The FDE Handbook

A Practical Playbook for Forward-Deployed Engineers

Richard Xu

0to1.site

September 2026 · v1.0

Open-source repo: github.com/RichardXu001/the-fde-handbook

Preface · Where Do AI Projects Actually Get Stuck?

> A complex system that works is invariably found to have evolved from a simple system that worked.
> — John Gall, Systemantics

In 2026, nearly every industry summit is saying the same thing: AI is what will decide this round of business competition. Capital is chasing the "agent" narrative, chip makers keep setting new market-cap records, and any startup pitch with the word "AI" in it gets an easier hearing. That heat is real.

But point the camera at enterprise production systems and the picture looks completely different. A report from MIT NANDA puts a number on it: 95% of organizations get no measurable P&L impact from generative AI [1]. In plain terms: the GPUs are bought, the tokens keep burning, and the books show nothing coming back.

Behind the number are real people and real frustration. In 2021, an engineer who had done deployment work wrote on Hacker News that it had made him a worse engineer [2]. Five years later, an AI delivery engineer in China put it a different way, but the meaning hadn't changed: after four iterations, the biggest thing the project delivered was emotional reassurance for leadership [3].

Every demo looks stunning. So why does everything stall the moment it hits production? Where, exactly, is the block? Different people give different answers. What follows is an attempt to put those answers side by side and see whether they are, in fact, describing the same thing.

I. The Block Is Real

The block in AI adoption isn't a feeling — it's a fact with numbers behind it.

MIT NANDA's 2025 report, The GenAI Divide, gives three figures [1]:

[1] MIT Project NANDA, The GenAI Divide: State of AI in Business 2025

[2] Every engineer should do a stint in consulting (Hacker News thread)

[3] "AI Job Sense: Stop the Agent Rat Race"

- **95% of organizations report no measurable P&L impact from AI** — the report's own phrase is "no measurable P&L impact," which is not the same claim as "95% of projects failed," though the two are often conflated;
- Only about **5%** of custom enterprise AI tools ever reach production;
- On procurement, buying (about 67%) far outpaces building in-house (about 33%) — but buying doesn't solve the problem either: purchased tools stall at the demo stage just the same.

The third figure is worth pausing on: most companies' first instinct is to buy, and the report shows the block persists after purchase. That points to a shortage not of AI tooling, but of **the capability to install that tooling into the business**. The report's own explanation points the same way: the block isn't model capability — it's a learning gap (organizations haven't yet learned how to use it) and integration.

II. The First Wall: Integration

So where, specifically, does "installing it into the business" get stuck? The answer on the ground is more mundane than it sounds.

Most enterprises where "AI doesn't work" hit a wall that has nothing to do with the model — it's **the systems wall that's gone unfixed for twenty years**: data sits in seven or eight systems that don't talk to each other, workflows span three different tools, and the interfaces were written by someone who left a decade ago. AI hasn't fixed that wall, and agents won't fix it automatically either (more in Chapter 9) [4] .

Model capability has jumped several orders of magnitude in a few years. This wall hasn't moved an inch. It doesn't discriminate by model — swap in a stronger one, and the wall is still there.

III. The Second Wall: Permissions

The second wall is more hidden: agents can't budge the enterprise's permission system.

An agent only holds the permissions of the human user behind it, and unlike a human colleague, it can't talk its way around a permission wall with a favor or a workaround when it gets stuck [4] — so agents deployed into enterprises often die at step one: they can't even get the data they need (more in Chapter 10).

[4] a16z: Box CEO on AI agents and why enterprises can't keep up (YouTube)

Integration and permissions share one thing in common: **neither is a technical problem — both are "nobody is responsible for fixing this" problems.** The model vendor isn't responsible. The software vendor isn't responsible. The customer's IT department has its own boundaries. The block lives in the cracks nobody owns.

IV. The Week After the Demo

The following is a composite drawn from multiple real lessons; the people and company are fictional, used to show the shape of the block.

A customer-service agent performs flawlessly on demo day: smooth answers, accurate conclusions, a beautiful dashboard. Then it connects to the real ticketing system for its first week —

Day one, the permission wall: ticket data lives in three systems, and the agent's account only got read access to one of them. Waiting on approval.

Day three, a tool misfire: permissions finally cleared, and in one batch operation the agent wrote test data into a production field. The rollback ate an afternoon.

Day five, unmeasurable regression: after a version change, the support lead says it "feels worse," the engineer says "I can't tell the difference" — nobody has a baseline, so nobody can prove anything either way.

Week two, the business doesn't use it: front-line support still works from the old ticketing UI — "the new system takes one extra click." The system is live and green. Zero users.

Every one of these walls gets its own chapter later in this book: permissions (Chapter 10), tool boundaries (Chapter 11), provable outcomes (Chapters 12–13), nobody uses it (Chapter 15). For now, the point is just to see them lined up together — **what separates demo from production was never "a bit smarter." It's a series of walls, each with a name.**

V. Quality Has a Measurable Depth

Beyond the walls, there's a subtler, quantifiable block: quality.

An insurance practitioner posted a number on X (formerly Twitter): their internal agent's real-world accuracy at first launch was about 70%, against a contractual commitment above

[5] Inside an Applied AI company (Pace long-form post)

98% [5] . Those 28 points between 70 and 98 are exactly what "delivery capability" means in practice — not something you get by tuning a prompt, but something built up through a whole loop of evaluation and regression testing (fully unpacked in Chapters 7 and 12).

There's a slower gap too: model capability jumps a full generation within a year, while adoption on the ground lags far behind [6] . This matches a feeling a lot of people share — AI capability sprints forward, and the world feels strangely unchanged, much the way self-driving's technology curve has climbed steeply for years while the traffic outside looks exactly the same as it always did. This "adoption gap" corroborates MIT's data: the block isn't capability, it's adoption and integration. Chapters 15 and 28 take this up in full.

VI. Enter the Protagonist

Put the threads together:

- The block is real and measurable;
- Its names are integration, permissions, evaluation, organizational incentives — none of them a model problem;
- What these walls share is that **nobody is directly responsible** — no one owns "shipped, usable, maintainable" end to end.

Which gives us this chapter's core claim: **model capability is a commodity; the capability to install a model into a specific business is scarce.** The 2026 surge of the FDE (Forward-Deployed Engineer) role is the market pricing exactly that scarcity.

It isn't a new invention. Palantir ran this exact playbook over twenty years ago, built around the forward-deployed engineer as an organizational form: put the engineer on-site with the customer, accountable for what actually runs in production. In 2026, the market needs it again — consulting giants have turned it into a service line, and model companies are building their own delivery organizations in-house (Chapter 4 lays out the timeline day by day).

The six parts and twenty-eight chapters that follow take apart the question "which bridges need fixing, and by whom": what the role is (Part One · Role and Boundaries) → how delivery happens (Part Two · From Discovery to Production) → the engineering wall (Part Three · Engineering Foundations) → adoption and handoff (Part Four) → the business (Part Five · Business and Productization) → people and organization (Part Six).

The Gall's Law quote that opened this chapter says the same thing another way: a complex

[6] Y Combinator: The FDE Playbook for AI Startups (Bob McGrew)

system that works was never designed that way — it evolved, step by step, from a simple one that worked. And "evolved" only happens because someone stayed accountable for every one of those steps.

The first item on that list is the most basic one: what is an FDE, actually — what result is this person on the hook for?

Author's Preface

I. In 2026, Everyone Is Asking the Same Question

In 2026, the FDE — Forward Deployed Engineer — is suddenly everywhere: the consulting giants have turned it into a flagship service line, the model companies are stepping in to build delivery organizations of their own, and job boards keep adding postings with the exact title week after week.

Beneath the noise, I kept hearing the same set of questions — and the people asking them were getting easier to name:

- Engineers ask: how is this different from outsourcing, from on-site consulting? Will I be the one told to transfer into this role?
- Tech leads ask: we hired AI engineers and the projects still won't land — should I be hiring an FDE instead?
- Customers ask: the vendor says "it's working great" — how do I actually know when it's ready to go live?
- Founders ask: how do the economics of this business actually work?
- Skeptics ask (and with reason): isn't this just consulting wearing a new name?

Behind all these questions sits the same more fundamental thing: **the stretch of road between an AI demo and customer value — which bridges need building, and who builds them.** That is what this book answers. It does not answer "is FDE a fad" — that's a subjective question. It answers "what is an FDE, how do you do the job, how do the numbers work, what happens to the people" — the practical, operational questions.

II. Why I'm the One Writing This Book

I'm an AI entrepreneur myself, and I've done a number of enterprise AI projects — at the start, feeling my way forward as I went, like everyone else.

Plenty of articles have been written about FDE, but books that treat it systematically are rare. And on this topic, a great deal of the material has already warped through round after round of retelling — every link in the chain simplifies and extrapolates, and by the time it reaches the last link, the truth has gone as blurry as a photocopy of a photocopy, ten generations deep.

So I wrote this book from my own project experience and the experiences of friends around me, combined with community discussions, public materials, and a number of excellent open-source projects — every cited source is marked in the body text and the appendices.

The cases in this book come from three kinds of channels: **my own delivery and observation experience; first-hand accounts from people I know personally (all used with consent, and de-identified); and public materials** (interviews, podcasts, community discussions, company announcements, open-source projects, and more — see Appendix C).

III. How to Read This Book

The book is six parts, and every chapter answers one question — and only one:

1. **Part One · What an FDE Actually Is** — what the role really is, who it resembles and who it doesn't, whether it's a passing wave, whether AI will replace it.
2. **Part Two · How the Problem Gets Solved** — once you're on site at the customer: how to pick the problem, how to build a minimum viable deployment, how to answer "when can this go live."
3. **Part Three · Engineering Foundations** — data and permissions, constraining agent actions, evaluation, proving yourself, cost — the eight walls between PoC and production, climbed one at a time.
4. **Part Four · The Real Work Begins After Go-Live** — why nobody uses the system after it ships, why this has to be an executive-sponsored project, how to hand the system over so the customer can run it themselves, and why China plays by a different set of rules.
5. **Part Five · How Does This Business Actually Work?** — how to count the big ledger, how to charge, why customization must decline, how field experience becomes product.
6. **Part Six · People and Organization** — the FDE's day, the capability model, making the transition, building the team, going solo — and the role's endgame.

Each chapter's primary audience is marked right in the briefing at its head — engineers, tech leads, customers, founders, skeptics: each will find their own chain of questions laid out in order in the table of contents.

Readers with time to spare should start at Chapter 1 and read straight through. Readers in a hurry can enter at Part Two or Part Three — the engineering chapters stand on their own. Readers who just need to make a decision can go directly to Part Five for the economics, then circle back to Part One.

IV. Before We Begin

What this book has to say really comes down to one line: **do the things that don't scale — at scale.** The first half is the FDE's daily reality: every customer engagement is one-of-a-kind, unrepeatable grunt work. The second half is where the craft's value lives: distill what can be repeated out of that grunt work into product, so the next customer gets there faster.

The name FDE may change; the responsibility won't. If you take away only one thing from this book, I hope it is this one.

One last, honest thing: at most, this book can flag the potholes I've already fallen into, so you skip a few steps of wasted motion. Real judgment still has to be forged on-site, at the customer, bruise by bruise. However closely you read, it's no substitute for stepping onto the field yourself.

You're welcome to follow me on X (Twitter) at @richard_xu_001, or come browse my personal site at 0to1.site, where I keep sharing what I'm seeing and learning.

So begins the book.

Richard Xu

September 2026 Hongqiao, Shanghai

Part One · What an FDE Actually Is

> This part has five chapters, and together they answer one question: **what, exactly, is this role that just exploded onto the scene.**

What it is (Chapter 1, setting the title aside to look at the responsibility structure); which roles it resembles, and which it doesn't (Chapter 2, three testable behavioral tests); whether it's just outsourcing in a new coat, or a passing trend (Chapter 3, pulling the hype apart from what's real); why it exploded specifically in 2026 (Chapter 4, a verifiable timeline, not a mood); and whether AI will replace it (Chapter 5, which tasks are being eaten, and which layer isn't).

The five chapters form a single chain of questions: **Define → Distinguish → Interrogate → Time It → Ask the Hardest Question**. Each chapter creates the problem the next one has to solve. Chapter 1 sets up the definition of a "responsibility structure." Chapter 2 immediately uses it to cut adjacent roles apart. Chapter 3 lets the skeptics test whether this definition is old wine in a new bottle. Chapter 4 asks why, if it isn't hype, it exploded specifically now. Chapter 5 asks the sharpest question of all: what if AI eats it?

This part is written primarily for **anyone who wants to see clearly what FDE actually means**: engineers about to enter the field, managers writing or reading a job description, and anyone who's turned skeptical after being bombarded with the term. Engineering detail isn't covered here — that's Part Three's job. How the money works isn't covered here either — that's Part Five's job. This part does exactly one thing: turn the three letters "FDE" into something you can pick up and test for yourself.

Chapter 1 · What an FDE Actually Is — What Result Is This Person on the Hook For?

> The price of greatness is responsibility.
> — Winston Churchill, Harvard address, 1943

In 2026, three people all have "Forward Deployed Engineer" on their business cards. Their jobs look nothing alike.

The first spends about 75% of every day on model optimization and engineering code. The second serves ten customers at once, splitting time evenly between meetings and development, always on call for production issues. The third sits in a customer's office gathering requirements, builds a prototype with AI on the spot, and hands it to the backend team once it's confirmed.

Same title. Three different jobs.

So judging whether a role is really an FDE role can't rest on the business card. It has to rest on what result the person is on the hook for, and how far that responsibility goes.

I. First, Admit It: The Title Doesn't Help

Define FDE off any single company's job posting, and you won't get the full answer.

The same title can mean completely different jobs and requirements. Baseten lists model optimization and infrastructure engineering as the core of the role [1]. A European company's 2026 remote FDE listing asks for Python, SQL, and the ability to build a PoC [2]. One FDE at a major Chinese tech company described a different routine in a Linux DO forum post: visit customer sites to gather requirements, build a prototype with AI, then hand it to the backend team once it's confirmed [3]. Three companies, one title, three different jobs.

The reverse also happens: some roles carry the full weight of FDE work under a different name. A company might call it "Solutions Engineer," "Delivery Engineer," "AI Application Architect," or simply "the person on the Customer Success team who writes code."

