendix B — The Top 10 KPIs: Definitions & How to Measure

The reference behind Chapter 12. One entry per KPI: what it is, how to measure it, what good looks like, and which of the five accountabilities it serves.

The scorecard in Chapter 12 gives you the ten numbers on one page. This appendix tells you how each one is built, so anyone running your delivery can compute it the same way twice. Four of the ten come straight from the DORA four key metrics (DORA, 2024): lead time for changes, change-failure rate, time to restore, and (by extension) the delivery rhythm those metrics assume. The rest are the measures a non-technical founder can read without learning to run a sprint.

A note on "what good looks like." Where a public benchmark exists and is sourced, the table names it. Where one doesn't, the column gives you a direction of travel rather than a borrowed number, because a made-up target is worse than an honest one. The point of a KPI isn't to hit somebody else's figure. It's to make a hidden thing visible and give one person the job of moving it.

The ten, at a glance

#	KPI	ACCOUNTABILITY IT SERVES
1	Lead time for changes	Software delivery
2	Delivery predictability	Software delivery
3	Cycle time per task	Team performance
4	Bus-factor / key-person concentration	Team performance
5	Change-failure rate	Operational reliability
6	Time to restore (MTTR)	Operational reliability
7	Defect escape rate	Technical quality
8	Rework ratio	Technical quality
9	% of work tied to a business outcome	Roadmap & business alignment
10	Roadmap commitment accuracy	Roadmap & business alignment

1. Lead time for changes

What it is	The clock from "a change is started" to "that change is live in front of users." A DORA key metric (DORA, 2024).
How to measure	Timestamp when work begins on a change and when it reaches production. Take the median over a rolling window, not the average — one slow item shouldn't move the headline.
What good looks like	Short and steady. Trending down over a quarter matters more than any single figure. If lead time is measured in weeks, your delivery has a queue you can't see.
Serves	Software delivery.

2. Delivery predictability

What it is	How closely what you said you'd ship matches what you actually shipped, over a defined window. The honesty check on the cadence.
How to measure	At the start of each cycle, record what was committed. At the end, record what shipped. Predictability = delivered ÷ committed, tracked over time. Count only work that met its definition of done.
What good looks like	High and stable, without padding the commitment to game it. A team that promises little and delivers it isn't predictable. It's quiet.
Serves	Software delivery.

3. Cycle time per task

What it is	How long one task takes from "picked up" to "shipped." Lead time looks at the whole change; this looks at the unit of delivery, the task.
How to measure	Per task, log the time from in-progress to done. Report the median and the spread. A tight spread tells you the work is genuinely task-shaped; a wide one says some "tasks" are really projects in disguise.
What good looks like	Consistent, with few outliers. When most tasks land inside a narrow band, capacity becomes something you can forecast instead of guess.
Serves	Team performance.

4. Bus-factor / key-person concentration

What it is	How much of your delivery depends on a single person. A bus factor of one means one resignation, or one bad week, stops the work.
How to measure	For each critical area, count how many people can ship a change there unaided. Map ownership across the codebase and flag anything with exactly one name on it.
What good looks like	No single owner for anything that matters. At least two people who can keep each critical path moving.
Serves	Team performance.

5. Change-failure rate

What it is	The share of changes that cause a failure in production — a rollback, a hotfix, a degraded service. A DORA key metric (DORA, 2024).
How to measure	Over a window, divide the number of changes that caused a failure by the total number of changes deployed. Define "failure" once, in writing, and apply it the same way every time.
What good looks like	Low and falling. Some failure is normal; a rate that climbs is a quality signal, not bad luck.
Serves	Operational reliability.

6. Time to restore (MTTR)

What it is	When something breaks, how long until service is back. A DORA key metric (DORA, 2024).
How to measure	Timestamp the start of each incident and the moment service is restored. Report the median time to restore across incidents.
What good looks like	Fast and predictable. Founders fixate on never failing; the reliable measure is how quickly you recover when you do.
Serves	Operational reliability.

7. Defect escape rate

What it is	The share of defects that reach customers instead of being caught before release. A bug found in review is cheap. A bug found by a user is not.
How to measure	Over a window, divide defects reported by users by total defects found (users plus those caught internally before release). Track the trend.
What good looks like	Most defects caught before they ship, and the escape rate trending down. A rising number means your safety net has holes.
Serves	Technical quality.