[1] Baseten, "What I Learned as a Forward-Deployed Engineer Working at an AI Startup" (Het Trivedi)
[2] European remote FDE job description (reposted by @afahmy_dev)
[3] "Anyone Out There Actually Doing FDE Work?" (linux.do thread)

This isn't just naming chaos. FDE took shape inside specific companies first, and different organizations then adopted it on their own terms — so the job's shape naturally varies. But the core responsibility underneath tends to look much the same.

II. A Condensed Definition

To understand why FDE exists at all, start with Bob McGrew. He lived through Palantir's FDE model firsthand, then went on to become OpenAI's Chief Research Officer. On a Y Combinator podcast, he summed up the role this way: an FDE **delivers many capabilities to a single customer**. A traditional software engineer does the opposite — **builds a single capability for many customers** [4] .

A product engineer turns one capability into a general-purpose product and ships it to many users. An FDE faces one specific customer and has to wire several capabilities into that customer's systems, permissions, workflows, and working relationships — get them running, and keep them running. McGrew breaks the process into four moves: get on-site, prototype fast, iterate continuously, wire it into production. All of it happens on-site, at the customer.

At Palantir, the people doing this work split into two teams: Echo and Delta. Echo is the business front line — on-site analysts and account managers who decide what problem the customer should solve. Delta is the technical front line: deployment engineers in the strict sense, who turn the plan into a system that actually runs [4] [5] . Chinese companies often don't split the work across two teams at all — the same person or small group carries both [5] .

This division doesn't run along seniority, tech stack, or whether someone knows machine learning. It splits along exactly one line: who decides what to do, and who makes it happen.

One clarification before moving on: FDE refers to both a role and an organizational unit. Its complete form isn't one person — it's a small team stationed at the customer. Echo/Delta and the Chinese trio in Chapter 26 are the same small team, cut two different ways; a solo FDE is that same unit, compressed into a single person. Most of this book takes an individual's point of view, but behind that "individual" always stands a team that can be split apart or merged back together.

III. Where an FDE's Time Actually Goes

[1] Baseten, "What I Learned as a Forward-Deployed Engineer Working at an AI Startup" (Het Trivedi)

[4] Y Combinator: The FDE Playbook for AI Startups (Bob McGrew)

[5] Tencent Research Institute, "FDE Model: Industry Observations and Practice"

Job descriptions don't always reflect the real work; calendars usually tell the truth better. Het Trivedi, an FDE at Baseten, publicly broke down his own time: about 75% on software engineering and model optimization, 15% on technical consulting, 10% on customer relationships [1] .

These numbers clear up two common misconceptions. An FDE isn't a "presales rep who happens to code," closing deals mainly through conversation — engineering eats most of the time. Nor is an FDE just an outsourced coder sent in to write to spec — real time still goes into understanding the problem and working with the customer.

This is one company's design for the role, not an industry average [1] . But it makes one thing clear: FDE work is mostly engineering, and that engineering has to happen inside a customer relationship. This pattern comes back again and again later in the book: what you're responsible for is where your time goes.

That's a U.S. sample, though. Chinese FDEs likely spend more time untangling business problems and maintaining customer relationships. One practitioner said about 60% of their time goes to exactly that, with technical work taking a smaller share — Chapter 18 works through the full China time breakdown.

IV. The Four Responsibilities

On a Chinese developer forum, one FDE summed up their workflow this way: "Dig into the business → design the process-improvement plan and technical architecture → build → train / roll out gradually / adjust business processes → feedback / monitor / fix / optimize cost" [6] .

This process breaks down into four categories of responsibility an FDE has to carry.

A complete FDE is on the hook for all four at once.

1. **Discovery** — deciding which of the customer's scenarios is actually worth doing. Not building whatever the customer asks for, but sorting a "vague wish" into a "problem worth solving." (Chapter 6)
2. **Production** — getting it running safely, stably, and maintainably in the customer's real environment, not stuck in a demo. (Part Three)
3. **Adoption** — making sure real users actually use it; going live isn't the same as being used. (Chapter 15)

[1] Baseten, "What I Learned as a Forward-Deployed Engineer Working at an AI Startup" (Het Trivedi)

[6] linux.do post #2580046

4. **Feedback** — carrying what was learned on-site back into the product, so the next customer moves faster. (Chapter 22)

Looking back at that forum post: "dig into the business" is Discovery, "build" is Production, "train / roll out gradually" is Adoption, and "process improvement, distilled" is Feedback. It lines up closely with the four responsibilities (see Figure 1–1).

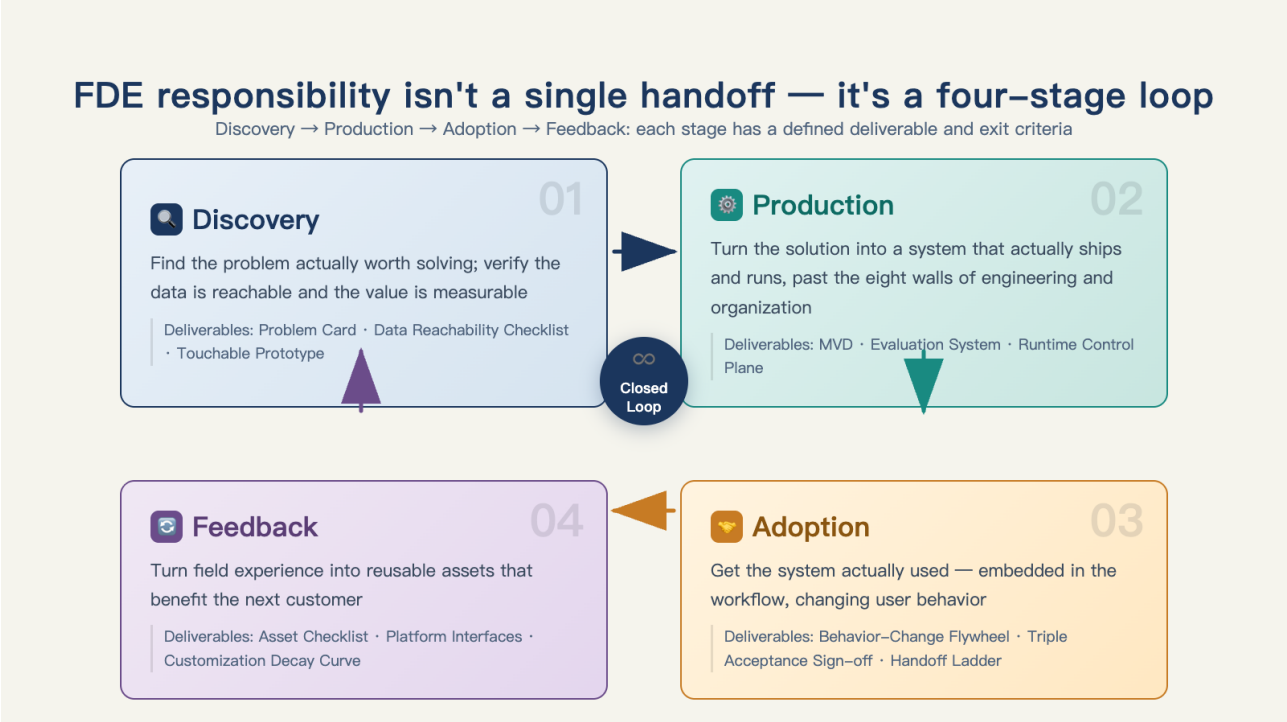

Figure 1–1: FDE responsibility isn't a single handoff — it's a four-stage loop of Discovery, Production, Adoption, and Feedback.

Carrying all four at once is what makes the complete form of FDE. Carrying just one or two can still be part of FDE work — that isn't a ranking. Most roles only cover one piece to begin with; what matters is knowing which piece is yours.

V. Three FDEs, Same Week

The following is an anonymized composite, used to demonstrate how the four responsibilities apply.

P, a model-platform FDE. Mornings are spent tracking down latency spikes in the customer's model inference. Afternoons, he walks the customer's algorithm team through the trade-offs of a batch-processing interface. His responsibility centers on Production — discovery comes from platform colleagues, driving adoption is the customer's own team's job, and feedback flows through the platform roadmap. On the four-part map, he mostly sits in the second box.

M, a multi-customer FDE. Ten active customers, fifty hours a week, split evenly between meetings and building. Morning: signs off on a data interface design in a joint session with Customer A. Midday: handles a production alert for Customer B. Evening: writes a status report for Customer C. He touches all four responsibilities, but none of them gets enough time — his risk isn't capability, it's attention diluted across ten customers.

W, an on-site prototyping FDE. Inside the customer's office: mornings go through requirements with the business team, midday assembles the confirmed workflow into a clickable prototype with AI tools, afternoons hand the prototype to his company's backend team to build the production version. Discovery and prototyping are his; Production and Feedback live elsewhere. On the four-part map, he covers only part of the first box.

All three carry the FDE title. Mapped against the four responsibilities, they carry very different loads.

Which one is the "real" FDE? All three earn the title. The more useful question is where each one sits. Only once the position is clear does it make sense to talk about performance metrics, career transitions, or business models.

VI. To Close

This chapter comes down to three sentences:

- FDE isn't a fixed job description — it's a **responsibility structure**: accountability for results that actually run, stay maintainable, and can be measured inside a customer's business.
- Don't judge by the title. Check four things instead: **does the code go to production, does field experience flow back into the product, is adoption someone's job, and how far does the accountability for results actually extend.**
- Wide variation under the same title is the norm. The four-part map measures position, not rank.

With the standard set, a new question shows up immediately: where exactly does this responsibility structure end and its "old neighbors" — presales, consulting, outsourcing, customer success — begin? That's Chapter 2.

Chapter 2 · FDE vs. Presales, Consulting, Outsourcing, and Customer Success — What's the Actual Test?

> If names are not correct, language will not be in accord with the truth of things; and if language is not in accord with the truth of things, affairs cannot be carried to success.
> — Confucius, The Analects, "Zi Lu"

"What's the difference between FDE and presales, or consulting?" gets asked constantly, and the answers never agree. Some say FDE understands the business better. Others say it leans more toward engineering.

Start with two roles. Both go into an enterprise customer and deploy an AI system that actually runs the business; both write code, connect systems, and watch outcomes. Company A calls the role "Solutions Engineer": he writes demo code, leaves once the deal closes, and bills travel on a per-diem basis. Company B calls similar work "FDE": the code goes to production, field experience flows back to the team, and pricing is tied to business results.

Look only at daily tasks, and the two job descriptions read almost identically. Look at the contract and how each is measured, and the responsibilities turn out to be entirely different.

This is common. FDEs at different companies can simply be different roles to begin with. The same person, moving to a different contract, can go from FDE to Solutions Engineer even if the day-to-day work doesn't change.

If the name can't define the role, what should?

I. Comparing Job Descriptions

The most intuitive approach is a table sorted by "does it write code, who is it for, when does it leave":

Role	Writes Code?	Who For	When They Leave
Software Engineer (SWE)	Yes, product code	Own company's product	Keeps iterating, never leaves
Presales / Solutions Engineer (SE)	Yes, demo code	Prospective customers	Leaves once the deal closes
Consultant	Mostly not; produces plans and roadmap docs	Customer's business side	Leaves after milestone sign-off
Outsourced / on-site developer	Yes, code the customer specifies	Customer project	Leaves when the contract ends
Customer Success Manager (CSM)	Mostly not	Signed customers	Ongoing service, never leaves

Presales can end up writing production code. A consultant can end up on-site for a year. An outsourced team can end up knowing the business cold. The table lists what a role usually does, and the exceptions live exactly inside that word "usually." Once the exceptions pile up, the table stops being useful for judgment — it's only good for reminding people not to assume (see Figure 2-1).

Figure 2-1: Roles that touch the same customer differ in where responsibility ends and whether experience flows back.

Why are these exceptions so common?

Jove Zhong, who heads AI Agent FDE at Cresta, put it this way: "Anything you can teach the

[1] Tencent Research Institute, AI Lens Roundtable, Episode 6: "FDE Non-Consensus Views and a Field Guide from Silicon Valley Founders"

customer to do isn't FDE." [1] He contrasts this with traditional SAP implementation. SAP's rules are explicit enough to write into a manual; once the customer learns them, their internal team can keep maintaining the system. That kind of work can be fully handed off — it sits closer to consulting or implementation.

AI agents, in his account, don't work that way. Even a customer with hundreds of engineers on staff may not be able to keep hallucination rates and response latency reliably under control. The problem isn't that the customer lacks documentation — it's that the judgment calls shift with the model and the scenario, so they can't be written into a manual once and for all. When that's the case, someone has to keep working the problem on-site.

That distinction exposes the table's hidden assumption: it treats "does it write code" and "who is it for" as fixed properties. But when the work gets too complex to hand off, the role's boundary shifts along with the project. What looks like Solutions Engineer work today may need a permanent on-site presence tomorrow. Any work that can be fully handed off — whatever it's called — eventually becomes a one-time knowledge transfer.

Since the work itself shifts, content alone can't define the role. This chapter looks at three observable tests instead: whether the code goes to production, whether field experience flows back into the product, and whether pricing is tied to effort or to results.

II. Test One: Where the Code Goes

First question: **where does this person's code end up running?**

Presales writes demo code to close a deal. Once the demo is over, that code has usually done its job. An FDE writes production code so the system keeps running — the code has to pass the customer's review, enter the deployment pipeline, and meet the customer's operational requirements.

The same line of code carries different weight depending on where it lands. A bug in demo code might cost a deal. A bug in production code might hit the customer's actual business.

AI coding tools have widened how much an FDE can touch directly. In the past, one FDE could barely keep up with multiple customers' codebases at once. Now, as long as he can state clearly what result he wants, AI can help him modify TypeScript, Swift, or database code [1] . That doesn't just let him ship code to the customer — it also makes it easier to carry problems and

[1] Tencent Research Institute, AI Lens Roundtable, Episode 6: "FDE Non-Consensus Views and a Field Guide from Silicon Valley Founders"

fixes discovered on-site back into his own company's product line [1] .

III. Test Two: Where the Experience Goes

Second question: **once the project ends, where does what was learned on-site go?**

Experience can end up in three places, each mapping to a different role.

The first: back into the product. The FDE carries requirements, problems, and reusable components discovered during customization back to the product team, turning them into capability the next customer can use directly. This is the path FDE experience is supposed to take.

The second: into a report. Experience gets written up as closing documentation and case studies, providing material for the next pitch. This is closer to project-based consulting.

The third: it stays in someone's head. When the person leaves, the experience leaves with them. This is common on outsourced projects.

Bhaulik Patel, who has studied this business structure closely, put it bluntly: an FDE organization where nothing flows back is, at bottom, **"an expensive professional services organization hiding inside a software company"** [2] — whatever the business cards say.

Run this test against a common Chinese setup: the FDE visits the customer, gathers requirements, builds a prototype on-site, and hands it to the backend team once confirmed. The experience stops at the handoff document. By this second test, that's closer to **outsourced prototyping** than complete FDE work [3] . That's not a knock on it — under certain procurement structures, this division of labor can make perfect sense (Chapter 18 unpacks this). The test is for locating a role, not judging it.

IV. Test Three: How It's Priced

Third question: **is this role priced by effort, or by result?**

Bill by the day, and the customer is buying time — how much that time produces is the customer's own call to judge. Bill by result, and the customer is buying an improvement in some outcome — the vendor also carries some of the risk that the outcome doesn't improve.

[2] @deployengineer (Bhaulik Patel), essay series

[3] "Anyone Out There Actually Doing FDE Work?" (linux.do thread)

[4] Crossroads × Rolling AI, "A Conversation on FDE" (podcast)

Neither model is inherently better, but each steers behavior differently. An organization billing by the day cares whether its people are fully utilized. A team billing by result cares more about whether the problem actually got solved. One AI delivery partner put it plainly: bill by the day, and the upside from any efficiency gain has nothing to do with you [4] .

Of the three tests, pricing is the easiest to verify — it's written into the contract and the invoice. A role can claim to care about customer success all it wants; how it's billed reveals what accountability it actually carries.

V. One Project, Three Different Structures

The following example exists purely to show how the three tests combine.

A retail company wants to deploy a "smart replenishment agent." Same project, three contract structures, three different roles growing out of it —

- **Structure One: outsourced, billed by the day.** The vendor sends two engineers on-site for six months, billed daily. Their code goes to production, but the experience stays with them personally; nobody guarantees the outcome, and hours worked are the deliverable. Against the three tests: ① production code √ ② no feedback × ③ billed by effort √ — this is outsourcing, on-site or not.
- **Structure Two: fixed-scope delivery plus a feedback clause.** Same engineers, but the contract now states that "reusable components built during customization belong to the vendor and flow back to the product line." Experience starts flowing back, but accountability for the outcome still ends at sign-off. Against the three tests: ① √ ② √ ③ mixed — this is delivery with product accountability, already standing on FDE's threshold.
- **Structure Three: revenue share on results.** No fee unless the baseline improves; a share of the improvement beyond that. The vendor proactively cuts fake requirements and works only on what's measurable. Against the three tests: ① √ ② √ ③ billed by result √ — this is FDE with the complete responsibility structure.

What this demonstrates: what changes the role isn't the engineer, and it isn't the technology — it's the contract structure and where the experience goes. The same engineer, in three arrangements, carries three different kinds of accountability.

VI. The Three-Question Position Card

Collapse the three tests into a card — useful for self-assessment, or for turning back on an

employer in an interview:

> **The Three-Question Position Card**
> 1. Does this role's code go to production? (Ask the engineering lead: is last project's code still running? Who maintains it?)
> 2. Where does field experience go once the project ends? (Ask the interviewer: did the last project's custom components make it into the product line? Is there a "feedback review" mechanism?)
> 3. Am I measured by hours or by results? (Ask HR and the business lead: what metric does this role carry? Adoption rate? Retention? Or billable-day utilization?)

There's no standard answer to the three questions — what matters is whether the organization can answer clearly at all. An organization that can answer all three has at least a clear accountability structure, whatever the role is called. An organization that can't answer any of them has a role whose title, however new, says nothing about who's actually on the hook.

VII. Reality Is Messier

Time for honesty: fitting all three tests cleanly is an ideal type — reality is rarely black and white.

Most real-world roles are hybrids. Someone might spend seventy percent of their time on production code and thirty supporting presales; most organizations only manage partial feedback loops.

So use these tests for positioning and planning, not for pass/fail grading. First see clearly where you — or the role you're targeting — actually sit on all three tests. Then decide which direction to move, and what it will cost to get there.

The tests exist now, and they can locate a position. But once the position is located, a new question follows immediately: is this responsibility structure actually new, or is it something old wearing a new name?

Chapter 3 · Is FDE Just Outsourcing in a New Coat, or a Passing Trend?

> The thing that hath been, it is that which shall be; and that which is done is that which shall be done: and there is no new thing under the sun.
> — Ecclesiastes 1:9

One FDE consultant admitted it outright: "FDE is old wine in a new bottle." He breaks the business down into "a technical consultant plus an on-site SaaS engineer" — "some of the hype is real, some of it isn't." [1]

Coming from a practitioner, that might seem to settle the chapter right there. But it's only half right. This chapter draws the specific line: which symptoms are just hype, and which underlying need won't disappear once the buzz fades. Meeting the skepticism head-on is the only way to get the question straight.

I. Four Layers: What's Left When the Project Ends

Judging whether a team is really doing FDE work doesn't start with the job title. Start with what's left after the project ends [2] . This breaks into four layers:

- **Nothing but a system left with the customer**: that's outsourcing. Once the system ships, the relationship ends.
- **Some experience carried back, but not reusable next time**: that's project-based delivery. The experience becomes closing documentation, useful reference material for the next pitch.
- **Experience turned into a Skill, a template, or a product capability**: that's FDE. Field experience gets organized into something reusable.
- **That accumulation visibly lowers the cost of the next customer**: that's FDE at scale. The team starts benefiting continuously from its past projects.

These four layers line up with Chapter 2's second test — where the experience goes — but push one question further: once the experience is back, does it actually make the next customer

[1] Turning Point (破局点), Episode 31: "FDE, Chinese Style"

[2] Tencent Research Institute, "FDE Model: Industry Observations and Practice"

faster? Chapter 21 covers how customization declines; Chapter 22 covers how experience gets back into the product. Neither gets unpacked here (see Figure 3–1).

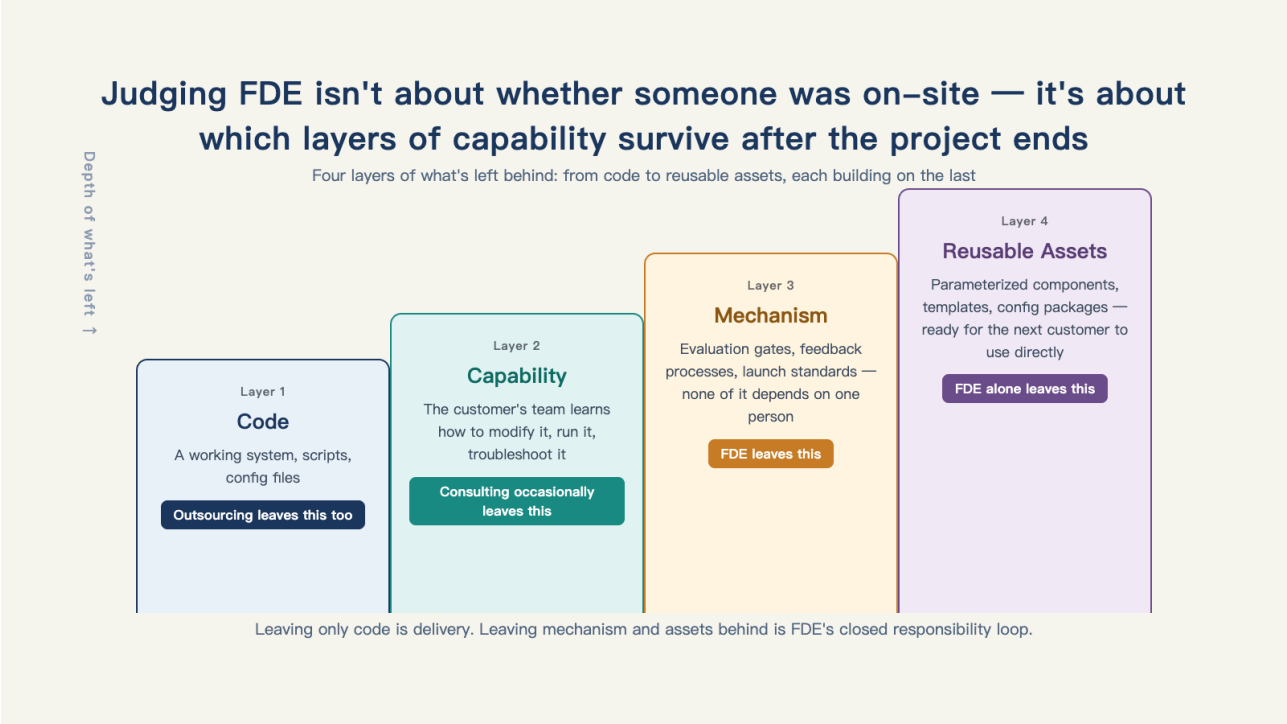

Figure 3–1: Judging FDE isn't about whether someone was on–site — it's about which layers of capability survive after the project ends.

On developer forums, the "is FDE just consulting in a new coat" debate keeps resurfacing. One practitioner put it more sharply: a lot of "FDE training" amounts to "selling the information gap around AI to anxious small–business owners with money to spend" [3] .

Practitioners admit there's an old side to it, while the media and training market keep repackaging it as something new. So where, exactly, is the bubble?

II. Three Concrete Symptoms of Hype

Separating out the common patterns, the hype shows up in three concrete ways:

First, title inflation. Renaming an existing implementation or presales role to FDE is the easiest way to chase the trend. The most common version: repackaging "can write code with an AI coding tool" as FDE. One user on a Chinese developer forum put it this way: "Vibe–coding in Claude Code doesn't make you an FDE by itself — that just solves the development

[3] "Anyone Out There Actually Doing FDE Work?" (linux.do thread)

[4] linux.do post #2580046

problem." [4]

Second, the training grift. Courses sell jargon and tool mechanics to anxious career-switchers, pitched as "switch now or it's too late." This isn't unique to FDE — any hot new title attracts this kind of training. What makes FDE different is that its core is real delivery accountability, and that can't be learned from a course alone — it has to be built in real situations (Chapter 25 unpacks this).

Third, organizational drag. Some companies create the role to follow the trend, without building a feedback mechanism or measuring by results. FDE ends up as nothing more than expensive on-site labor. This is exactly the situation practitioners describe as "an expensive professional services organization hiding inside a software company" [5] .

All three share one thing: they land at the weak end of all three tests from Chapter 2 — no production code, no feedback loop, billed by the day. The hype isn't in the word "FDE." It's in how people use it.

III. The Real Half

Now the other half. Telling a bubble apart from a durable need can't rest on how much buzz there is — it has to rest on whether the conditions underneath will disappear.

"Prompt Engineer" makes a useful contrast. After ChatGPT drove generative AI into the mainstream, the title ran hot, then cooled fast. The reason: prompting techniques depended on whatever the model happened to be weak at that moment. Every generation the model improved, a chunk of those techniques lost their value. Prompt engineering was a skill the tools themselves were bound to erode.

FDE isn't facing a skill that can be mastered in isolation — it's facing an ongoing relationship: the customer has a real production environment, and a vendor has to deliver a system into it. In that relationship, someone always has to own connecting the system, keeping it running, and keeping the customer actually using it. A stronger model doesn't automatically sort out the customer's permissions, workflows, and internal politics. The deeper an agent reaches into the business, the more complicated system integration can get, not less [6] .

There's financial evidence too. An engineer who spent years at Palantir recalled that the

[5] @deployengineer (Bhaulik Patel), essay series

[6] a16z: Box CEO on AI agents and why enterprises can't keep up (YouTube)

[7] "Reflections on Palantir" (Nabeel Qureshi)

company's 2023 gross margin ran around 80%, versus roughly 32% for a traditional consulting giant like Accenture [7] . If FDE were just relabeled on-site outsourcing, its profit structure should look like a consulting firm's. Instead it looks more like a software company's. Where does the difference come from? From field experience being repeatedly folded back into the product — exactly the fourth layer above. Chapter 22 explains the mechanism behind it.

IV. Conclusion: Half Right, Half Wrong

This chapter's conclusion splits into two halves, each holding on its own:

- **The need is real.** Getting AI safely into a customer's production environment is a durable need. Production environments are complex, and trust between organizations has to be built — neither condition disappears anytime soon.
- **The role won't always be called FDE.** FDE, an upskilled consulting team, or a customer's own internal team can all end up carrying this work. The market will change titles, and the buzz will fade. The problem stays.

How long the concept itself stays fashionable, which parts of the work AI ends up replacing, and whether the title itself eventually disappears — those questions belong to Chapters 5 and 28.

One question about timing is still left standing: if this isn't a passing trend, why did it blow up specifically in 2026? Chapter 4 lays out what happened in the first half of that year.

Chapter 4 · Why Is Everyone Talking About FDE in 2026?

> We always overestimate the change that will occur in the next two years and underestimate the change that will occur in the next ten.
> — Bill Gates, The Road Ahead (1995)

In May 2026, two things happened between two Mondays. On May 4, Anthropic announced it was forming a delivery joint venture with Blackstone, Hellman & Friedman, and Goldman Sachs [1] . On May 11, OpenAI set up a holding subsidiary called DeployCo, reportedly funded with over $4 billion and staffed with around 150 FDEs on day one [2] .

This made a lot of noise. Start with an **event timeline** to see exactly why FDE exploded in 2026.

I. Timeline: 2003 to 2026

Date	Event
2003	Palantir founded; the FDE practice takes shape by the mid–2000s
c. 2007	Early employees embed in customers' classified environments for the first time; on–site delivery takes its recognizable form
2026–02	Microsoft publishes its Agent Factory white paper; Fujitsu discloses AI–driven development investment in an investor briefing
2026–03–18	Accenture and Microsoft jointly launch an FDE practice; official language cites "thousands of AI engineers" embedded directly with customers
2026–05–04	Anthropic forms a delivery joint venture (with Blackstone, H&F, and Goldman Sachs)
2026–05–11	OpenAI launches DeployCo (a holding subsidiary; reportedly over $4 billion, with roughly 150 FDEs acquired)

Stretch this table into two curves and it's easier to read: the complexity of integration and adoption has been there since the ERP era and never really went away; AI model capability is just the new variable that only started getting plotted after 2023 (see Figure 4–1).