8. Rework ratio

What it is	How much effort goes into redoing or repairing work already "finished," versus building something new. The tax the spec graveyard and accumulated debt charge you.
How to measure	Tag each task as new work or rework (fixing, reworking, or paying down debt on prior work). Track rework as a share of total effort over time.
What good looks like	A modest, stable share. Climbing rework is the codebase telling you it's rotting before anyone says the word "debt."
Serves	Technical quality.

9. % of work tied to a business outcome

What it is	The share of delivered work that traces to a named business goal — revenue, retention, a cost saved, a risk closed. The antidote to building features nobody asked for. Pendo found 80% of features are rarely or never used (Pendo, 2019); this is how you stay out of that 80%.
How to measure	Require every task to name the outcome it serves. At the end of a window, divide work tied to a stated outcome by all work shipped.
What good looks like	High, with each linked outcome real rather than a label bolted on after the fact. Work that traces to nothing is the first thing to question.
Serves	Roadmap & business alignment.

10. Roadmap commitment accuracy

What it is	When you commit a roadmap item to the business, how often it lands roughly when you said. Predictability (KPI 2) measures the cadence; this measures the promises you make outside engineering.
How to measure	Record each roadmap commitment with its target window. When it lands, mark hit or missed against that window. Accuracy = hits ÷ commitments, over time.
What good looks like	High enough that the business can plan around your word. A roadmap nobody trusts is just a wish list with dates.
Serves	Roadmap & business alignment.

The Ten KPIs

One page, five accountabilities, ten numbers, and the single owner who moves each.

KPI	CURRENT	TREND	OWNER
Software Delivery			
Lead time for changes	2.1 days	↘	Tech Lead
Delivery predictability	88%	→	Tech Lead
Team Performance			
Cycle time per task	1.4 days	↘	Eng Manager
Bus-factor	1 area	→	Eng Manager
Operational Reliability			
Change-failure rate	6%	↘	SRE Lead
Time to restore	42 min	↘	SRE Lead
Technical Quality			
Defect escape rate	9%	↘	QA Lead
Rework ratio	18%	→	QA Lead
Roadmap & Alignment			
% work tied to a business outcome	74%	→	Founder
Roadmap commitment accuracy	81%	→	Founder

A last word on using these. Measure a handful well rather than all ten badly, and define each one in writing before you track it, so the number means the same thing next quarter as it does today. Ten numbers on a page are easy to admire. The harder question, the one Chapter 12 leaves you with, is who owns moving them.

Appendix C — Sources & Further Reading

Every number in this book is anchored to something you can check. Here is the list, with the original source for each figure and the basis for any currency conversion.

A note on the numbers before the tables. We lead with the hard anchors: the Standish CHAOS data on project success, Flyvbjerg and Budzier on cost overruns, the BLS loading factor, and the DORA four-metric framework. Where a figure comes from a vendor's own report, we say so and attribute it to the vendor rather than dressing it up as independent fact. Every dollar figure is in US dollars. Where a source published in another currency, the conversion basis is recorded in the right-hand column so you can redo the sum yourself.

Why software projects fail (Parts II–III)

The spine of the task-based argument is here. Small, scoped work succeeds far more often than big-bang projects, and the rare disaster is what bankrupts a budget, not the average.

CLAIM USED IN THE BOOK	SOURCE	WHERE TO FIND IT
Modern projects resolve 29% successful / 52% challenged / 19% failed	Standish Group, CHAOS Report 2015	infotech.com/agile/CHAOSReport2015-Final.pdf
Small projects succeed 61% of the time vs 6% for "grand" projects; size is the dominant driver	Standish Group, CHAOS Report 2015	same as above
Across 1,471 IT projects, average cost overrun 27%	Flyvbjerg & Budzier, "Why Your IT Project May Be Riskier Than You Think," HBR 2011	arxiv.org/pdf/1304.0265 · ssrn.com/abstract=2229735
One in six IT projects is a "black swan": ~200% cost overrun, ~70% schedule overrun	Flyvbjerg & Budzier, HBR 2011	same as above
80% of features are rarely or never used; ~12% drive 80% of usage	Pendo, 2019 Feature Adoption Report	go.pendo.io (2019 Feature Adoption Report)

The Pendo figure is a vendor's own product-analytics report. We quote it as Pendo's and attribute the spec-graveyard point to it directly, not to anyone else.

What the CTO role actually covers (Parts I–II)

The five accountabilities aren't ours alone. They map cleanly onto how authoritative role descriptions define the job, which is why the spine holds. The reading below is where the mapping comes from.