[1] Anthropic, "Building a New Enterprise AI Services Company with Blackstone, Hellman & Friedman, and Goldman Sachs"
[2] Forward deployed engineer is AI's hottest job as OpenAI and Google race to hire

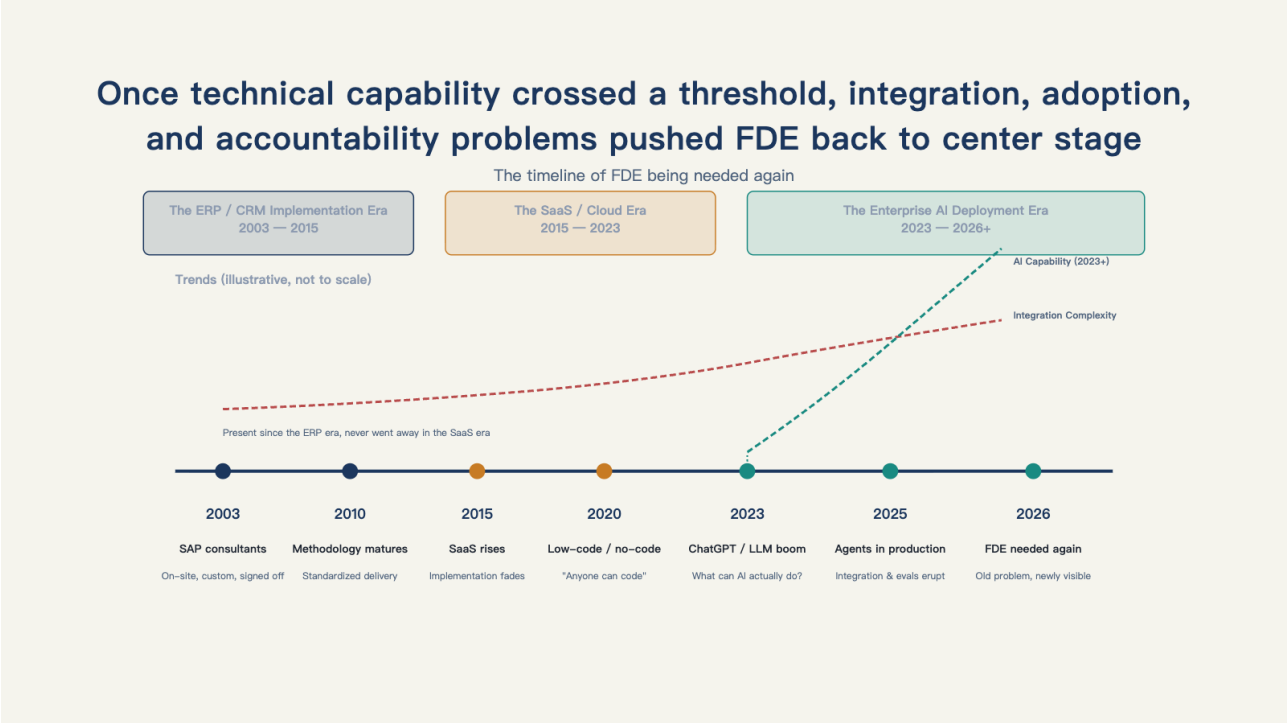

Figure 4–1: Once technical capability crossed a threshold, integration, adoption, and accountability problems pushed FDE back to center stage.

II. Reading the Timeline

Connect the raw facts on the timeline, and a layered reading emerges.

These layers all point to the same thing: consulting giants, model companies, cloud vendors, and the originator of the FDE concept itself — organizations with essentially no overlap — arrived at the same market pain point and the same entry angle almost within the same window. The territory had grown too big for any of them to keep ignoring.

Layer one: the consulting giants move in, and FDE turns from tactic into procurement category. Once Accenture made it a flagship service line [3] , "forward deployment" stopped being one product company's private playbook and became a standard line item in the enterprise procurement catalog — a buyer could now purchase it like any standard product: budget it, put it out to bid, sign it off. Turning it into a category also raised the bar — everyone who enters after starts from a higher floor.

Layer two: model companies enter, because "just selling an API isn't nearly enough." The most capable model companies building their own delivery organizations amounts to a public

[3] Accenture Newsroom, "Accenture Launches Microsoft Forward Deployed Engineering Practice to Help Organizations Scale AI Across the Enterprise"

admission: model capability alone doesn't translate into customer value. Delivery is the second half of monetizing a model — the enterprise-scale version of the Preface's argument that the block is real.

Layer three: cloud vendors enter too, on the logic that "compute only becomes revenue once it lands in production." Cloud vendors sell compute, but customers only buy it on the premise that they can actually put it to work in production — a pile of idle GPU quota doesn't renew a contract. The motive is identical to the model companies': selling raw capability isn't enough; someone has to connect that capability into the customer's actual business before payment keeps flowing.

Layer four: the originator of the FDE concept starts turning it into an AI product. Palantir's "AI FDE" agent — natural-language operation, closed-loop execution, bound to the user's existing permissions [4] — means the FDE concept has begun iterating on itself: the execution work of a human FDE is being packaged into the product.

Stack these layers up, and they answer the question above: the flurry of moves in 2026 wasn't a media event. It was a sequence of actions — the market pricing "delivery capability" as a scarcity independent of "model capability."

III. Why the Productivity Payoff from General-Purpose Technology Always Arrives Late

The reading above answers "who's entering." But an uncomfortable question won't go away: with capital spending across the economy already this large, why hasn't the aggregate productivity number visibly jumped? "AI matters" isn't a good enough answer to that challenge. Set it against a longer technological history, and it actually comes into focus.

Economic historian Paul David documented the lag in factory electrification: electricity entered American factories in the 1880s, and it took roughly forty years before the productivity payoff clearly showed up [5] . The reason wasn't that factories didn't know how to use electricity — it's that early factories simply swapped the steam engine for an electric motor in place, while keeping the line shafts, belts, and multi-story layouts that had been designed around steam in the first place. Electricity's real advantages — distributed power, independently scheduled processes, more flexible floor plans — only showed up once the factory floor itself was redesigned around electricity. By the historical statistics that paper cites, electric motors

[4] Palantir Foundry, "AI FDE" product documentation

[5] The Dynamo and the Computer: An Historical Perspective on the Modern Productivity Paradox (Paul A. David)

accounted for under 5% of driven manufacturing horsepower in the U.S. in 1899, had climbed past half by around 1919, and reached roughly three-quarters by 1929 — swapping the equipment itself wasn't the slow part; redesigning the floor and the workflow around electricity was [5] .

Personal computers went through the same stage. Economist Robert Solow's endlessly quoted line from a 1987 book review — "You can see the computer age everywhere but in the productivity statistics" — wasn't saying computers were useless. It was saying that companies at the time were using computers as expensive typewriters: type on the computer, print it out, bind it, hand it to a person to file — the process itself never changed [6] .

Different technologies don't map onto each other mechanically, but this technological history offers a verifiable warning: general-purpose technology tends to get adopted first, get crammed into the old process next, and only shows up as a productivity leap after the organization itself gets redesigned around it.

AI is easy to misread today for exactly this reason: model capability improves monthly, and you can see it. The cost of organizational restructuring — redrawing department boundaries, budgets, permissions, performance metrics — plays out over quarters or years, and you can't.

The productivity number not jumping yet doesn't prove AI has no value — it shows the organizational rebuild hasn't caught up with model capability. Which echoes exactly the previous section's reading: everyone independently putting real money into "delivery" this year is exactly the work of catching organizational restructuring up to speed.

IV. Closing, With a Cool Head

A set of structurally different organizations — consulting giants, model companies, cloud vendors — made real, structural investments in "delivery" within the same window. Why that investment is necessary already has a precedent: the path electricity and the personal computer both walked before it.

Whether the role survives long-term, whether the concept cools off, whether today's entrants come out ahead — that's a separate question. The timeline proves investment, not return. Return is Part Five's ledger, and Chapter 28's final question.

The timeline itself is still holding one thread, buried in layer four: the concept's originator has already turned FDE into an AI product. The people who invented this role are now using AI to

[6] We'd Better Watch Out (book review) (Robert M. Solow)

replace the role's own execution work. What's left of the role after that is the question that follows directly once the facts are on the table — Chapter 5 answers it head-on.

Chapter 5 · Will AI Replace the FDE?

> A computer can never be held accountable, therefore a computer must never make a management decision.
> — IBM, internal training material, 1979

Palantir, the company that brought the FDE concept into the world, turned it into an AI product with its own hands in 2026: the "AI FDE" agent operates Foundry through natural language, executes autonomously, runs in a closed loop, and every action stays bound to the user's existing permissions [1] . The concept's originator was the first to automate its own execution work.

That sounds like an obituary for FDE. It's actually this chapter's first piece of evidence — it demonstrates with precision exactly what AI eats (tasks) and what it can't (accountability). Separating those two is the premise of everything that follows.

FDE exists in the first place because it sits at the intersection of **technical capability, business value, and acceptable cost** — only when "AI can do it," "the customer will pay for it," and "the return justifies the investment" are all true at once does a person need to connect it into production. Every step model capability takes forward, every notch inference cost drops, that intersection widens, and the range of business AI can cover on its own grows with it. Understanding this makes clear what the question "will AI replace FDE" is actually asking — not whether the role disappears, but which direction the boundary of that intersection is moving.

[1] Palantir Foundry, "AI FDE" product documentation

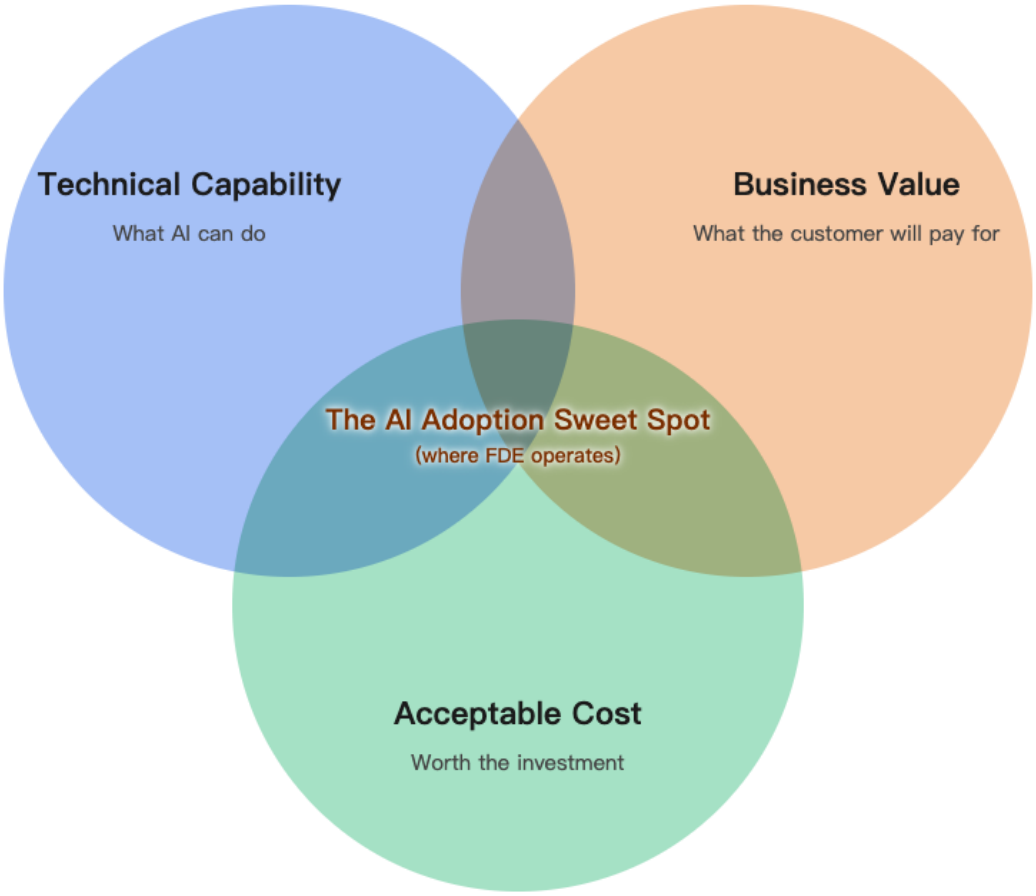

Figure 5–1: No two of technical capability, business value, and acceptable cost are hard to overlap — the hard part is getting all three to overlap at once. That intersection is FDE's operating range.

I. Will AI Replace FDE?

AI never replaces a role — it replaces **tasks**. Whether a role gets replaced comes down to whether the tasks left over still add up to enough responsibility to hold a job together. Put differently: inside FDE's work, which tasks are being eaten, and which aren't?

Break FDE's work down into a task list, tag each layer, and three tiers emerge:

Tier	Tasks	Status
Being automated	Writing integration code, boilerplate connectors, generating first-draft docs, demo materials, routine data processing	Largely eaten
Partially automated	Evaluation design, failure classification, comparing solution options, drafting data contracts	AI drafts, humans finalize
Hard to automate	Owning accountability, building and maintaining customer trust, vouching for permissions, negotiating sign-off, being accountable for adoption	Essentially untouched

The first tier's list keeps growing, and faster. That's not a threat — it's leverage. The real dividing line sits in the third tier, and each item there deserves its own "why."

II. Why the Accountability Layer Is Hard to Replace

First, B2B procurement is fundamentally about transferring risk. What a customer buys from an outside team isn't just capability — it's "someone to answer for it if something goes wrong." A model can't sign a contract, can't be held accountable, can't hold permissions — these aren't technical limits, they're legal and organizational ones. An entity that can't be sued can't hold up its end of a contract that says "accountable for the outcome."

Second, permissions belong to people, not to models. Practitioners on the ground keep observing the same thing: an agent only holds the permissions of the human user behind it, and unlike a human colleague, it can't work around a permission wall by asking a favor when it gets stuck — so it often dies at the very first step of enterprise deployment (Chapter 10 unpacks this) [2] . The permission system trusts "a person plus an identity." AI can only inherit that trust through a person.

Third, judgment is the price of admission for staying in the decision loop. One practitioner nailed it: what a vendor gets paid for is "aesthetic judgment and the ability to build the engineering system around it." He added an observation worth sitting with — once a person's ability falls behind AI's, they can no longer tell good AI output from bad, and end up dominated by it, reduced to an "AI puppet" [3] . Flip that around, and staying in the decision loop requires your judgment to keep outrunning the tool in your own hands.

[2] a16z: Box CEO on AI agents and why enterprises can't keep up (YouTube)

[3] Turning Point (破局点), Episode 31: "FDE, Chinese Style"

[4] Silicon Valley 101, Episode 240: "The Hottest New Job in Silicon Valley — FDE" (YouTube)

Fourth, some information can only be gathered on-site, by a person. One Silicon Valley practitioner made the case this way: he spends his days in meetings and meals with the customer, pulling back the tacit information and unspoken understanding AI can't reach; his evenings turn that information into AI's context. AI can produce several options and a table weighing their trade-offs, but choosing which one — and living with the consequences of that choice — still has to be a person. "We're AI's hands and feet." [4]

Put the four together, and you get the full logic behind this chapter's epigraph: a computer can never be held accountable, so the responsibility structure has to rest on a person who can be. AI keeps getting better at **doing things right** — the execution layer. But the responsibility for **doing the right things** — the judgment-and-guarantee layer — still needs a person who can be trusted, held accountable, and entrusted with permissions to carry it (see Figure 5-2).

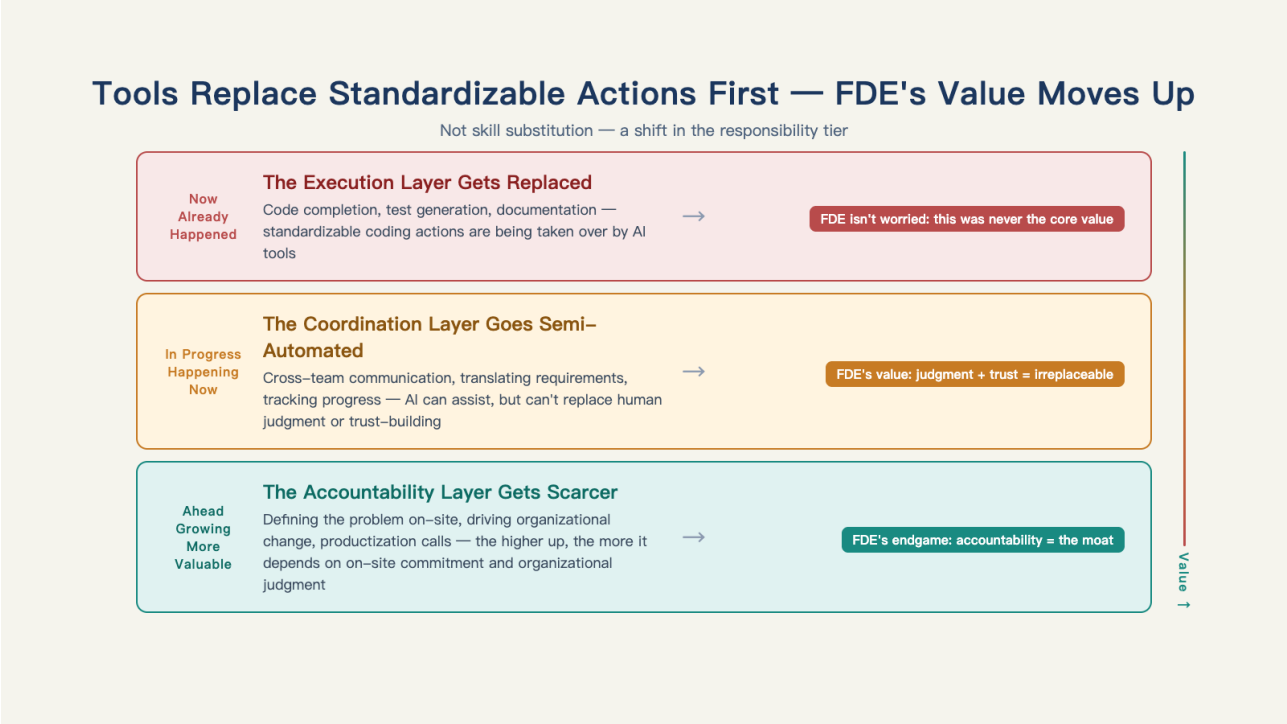

Figure 5-2: Tools replace standardizable actions first; FDE's value shifts upward with field accountability and productization capability.

III. Three Ways of Doing the FDE's Job

An illustration of how FDE work looks across different eras — this scenario is illustrative, not a description of any specific company's actual practice.

The customer says: "Fill in this quarter's sales data. We need it by Monday."

The FDE of ten years ago (human): flies to the customer's site, negotiates data access with IT,

writes scripts to pull data from three systems, cleans it, reconciles it, and hands over an Excel file — one week. What he learns — what this customer's data actually looks like, how their processes run — stays in his head.

If it's handed entirely to AI (the technical ceiling, as Palantir's documentation describes it): it receives the instruction in natural language, understands the intent, generates the Foundry operations, executes the data pipeline within the user's existing permissions, logs everything, and pauses for clarification on anomalies — same job, same day. It doesn't know, and doesn't need to know, this customer's organizational history. It inherits the user's permission boundary — and every limitation that boundary carries.

Today's FDE (human plus AI): the person breaks the instruction into pieces AI can execute, and watches the exception and approval points; the execution itself is handed to AI to amplify. The person's day shifts from "writing scripts" to "defining the problem, reviewing the actions, handling exceptions" — exactly the second- and third-tier tasks.

Compare the three, and the conclusion lands on the person: in the combined form, AI amplifies a person's output per unit of time even further — but that amplification only works if **the person is standing in the judgment-and-guarantee layer**.

Anyone still standing in the execution layer will find less and less room to stand.

IV. How the Structure Evolves

Look at two scales.

Zoom in: the engineer from Chapter 1 juggling ten customers at once is already leaning on tool leverage to cover all ten — an early form of the "human plus AI" combination. The next step is agent leverage: the person handles only judgment, guarantees, and the customer relationship, while agents amplify the execution. His "ten" will turn into a bigger number, but his worth was never staked on that number — it's staked on which layer he stands in.

Zoom out: McGrew has a clever way of putting it — today's AI startups are, in effect, the FDEs of foundation model companies. The model companies build capability; the AI companies pack it into specific industries [5] . This structure recurses upward, layer after layer: there's always a gap between capability and application — the gap just keeps getting pushed to a higher floor. The job of delivering capability the last mile never disappears. It just resurfaces one floor up.

[5] Y Combinator: The FDE Playbook for AI Startups (Bob McGrew)

So what's the endgame? Worth imagining: if that day ever comes — the model hits its limit, stops improving, and cost can't fall any further — the expansion of this intersection stops too.

At that point, FDE would most likely split into two paths: one settling into vertical SaaS, selling standardized software; the other settling into professional consulting, selling standardized service. Today's FDE is neither — precisely because the territory is still expanding and the moment for standardization hasn't arrived, it has to stay in the middle, holding the ground where technology and product can't yet reach. How long that middle ground lasts depends entirely on how much further the model still has to evolve.

V. To Close

Two closing lines, for two kinds of reader.

If you're already doing FDE work: don't stake your worth on the execution layer — it's being eaten right now, not as a future risk, but in the present tense. Move your center of gravity to the judgment–and–guarantee layer: an eye for evaluation, designing sign–off, building trust, the nerve to be held accountable. Chapter 24 goes deep on how to build these.

If you're about to enter the field: the good news is the barrier to entry keeps dropping. The bad news is "knowing how to use the tools" keeps getting cheaper. Get one thing straight before you walk in: come because "I'm willing to carry the responsibility," not because "I know how to use these new toys."

But knowing what it is and knowing how to do it are two different things. The next chapter goes into the field: right after the customer says "we want AI," the first make–or–break task is digging the one problem actually worth solving out of a pile of vague wishes.

Part Two · How the Problem Gets Solved

> This part has three chapters, and together they answer one question: **from the moment a customer says "we want AI" to the moment a system is actually running in production, what does that path look like?**

The three chapters form a single pipeline: each one's output is the next one's input. First, find the right problem (Chapter 6 — discovery isn't brainstorming, it's filtering: take the customer's vague wish list, widen it out, then run it through five filters until a single worthwhile entry point remains; pick the wrong problem, and the harder the team works afterward, the more completely that effort goes to waste). With the problem settled, build the first version that's actually "real" (Chapter 7 — a Minimum Viable Deployment, or MVD: real data, real users, a real baseline — measuring the actual gap is what this stage delivers). With the gap measured, you can finally answer the question the customer is bound to ask (Chapter 8 — "so when can this go live" isn't a technical moment, it's a two-way acceptance contract). Without a problem that survives Chapter 6's five filters, an MVD has nothing to be "minimal" about; without the baseline Chapter 7 measures, there's nothing to write on Chapter 8's acceptance sheet.

This part's first readers are **people about to walk onto the customer site**: vendor-side delivery engineers, the technical lead running the deployment, and the customer-side owner who has to ask questions — and answer them — at the review table. The engineering details of moving a system into production — data, permissions, evaluation, billing — belong to Part Three. How this business gets paid belongs to Part Five. This part covers exactly one stretch of road: from a vague wish to a signed acceptance.

Chapter 6 · On-Site at the Customer — How Do You Find a Problem Worth Solving?

> It is a capital mistake to theorize before one has data.
> — Arthur Conan Doyle, "A Scandal in Bohemia," The Adventures of Sherlock Holmes

At the first on-site meeting, the customer sits down and hands you a list of seven wishes: smart quoting, meeting-minute generation, photo-based quality inspection, vendor-email sorting... Someone in the room raises a hand for each one, and every single one sounds like something "AI could probably do."

It sounds like a wishing well. But the delivery team only has the headcount to build one of them.

How do you choose? Many projects don't fail because the execution was bad — they fail because the wrong problem got picked at the very start. Pick the wrong problem, and the harder the team works, the more completely that effort is wasted.

This chapter is about exactly that: where to dig for the real problem, how to run what you dig up through five filters, and what's left once you've cleared all five.

I. The Real Need Is Rarely Found in a Meeting

Start with how discovery actually works. Most projects understand "discovery" as "interviews": book a string of meetings, take a stack of notes, go back to the office and write it up. The biggest risk on that path isn't missing a question — it's that **every sentence you hear has already been translated once**. By the time it reaches you, it's no longer first-hand data.

A widely-circulated field method calls this step "shadowing": follow a real user through a real day of their work — not interviewing them, but sitting beside them and watching them work, watching which systems they open, which spreadsheets they copy-paste between, where they frown, which "official process" they quietly route around. What you're watching for is **the workaround, not the workflow**: the official flowchart tells you how the organization is supposed to run; the workaround tells you how it actually runs. Anthropology calls this method participant observation; the Toyota Production System calls it genchi genbutsu — "go and see for yourself."

[1] Fan Bing, Forward Deployed Engineer (FDE) (XDash open-source book)

OpenAI's FDE team did exactly this on the John Deere project: they flew out to Iowa and followed agronomists and farmers into the fields, watching how spraying decisions actually got made, which information actually entered the decision, and how the hard deadline of the growing season governed everything. The old process they eventually replaced — agronomists calling farmers door to door, giving verbal advice on equipment use — appears in no document anywhere. You only see it by going out to the field [1] (see Figure 6–1).

Figure 6–1: The real need hides in real actions, exception handling, and unwritten rules.

Why does it have to be seen with your own eyes? Because **users themselves can't tell you** — and "can't tell you" happens on two levels.

The first level is that they won't say it outright. Ask directly, "do you think this plan is good?" and you'll mostly get politeness back. There's a small startup-world book devoted entirely to this problem: Rob Fitzpatrick's The Mom Test. The title comes from a pointedly cruel piece of advice — don't ask your mother "is my business idea any good?" Because she loves you, she'll say yes no matter what. A customer's compliments work the same way. The fix is to move the question from opinion to fact: don't ask "do you think you need this" or "would you use it in the future" — ask "the last time this came up, what did you actually do about it." Ask about the past, not the future; ask about one specific instance, not the general case. Politeness can't be checked against anything. Facts can be cross-examined.

[1] Fan Bing, Forward Deployed Engineer (FDE) (XDash open-source book)

[2] "100 Questions About FDE" (open-source ebook)

The second level is that they genuinely don't know it's worth saying. A soldier's need for a roadside-bomb early-warning tool is one no interview will ever surface, because soldiers don't know "software could actually do that" — they assume the dread that comes with every patrol is just part of what patrolling is [1] . A Chinese enterprise AI deployment consultant makes a strikingly similar point: pain points rarely get spoken out loud; "they're hiding in the manual work people repeat every day, in the mess at some post nobody wants to own. You can't dig them up with a questionnaire — you have to camp out on-site" [2] .

When you can't make sense of it, there's a blunt fallback: ask "why" three layers deep — the first layer nails down what they want, the second asks why now, the third asks what happens if it doesn't get done. By the third layer, the real problem usually surfaces. A customer says "I want a knowledge-base Q&A system." Three layers down, the real problem might be that a veteran expert is about to retire, thirty years of experience can't be carried out the door, new hires take half a year to get up to speed and make constant mistakes in the meantime — the surface ask is a knowledge base, the deep problem is knowledge transfer.

There's a subtler layer of distortion still: **the person in front of you is not the user at all**. When the people making the request are a mid-layer digitalization team, "you think you're building a system for the actual users, but the whole conversation runs through a group of people carrying their own digitalization KPIs — by the time they relay front-line needs to you, it's already passed through a filter." The fix is straightforward: route around the middle layer and reach the front line directly — "even two or three interviews with real users produce a stronger signal than ten meetings with middle management." When there's genuinely no way to reach them, list the requirement assumptions one by one and have the middle layer confirm each against what the front line actually wants — not what the middle layer assumes they want [2] .

A real example: a company deployed an agent to handle refund requests. It read every request, checked it line by line against the refund policy, and rejected anything that didn't qualify — every single rejection was "correct." A few weeks later, long-standing customers started to churn. No one could figure out why, because every rejection the agent made really was called for under the written policy. Eventually someone sat down with the people who used to process refunds by hand, and found an unwritten rule that had never made it into any document: orders paid with a corporate credit card get approved automatically — those customers order every month, and haggling over one refund could cost the whole account. The policy never mentioned this; it was tribal knowledge the human processors had worked out for themselves years earlier. The fix wasn't a different model — it was turning that check into a fixed rule in the workflow, applied

[2] "100 Questions About FDE" (open-source ebook)

[3] "FDE Engineer 101: Bridging the Gap Between AI and Business" (Chinese-language compilation video)

before the model ever makes a judgment call [3] .