THE BOOK'S ACCOUNTABILITY	HOW PUBLIC DESCRIPTIONS FRAME IT	SOURCE
Roadmap & business alignment	"Aligning technology investment with business strategy"	Splunk; edstellar
Team performance	Structure, capability and culture of the technical organisation	Splunk
Operational reliability	Accountable for system uptime and reliability at scale	edstellar
Technical quality	Target architecture, build-vs-buy, security and compliance	Splunk; edstellar
Software delivery	Under-emphasised in most public CTO descriptions, which skew to vision	observed across the sources below

Further reading on the role:

- Splunk, "The Chief Technology Officer (CTO) Role and Responsibilities" — splunk.com/en_us/blog/learn/chief-technology-officer-role-responsibilities.html
- edstellar, "Chief Technology Officer Roles and Responsibilities" — edstellar.com/blog/chief-technology-officer-roles-and-responsibilities
- Wikipedia, "Chief technology officer" — en.wikipedia.org/wiki/Chief_technology_officer

On the fractional model specifically, one vendor describes it as leaning "toward strategic guidance, oversight, and experience while eschewing the leadership role of managing teams." That is the execution gap stated in a fractional provider's own words. Read it as a vendor's framing, which is how we used it.

- gofractional, "Fractional CTO" — gofractional.com/blog/fractional-cto
- altexsoft, "Fractional CTO" — altexsoft.com/blog/fractional-cto

What good delivery looks like (Part IV)

The KPI scorecard rests on a framework that has held up across a decade of industry research. We use the four metrics as the measurement frame. We do not quote a specific "Nx faster" gap

between elite and low performers, because that multiplier wasn't something we could verify against the report itself.

USED AS	SOURCE	WHERE TO FIND IT
The four key metrics: deployment frequency, lead time for changes, change-failure rate, time to restore	DORA, 2024 State of DevOps	dora.dev/guides/dora-metrics-four-keys/
Conceptual anchor for the four metrics	Forsgren, Humble & Kim, *Accelerate*	IT Revolution Press, 2018

The cost maths (Chapter 13)

The build-versus-subscribe sum is built from two public inputs: a base wage and a loading factor for benefits and overhead. Both are sourced. The base wage figures come from DataUSA, which republishes US Census ACS and BLS data with citation, so treat them as official-derived. The loading factor comes straight from the Bureau of Labor Statistics.

INPUT	FIGURE (USD)	SOURCE
US Software Developer, average annual wage (2024)	~$154,076	DataUSA (Census ACS PUMS / BLS) — datausa.io/profile/soc/software-developers
US Computer & Information Systems Manager, average annual wage (2024)	~$155,405	DataUSA — datausa.io/profile/soc/computer-information-systems-managers
Benefits as share of total compensation, private industry	≈ 29.9% (a 1.43× loading on base wage)	BLS, Employer Costs for Employee Compensation — bls.gov/news.release/ecec.nr0.htm
Fully-loaded developer cost (own calculation: ~$154,076 × 1.43)	≈ $220,000/yr (~$18,000/mo)	derived from the two rows above

For context on the senior-talent ceiling, not a typical SMB number:

CONTEXT FIGURE	FIGURE (USD)	SOURCE
Principal Engineer, total compensation (2025)	~$551,000	Levels.fyi 2025 — levels.fyi/2025/
Staff Engineer, total compensation (2025)	~$457,000	Levels.fyi 2025 — same as above

These two are a market-top reference. Treat them as the ceiling, not the rate you would pay for a first engineering hire.

Hiring timelines

One figure supports the point that hiring is slow and getting slower. It covers all roles, not engineering alone, and we flag that caveat wherever the number appears.

CLAIM	FIGURE	SOURCE
Time-to-fill rose from 43.6 days (2022) to 59.7 days (2025), +37%, all roles combined	43.6 → 59.7 days	Greenhouse recruiting benchmarks — greenhouse.com/recruiting-benchmarks

A word on what isn't here

You will notice some numbers the market throws around that this book does not quote: a headline day rate for a fractional CTO, a tidy "a bad hire costs 30% of first-year salary" figure, a specific elite-versus-everyone-else delivery multiplier. We left them out on purpose. We couldn't trace any of them to a source that survives a second look, and a book that asks you to anchor every spend to a checkable outcome can't itself run on numbers it can't stand behind. Where a figure wouldn't hold, we wrote the sentence without it.

All URLs were live and checked during the research pass for this book. Currency conversions, where a source published in a currency other than US dollars, use the basis recorded alongside each figure.