This case lays bare what discovery is really about: **the model being correct is not the same as the business being correct — a layer of unwritten process knowledge sits between the two.** That knowledge isn't in the requirements document, isn't in the system logs; it only lives in the hands of the people who handle the work every day. There's a similar case: a file-format migration had stalled for an entire year, and everyone assumed it was a technical problem, until someone sat down next to the engineer who kept blocking it and watched him work — he'd always double-clicked files to inspect the data, and the new format had nothing you could double-click. The team built him a tool that night that let him keep double-clicking. Two days later, he signed off on the migration [3] .

Which is why discovery needs people who understand the system to go in first. A digitalization lead at an investment firm, reviewing internal practice, put it this way: when you're packaging an AI skill, "the most important thing is knowing exactly what you're trying to package" — and getting clear on that takes architectural judgment that has to come from someone at the mid-management level or higher, guiding the process from the start [4] . Interviews collect the wish list; the architect judges the real need behind each wish. Both have to happen at once.

II. Filter One: Does This Problem Actually Matter?

Don't rush to assess technical feasibility on the list you've dug up. The first filter asks: **does this problem actually matter inside the customer's organization?**

Bob McGrew, a Palantir alum, put this bluntly: "the scope of a traditional implementation might be you start with something that's pretty close to what the product does, but you want to be solving one of the key problems that leadership has identified. If you're not solving one of the top five priorities for the CEO, it's probably not going to work. They probably won't have the energy to persist through the much more challenging route of getting effectively a new piece of the product built in a way that worked for them." [5] His reasoning isn't technical at all: if the problem requires an on-premise deployment, you'll have to fight the customer's IT team; every layer of the organization has someone who has to say yes before you can move, and those people don't think like a startup does and aren't aligned with the end user's goals — you have to find a way past them. Which is exactly why the problem you solve has to be important enough

[3] "FDE Engineer 101: Bridging the Gap Between AI and Business" (Chinese-language compilation video)

[4] Yilu Tongxing (一路瞳行), Episode 134: "From AI Assistant to Digital Employee"

[5] Y Combinator: The FDE Playbook for AI Startups (Bob McGrew)

that you can "bring in someone from the top to say, 'Yes, give them authority to operate. Give them, you know, the ability to use'" whatever they need [5] .

There's a second reason for the importance threshold: picking your battlefield. Which core production line is AI actually going to touch — R&D, contracting and delivery, manufacturing quality, sales operations, or some piece of customer service? It has to connect directly to delivery, revenue, gross margin, risk, or customer experience — not just "look like it saves time." Take the hard, high-value problem over the easy one; otherwise the payoff is negligible.

This lines up with a view held by some practitioners in China: what most existing AI solutions actually accomplish is "cost and efficiency gains within a limited scope" — bolt a little AI onto the old process, get a bit more efficient, spend a bit less; "your company's original processes, people, and business all stay exactly the same." And "if the people and the organization don't change, AI's capability simply can't come through" [6] .

Put the other way around: a problem worth handing to an FDE is usually one that, if you follow it far enough, leads straight into process and organizational change — that's exactly what earns it a place among the CEO's top five problems, and what makes it worth bringing in someone with the authority to say yes.

One of Palantir's early customers turned filter one into a textbook case. The FDE's first real customer was Airbus's factory in Toulouse; he moved there for a year and spent four days a week on the production floor. The entry point wasn't "here's what AI can demo" — it was the biggest problem the CEO told them about directly: ramping up A350 production. The team responded by doing exactly one thing: pulling work orders, missing parts, and quality issues ("non-conformities") — scattered across multiple systems — into a single interface where people could check off tasks, track parts, see the production schedule, and fuzzy-search past quality issues. Under the hood, none of it was more than basic software functionality, but actually shipping a "best-practice" interface into a real production environment ended up helping roughly quadruple the pace of manufacturing [7] .

The direction was worked backward from the biggest problem the CEO cared about — not forward from what the technology happened to be capable of.

Why does this filter come first? Because if the problem isn't important enough, every filter after it is wasted effort — a successful validation that nobody claims, a deployment that stalls because nobody's willing to sign off. Chapter 16 covers this in full: why AI adoption is an

[6] Turning Point (破局点), Episode 31: "FDE, Chinese Style"

[7] "Reflections on Palantir" (Nabeel Qureshi)

executive-sponsorship undertaking. For now, just hold onto the question: **is this on the CEO's top-five list?**

III. The Remaining Four Filters

Once a problem clears the first filter, it still has to pass four more: **value density, verifiability, data reachability, and tolerance for change**. Together with importance, that's five filters — not a flat checklist of equal-weight items, but a narrowing channel, where each filter eliminates one specific way the project could die (see Figure 6-2).

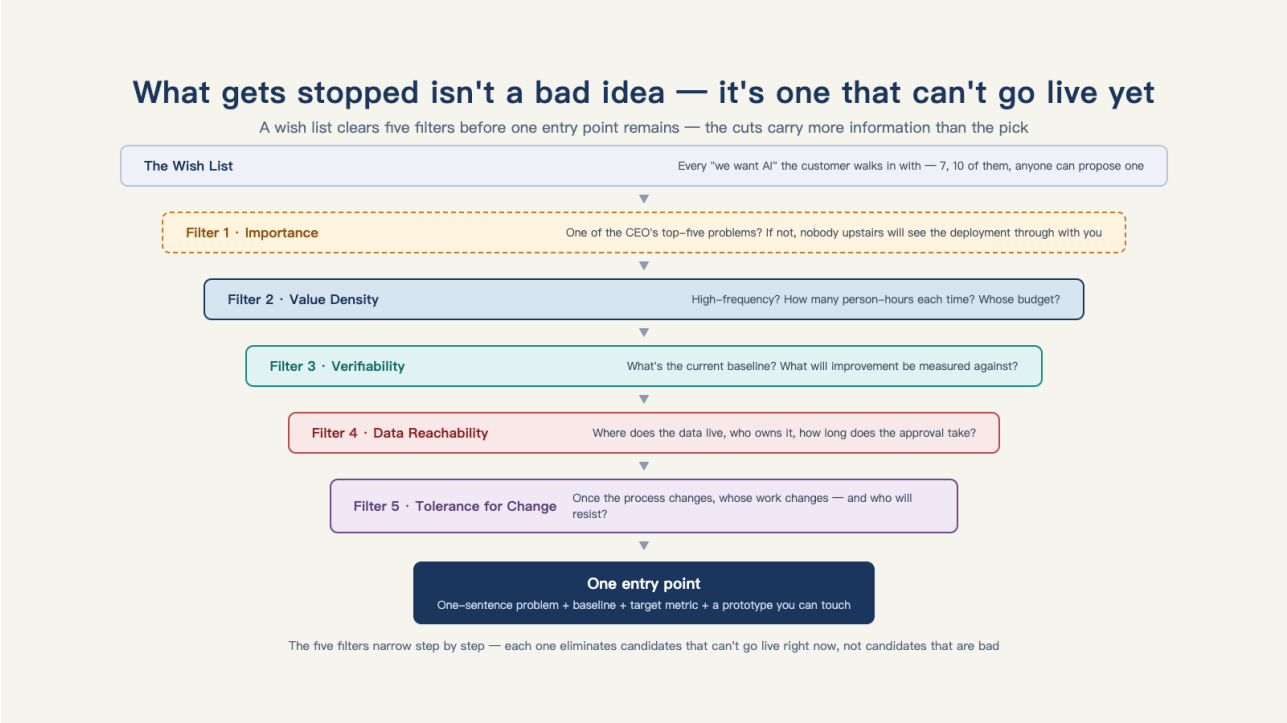

Figure 6-2: What gets stopped isn't a bad idea — it's an idea that doesn't yet meet the conditions for going live.

Filter two: value density. Three questions on-site are enough: who's doing this right now? How many times a day? What happens if it's done wrong? Answer those three, and you've basically diagnosed how healthy this problem actually is.

There are three plain signals for judging value:

High frequency — something done every day pays off from even a small improvement; something that happens three times a year won't save much even at ten times the efficiency;

Currently done by hand — only where the work runs entirely on manual labor, experience, and veteran know-how is there room for AI to step in;

Can get ten times better — a 10% improvement convinces nobody to switch; a 10x improvement

has people fighting to use it.

So "if a scenario is only 10% faster, don't even touch it — save your energy for the high-frequency, hand-done, ten-times-better ground" [2] . Don't forget the list of seven wishes from the start of this chapter: value density isn't just about the "how many headcount does this save" ledger — it's also about the "what market does this open up that we couldn't reach before" ledger [8] .

Filter three: verifiability. What's the current baseline for this problem — the baseline being the status quo: how long does the manual process take, what's the error rate, what's the cost? What will you measure improvement against? For a problem with no baseline, build the baseline first — the AI conversation comes second.

This filter stops the kind of problem where everyone applauds when it ships and, six months later, nobody can say whether it was actually worth it: the impact can't be quantified, and it's the first thing cut when budget season comes around. Experienced teams put "acceptance metrics" before "go live" — starting from a concrete scenario, they have business people and engineers score the output line by line, and the scores feed straight back into iteration. How to build acceptance metrics is a subject Chapter 12 covers in full.

Filter four: data reachability. What data does solving this problem require? How many systems is that data spread across? Who owns the permissions? How long does approval take? This has to be nailed down during discovery — it can't wait until production. In one real case, an internally launched customer-service agent project didn't try real customer data until its second iteration, at which point data-security concerns landed on the table immediately, and the project was forced into a budget-constrained, on-premise small-model deployment that badly capped the model's capability [9] . Data-security constraints are real, but they need to be understood — and designed around — during discovery, not discovered during production. One thing worth flagging when you draw the data map: **in almost every organization, the official data source and the data source employees actually trust are not the same thing** — some veteran employee's private spreadsheet, a report that only ever circulates by email, is often the one people doing the actual work use every day. Connect to the wrong source, and no matter how accurate the system is, nobody will trust it.

Filter five: tolerance for change. Once the solution goes live, whose job changes? Is this

[1] Fan Bing, Forward Deployed Engineer (FDE) (XDash open-source book)

[2] "100 Questions About FDE" (open-source ebook)

[8] Silicon Valley 101, Episode 240: "The Hottest New Job in Silicon Valley — FDE" (YouTube)

[9] "AI Job Sense: Stop the Agent Rat Race"

problem someone's actual power base? Whose position did the process you're about to automate used to support? This isn't office gossip — it's the project's survival. On demo day, the department that cooperated the whole way through applauds, and in the corner, a manager from another department says nothing — the process he's responsible for maintaining is exactly the one you just automated. The mature move is to design a "path forward" for the people who stand to lose, rather than a dead end — turning the gatekeeper into a coach, and turning a veteran's experience into training material for the system [1] . The list of problems this filter surfaces gets reused later — Chapter 15 on adoption and Chapter 16 on executive sponsorship both come back to it.

The filters get run wrong sometimes too — four mistakes come up most often:

Starting from the highest-value problem — usually unverifiable, and you've bet big before you've even started;

Starting from the easiest problem — usually nothing anyone really feels the pain of; even a successful build is just self-congratulation;

Starting from the CEO's personal preference — a scenario the boss picked off the top of his head, usually neither high-frequency nor measurable, and nobody dares push back;

Taking on a "just build us a demo for the board" job — a demo optimizes for looking good, production optimizes for being usable, and the two are different builds from the very first line of code.

IV. Running One Case Through All Five

Here is an anonymized real case.

The customer was a mid-sized manufacturer that walked in with seven wishes: smart quoting, meeting-minute generation, photo-based quality inspection, vendor-email sorting, an equipment-repair knowledge base, a sales-pitch assistant, and contract review.

First, filter one: this customer's CEO cared about exactly two things this year — delivery delays and raw-material cost. Of the seven wishes, smart quoting and contract review hung directly off delivery and cost; the equipment knowledge base was adjacent (downtime hits delivery); the remaining four were things "each department happened to bring up." Four got cut here.

Filter two (value density): smart quoting happens dozens of times a week, each one taking a salesperson two hours flipping through price sheets and past deals, and a mistake means losing money directly — high-frequency, hand-done, ten-times-better: passes. Contract review runs

about thirty times a year, two hours per review — genuinely painful, but not high-frequency; parked for this round. The repair knowledge base: every minute of downtime burns money, but a "knowledge base" by itself doesn't stop downtime — back up one "why" (is the real problem retiring experts taking their knowledge with them, or is it slow parts lookup?) — parked pending further digging.

Filter three (verifiability): smart quoting has clean historical data — three years of quotes and closed prices serve as the baseline, and improvement is measurable: passes.

Filter four (data reachability): the price sheet lives in ERP, past deals live in CRM, and permissions sit with two different departments — reachable, but approval takes two weeks, so the request goes in during discovery. Passes, with a condition.

Filter five (tolerance for change): the sales director who owns the quoting process is a longtime direct report of the CEO's and openly supports the project; the extra step for the sales team — "confirm the AI-suggested price" — can be designed to default to acceptance: passes.

Seven wishes, five filters, one survivor: smart quoting — a one-sentence problem statement ("cut sales quoting from two hours to ten minutes without raising the error rate"), a baseline (current average turnaround time and price-revision rate), and a data-permissions checklist (ERP price sheet plus CRM history, two-week approval). **The process of cutting six carries more information than picking the one that survived.** Every wish that got cut left a record behind: why it's not happening now, what condition it's missing, and when it's worth raising again. That record is itself next quarter's roadmap.

V. The Discovery Deliverables

Discovery is done. What should this stage actually deliver?

Not a slide deck — the discovery deliverables: three things that let next week start with actual work.

1. **A problem card**: a one-sentence problem, the current baseline, and a target metric — one page, no more. If it doesn't fit on one page, you haven't actually thought it through.
2. **A data-reachability checklist**: what data is needed, which system it lives in, who owns the permissions, how long approval takes — the raw record from filter four.
3. **An interactive prototype**: not a demo video, but something the customer's own people

[10] "I Thought FDE Meant Fighting Alone at the Customer Site" — What Two Months on the Job Actually Looks Like (note.com · AI Shift)

can click through with their own hands. A Japanese FDE, two months into the job, wrote down this field lesson: "over a perfect spec document, a prototype people can actually touch. Watch how the customer reacts, and correct course on the spot." [10] A spec document locks a direction in place; a prototype exposes it. The first thing a customer says the moment they get their hands on something real often overturns whatever consensus came out of the previous three meetings (see Figure 6–3).

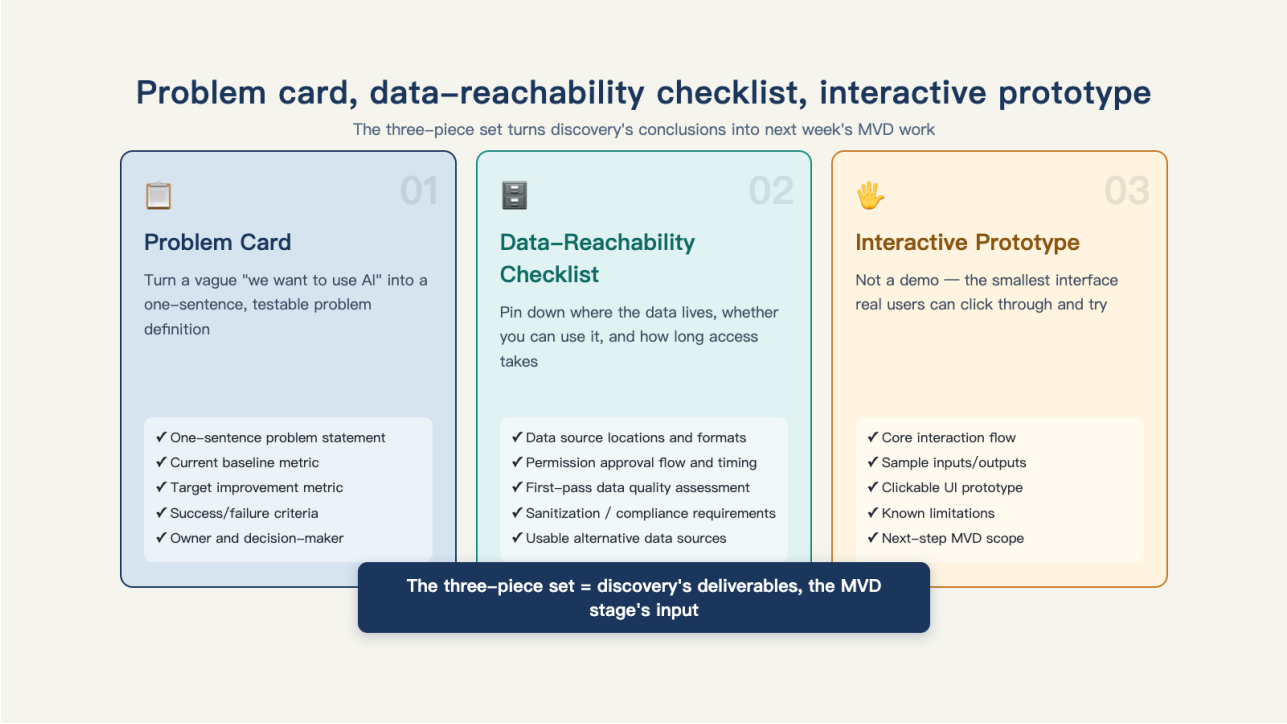

Figure 6–3: A problem card, a data–reachability checklist, and an interactive prototype let discovery hand off straight into next week's work.

With the three–piece set in hand, discovery is over. The one–sentence problem on that problem card now has to become a system that actually runs inside the customer's environment — how small and how real that first version needs to be is exactly what Chapter 7 takes up next.

Chapter 7 · What Actually Counts as a "Minimum Viable Deployment" — Not Just Another Demo?

> Make it work, make it right, make it fast.
> — Kent Beck, software engineer

Problem card in hand, the business side's next question is always: "So when do we get to see results?"

Is your first instinct "let me build you a demo"?

But there's a common trap here: many teams understand an MVD (Minimum Viable Deployment) as "a scaled-down demo" — cut 70% of the features and end up with something that's neither one thing nor the other. The result validates nothing and wastes time on top of it.

A demo optimizes for "looking like it works." An MVD optimizes for "actually measuring whether it works."

These are two different things. What kind of "small" is the right kind, and how do you avoid this trap — that's what this chapter is here to sort out.

I. The Gap

A practitioner at an applied-AI company in the insurance industry wrote publicly about their first day: "The first time we run a new agent against a customer's real workflow, it gets just a little over 70% of it right. On a recent premium audit deployment, our agent read a mess of inbound documents, structured and reasoned over the data, and wrote it back to an antiquated policy admin system. Day one, the customer's own experts graded it at ~70% accuracy. The number we promised in our contract was >98%." [1] His next line is the interesting part: "Everything interesting about my job lives in that gap." [1]

Why do those 28 points matter? Because you will never see them in a demo environment — demo data is clean, edge cases are friendly, there's no legacy baggage, and hitting 100% is the

[1] Inside an Applied AI company (Pace long-form post)

norm. **The gap only shows itself in a real environment**, and the later it's discovered, the more it costs: sign a 98% contract, then discover the starting point is 70%, and every one of those points has to be closed under the customer's watchful eye.

This is exactly why MVD exists. It's not there to deliver a scaled-down system — it's there to **measure the shape of that real gap as early as possible**: with the smallest possible engineering investment, inside the customer's real environment, against a real pain point, verify whether value can actually be created.

Worth clarifying the difference between MVD and MVP while we're at it: an MVP (Minimum Viable Product) validates "should this product exist at all," and the judge is the market. An MVD validates "can this solution produce value for this specific customer," and the judge is that customer's actual business owner.

One letter apart, and the difference is who's judging. This also explains the brutal reality of enterprise delivery: a solution validated successfully at ten prior customers can still fail at the eleventh — different data foundations, different organizational inertia, a differently-shaped pain point. Value can't be inherited from the last customer; it has to be re-validated on this one. So MVD has no template answer, only a methodology.

II. The Three "Real" Conditions

The line between MVD and a demo comes down to three "real" conditions: **real data, real users, real baseline**. Miss any one of them, and what you've delivered is still a demo wearing a different name (see Figure 7-1).

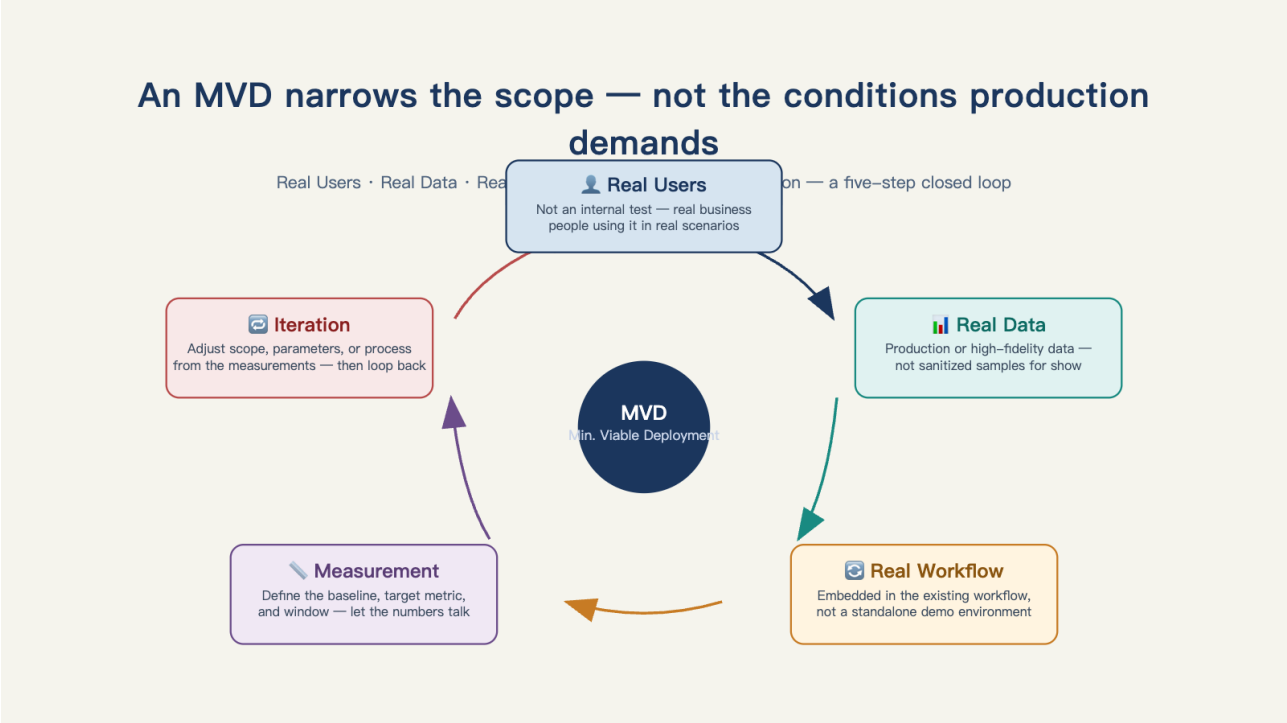

Figure 7–1: An MVD narrows the scope, not the real conditions required for production-grade validation.

Real data. Validating against a customer's sanitized sample data, or demo data you constructed yourself, is the first brick in the graveyard of proof-of-concepts. This isn't about ceremony — it's that real data is full of landmines: field meanings that don't match the documentation, 30% null values, three-year-old encoding rules, and worst of all, data that itself records a broken process. A solution that holds up on fake data hits fields that don't exist in any document the moment it goes live — and by then the real price has already been paid. So there's exactly one hard requirement: the customer has to bring their own real business data. It can be sanitized, it can be a sandbox copy — but **the distribution has to be real**.

Real users. At least one front-line user, actually using it every week and actually giving feedback — not a product owner using it on their behalf, not a courtesy click-through at the acceptance meeting. The test is plain enough: if that user takes a week off, does the work still find its way to this system? A demo performs for the decision-maker; usage grows in the user's own hands. MVD needs the latter.

Real baseline. Before going live, measure the baseline of the manual process: how long the task currently takes, how often it's wrong, what it costs. Without a baseline, the word "improvement" has nowhere to stand — this is the execution-layer continuation of Chapter 6's filter three (verifiability): whatever baseline got promised when that filter was passed has to become a set of numbers written down by the time you reach MVD.

Only once all three "real" conditions are met does an MVD start producing what a demo never can: **a measurable real gap, and a judgment that's been tested by actual use.**

III. Cut the Scope, Not the Value

The three "real" conditions make the gap measurement real; the next question is scope — how small do you cut, and how?

The right way to cut "small" is three cuts: **cut the number of processes** — build one end-to-end process, not a platform; **cut the user base** — serve one team; **cut the agent's boundary** — let the agent take over only the most measurable segment of the process, leaving the rest to people.

All three cuts stand on one non-negotiable rule: **what shrinks is the scope, not the value**. The most common mistake is understanding an MVD as "a gutted version of the big plan" — cut 70% of the features and end up with something that's neither one thing nor the other. The right move is to cut breadth, never depth: don't aim for "smart customer service covering the whole company" — aim for "covering only returns-and-exchanges tickets, but doing it end-to-end with zero human intervention"; don't aim for "supply-chain optimization across the whole group" — aim for "scheduling conflicts on this one production line, but genuinely saving 20 person-hours every week" [2] . The entry point has to be small so it can be deep; and once it's visibly deep, the business side notices it with their own eyes — and starts spreading the word on their own.

As for "how small is small enough," there's a ready-made test: "if users can use it without changing their existing habits, that's enough; if they have to reshape their workflow just to use your system, you've gone too far." [3] A professional team once spent several hundred thousand yuan and three months building full-chain AI automation, only to be outperformed by a simplified version a sixty-something school principal hand-built in three days — he only automated template generation, leaving every other step to be filled in by hand as before, and other principals loved it, precisely because it didn't force them to change their habits.

One comment gets right to the point: "the measure of success was never how advanced the technology is — it's whether the person using it still wants to open it tomorrow." [3]

Beyond scope, there's one more boundary to draw: should an MVD accommodate the customer's legacy systems?

Practice has settled on a middle path: "read old, write new" — deeply compatible on the data side, reading from wherever the customer's data actually lives (read-only, never write), even if

[2] Fan Bing, Forward Deployed Engineer (FDE) (XDash open-source book)

[3] "100 Questions About FDE" (open-source ebook)

that's a legacy mainframe or a spreadsheet on a shared drive; never coupled on the architecture side — code lives entirely inside your own controllable boundary and talks to the legacy system only through an interface, never gets pushed into it.

The reasoning is out in the open: a solution still in validation has better than even odds of being thrown out and rewritten, and the deeper the coupling, the more that costs; writing into a legacy system means going through change management, which runs on a monthly cycle — flatly incompatible with a weekly pace.

On process, defer to human habit rather than system habit — if the user's core action happens in a spreadsheet and email, the interface should live inside a spreadsheet plug-in and email too. A practitioner in China summed it up this way: the customer's core system has been running for a decade, so start with a lightweight plug-in on the periphery — "the test is simple: does this change force the user out of the system they already know. If it forces them out, hold off. If it doesn't, push forward." [3]

There's one more easily-overlooked principle: an MVD is the first step of a staircase, not a miniature version of the destination.

One investment firm's weekly-report automation climbed three steps — first a conversational format that cut out the writing, then an upgrade to voice generation, and finally cutting out even the voice step, reading work logs directly and auto-summarizing the weekly report. Every step could be accepted or halted on its own, and real use of the previous step became the requirements evidence for the next one [4] . Try to jump straight to what step three looks like, and you usually can't even stand on step one.

IV. Measured in Weeks, Not Months

The third non-negotiable rule is **fixing a hard deadline**: an MVD's validation cycle runs in weeks, not months — Palantir's own bootcamp compressed this down to one to five days, and public reporting puts the fastest go-lives at four weeks, with typical deployments running four to eight weeks [2] . The point of the deadline isn't speed for its own sake — it's to **force both sides into honest trade-offs**: whatever can't demonstrate value within these few weeks isn't core value yet. A six-month "minimum validation" will almost inevitably regrow into a big project that wants everything — which is just another groundbreaking for the graveyard of

[2] Fan Bing, Forward Deployed Engineer (FDE) (XDash open-source book)

[3] "100 Questions About FDE" (open-source ebook)

[4] Yilu Tongxing (一路瞳行), Episode 134: "From AI Assistant to Digital Employee"

proof-of-concepts.

A two-week sprint can be broken down by the day:

Day	What Happens
Day 1	On-site alignment: write the "single point problem" with the customer as one sentence, and the acceptance metric as one number
Days 2—3	Connect data: hook up only the minimum dataset this problem needs; lock down permissions, sanitization, and export method on the spot
Days 4—5	Build the skeleton: get the minimum prototype running — one that can query data and answer questions
Days 6—8	Plug into the real business flow: one or two real users start actually using it to do their work
Day 9	Collect usage traces and the issue list
Day 10	Demo and decision: not a feature demo for the IT department, but showing the business side "here's where your problem stands now" — decide on the spot whether to scale up, adjust, or call it off

The customer side has three matching commitments of its own: a business-side owner who can actually make the call, present throughout; an internal data-literate point of contact, present the whole way; and access to real data in place on day one, not "still going through the process." Miss any one of the three, and a two-week sprint is very likely to slide back into a traditional proof-of-concept [2] .

FDE: The Delivery Paradigm Revolution sums up this same rhythm as four steps: **understand the business** — walk in and map the business chain, the pain points, and the value lever first, without rushing to write a prompt or stand up an agent; **build collaboratively** — business experts and engineers turn the scenario into an agent, a workflow, or a prototype together, not the engineering team working alone; **use tooling to guarantee quality** — generate a test set from real chat logs, tickets, and business data, then run automated evaluation and A/B comparison; **validate the business outcome** — compare the AI group against the human group step by step, using people-facing metrics like reach rate, response rate, and conversion rate to prove the value. All four steps come down to one sentence: **sell the outcome, not the software** — customers don't pay for "we used AI," they pay for the business actually getting better.

V. Being Thrown Away Is the First Version's Destiny

MVD is done — what happens to the code? McGrew laid out the two-team division very clearly:

[2] Fan Bing, Forward Deployed Engineer (FDE) (XDash open-source book)

[5] Y Combinator: The FDE Playbook for AI Startups (Bob McGrew)

an FDE delivers a custom implementation fast, on the customer's site — built for this one customer only, roughness allowed, bugs owned; the product and engineering team's job is to look at that implementation and think from first principles about how to generalize it for the next five or ten customers, turning a one-off custom build into a reusable product [5] . One version of code shouldn't be asked to do both jobs at once — run in ten days and stay maintainable for ten years.

Which is exactly why the profile to avoid when hiring for the MVD stage is so specific: someone who cares about craftsmanship, insists on polishing the abstraction layer until it's exactly right, wants to write software built to be maintained for a dozen years — because that isn't this stage's job.

The code a fast-prototyper writes is sometimes beautiful — "but usually not, and that's not the key part of the job either. What matters is someone who can actually go deliver that outcome in the form of software, on a timeline. It may be that the first version they write has to be thrown away, and they write a complete second version." [5]

Put those two things together and it's clear: what an MVD actually accumulates as an asset **isn't code — it's test questions and understanding.**

The test questions are the eval set — given this customer's real inputs, which answer is the right one. The understanding is process comprehension — which requirements are general, which are specific to this one customer, which pitfalls the next customer will hit too. The code gets thrown away and rewritten; both of these travel forward intact. This also directly corrects a common fixation — "turn the MVD code straight into production code": whatever development time that saves is nowhere close to the cost of patching custom code into a production system, which is exactly what Chapter 9 unpacks.

Here's a self-check list to compare against:

Modifying demo code straight into an MVD — every assumption baked into the demo data leaks into the baseline, and the measured gap is fake;

Turning an MVD into a feature bundle — scope goes out of control, two weeks becomes two months, validation turns into development;

"Go live first, figure it out later," skipping the baseline — value can never be proven afterward, and there's nothing to say at the renewal negotiation table.

One last point — the exit condition: an MVD stage is allowed to launch without a complete

[5] Y Combinator: The FDE Playbook for AI Startups (Bob McGrew)

approval chain, without monitoring, without a degradation plan — **but it is not allowed to launch without an eval set.** It's the one thing an MVD must start building from day one, and how to build it is Chapter 12's subject. Only with a working eval set and a measured real gap in hand can you answer the question the customer is bound to ask next — "so, when can this go live?" That's what Chapter 8 covers.

Chapter 8 · What's the Answer to "So, When Can This Go Live"?

> Good fences make good neighbours.
> — Robert Frost, "Mending Wall" (1914)

Three hours into the go-live review meeting, the customer closes their laptop and asks the plainest question in the room: "So — when can this go live?" The room goes quiet for ten seconds. The vendor wants to say "the results are already good"; the business side wants to ask "who's on the hook if it breaks" — both sentences sit right at the tip of the tongue, and neither one gets said.

This scene keeps replaying across the industry because "going live" was never just a technical question — it's a two-way contract nobody has ever drafted: the vendor proves it hit the pre-agreed metrics, and the customer takes on operational responsibility against a pre-agreed checklist.

With nobody drafting the contract, both sentences stay stuck at the tip of the tongue, indefinitely.

This chapter is about how to draft that contract.

I. What Is Actually Being Asked, When Someone Asks About Going Live

Start by standing on the other side of the table. As the person responsible for bringing an AI system into the enterprise, the list of questions running through your head at the review meeting usually looks like this: what if the data connection fails? Who owns the permissions? Who's responsible if it gets an answer wrong? How do we trace accountability if something goes wrong? And when does this actually count as ready to go live?

Run down that list and one thing jumps out: **every item is an acceptance question, and not one of them is a model question.** The model's capability was already settled on demo day. The pressure point was never the model — it's acceptance.

"So when can this go live" sounds like it's asking for a date. What it's actually asking is: **who**

decides, against what standard, that this thing has succeeded.

Without answers to those three things, any date offered is a guess pulled out of thin air. With answers to all three, a date is just the natural, final signature.

So the standard answer isn't a sentence — it's a contract made of three documents: an **outcome acceptance sheet** (metric + baseline + who decides), an **operational readiness checklist** (monitoring / degradation / rollback / on-call), and a **responsibility handoff table** (who's accountable for each incident scenario). One document handles "is it worth going live," one handles "do we dare go live," and one handles "who do we call when something breaks."

II. The Outcome Acceptance Sheet: Five Elements, and Missing Even One Wrecks It

"The results are good" can't be accepted against anything. A metric that can actually be accepted has to spell out five elements in full: **baseline value** (the level of human performance before going live — the commitment from Chapter 6's filter three, and the numbers measured by Chapter 7's Minimum Viable Deployment, land in the contract here), **target value** (the level to be reached after going live), **measurement window** (tracked weekly or monthly), **denominator definition** (who counts, who doesn't), and **who decides** (who has final say) (see Figure 8–1).

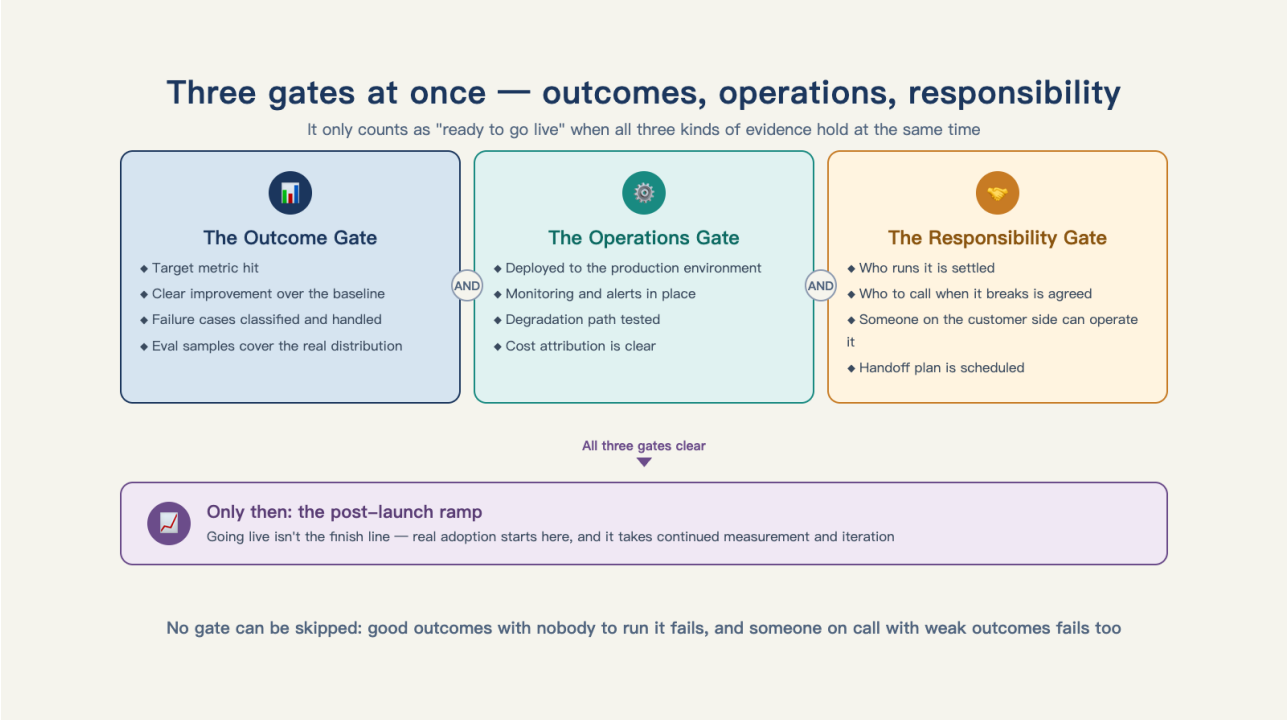

Figure 8–1: Going live means clearing three gates at once — outcomes, operations, and responsibility — and ramp-up validation still follows.

Of the five elements, the denominator definition is where the landmines hide most often — the same "90% resolution rate" can mean the denominator is every conversation, or only conversations routed to the AI channel, or only conversations the AI actually attempted to answer. Those three denominators produce numbers a full tier apart.

This is exactly where metric-definition discipline lives. The core value metrics for enterprise deployments — resolution rate, deflection rate — are precisely the metrics where every vendor's definition is the messiest. Take ServiceNow's official definition as an example: a "deflection" only counts if two conditions hold at once — no ticket filed within 24 hours of the interaction, and the user subsequently engaged positively (continued using the product, for instance). So before quoting any vendor's resolution rate or deflection rate, ask three things first: **what's the denominator, how long is the judgment window, and who's the judge.** One line nails this discipline exactly: "an 86% with a murky definition is worth less than a 51% with a clean one." [1]

What does a metric with all five elements spelled out actually look like? The insurance-industry company mentioned earlier gives a good template: the quality standard they agreed with the customer was 99.5% — translated into something checkable, that's "getting it right 995 times out of 1000 every week" [2] . What makes this number good comes down to three things: **it's countable** (995 out of 1000), **it has a window** (every week), and **it can be double-checked** (the customer can count it themselves). Countable, windowed, double-checkable — every metric on an acceptance sheet should survive all three questions.

The acceptance sheet has a negotiating function too: freeze the metric ahead of time, and go-live day loses the "this month's data is a special case" excuse. Every objection to the definition gets pushed back to the cheapest possible moment to raise it — before anything has even started running, when fixing it means changing one line of text.

III. The Operational Readiness Checklist: Who's Watching When the Alert Fires

Hitting the outcome target only answers "is it worth going live." The second document answers "do we dare go live." Four questions, one at a time:

Is anyone watching the alerts? When an alert fires, does it map to a name and an on-call schedule — not fall into a junk folder, not sink into a group chat. How the monitoring system is

[1] Fan Bing, Forward Deployed Engineer (FDE) (XDash open-source book)

[2] Inside an Applied AI company (Pace long-form post)

built is a later engineering topic; here it's only the acceptance-level question: when the alert fires, who sees it?

Is there a working degradation path, and has it been rehearsed? Including the plainest version: **an explicit fallback to Excel and manual work.** A lot of teams feel like writing "fall back to manual" into the plan is embarrassing, as if putting it on paper is admitting AI doesn't work. It's exactly the opposite: a plan willing to write down its own rollback is the plan that dares to go live — it admits the system will break, and it's already agreed in advance what happens when it does. This point gets repeated constantly in China: it's the item on the acceptance checklist most often skipped, and the one people are most grateful for when something actually breaks. A degradation path isn't a fig leaf for failure — it's a safety net, built on purpose.

Does a runbook exist, and has anyone on the customer side actually read it? The incident-response steps have to be written down, rehearsed once, and the customer's own point of contact has to hold a copy and have read it. A runbook that only lives on the vendor's laptop is the same as no runbook at all.

Who's on the on-call rotation? For the first few weeks after going live, who on the vendor's side is on call, and what's the committed response time — write it down in black and white. It's the least conspicuous item across all three documents, and the first thing anyone asks when something breaks.

The cautionary tale is short but it stings. An FDE at an AI startup noted in a weekly diary: an upstream provider (Cloudflare) went down, there was no contingency plan, and firefighting had to happen live — every one of their customers' systems felt it at once [3] . An upstream outage was never within your control. "No contingency plan" was. That's exactly what the operational readiness checklist is there to catch.

IV. The Responsibility Handoff Table: Every Way to Fail Gets a Name

The third document is the one most often missing, and the one that most tests both sides' good faith. The method is to list out every way the system could fail, one by one, and write down the **first responder** and the **escalation path** (how long, escalating to whom) for each. Here's an example:

[3] Hacker News, "A Week in the Life of a Forward Deployed Engineer" (10 customers, 50 hours)

What Went Wrong	First Responsible Party	Escalation Path
Model output error causing business loss	Who judges it, who blocks it, who pays for it	To both sides' leads within 1 hour
Data source failure / upstream outage	The data-side point of contact	To vendor on-call within 4 hours
Incident caused by user error	Customer-side system administrator	To a joint post-mortem, same day
Cost overrun	The platform provider	To the project lead, in the weekly report

Why is this table worth drawing at all? Back to the line that opened this chapter. When Frost wrote "good fences make good neighbours," he wasn't talking about suspicion — he was talking about **drawing the boundary before anything goes wrong**. Put the fence up in advance, and the neighbors stay friends. Wait until the sheep have trampled the garden to draw the line, and no matter how you draw it, it damages the relationship. A responsibility handoff table is the fence you put up before going live: whose fault it is when the model breaks, whose fault it is when the data breaks, whose fault it is when someone fat-fingers an action — write it down in black and white, and when something breaks, you find the right person off the list without burning the relationship. It's the front-end slice of the full handoff covered in Chapter 17 — what's handed off at go-live is "first response"; long-term operational responsibility is a bigger table of its own.

With all three documents in hand, run them past the review table and you'll find out exactly how solid they are. The following review-meeting minutes aren't a real case, but every item in them comes straight off the checklists above:

> **Go-Live Review Minutes (illustrative simulation)**
> Passed: all five elements of the acceptance sheet are complete; the "who decides" field names the actual business owner, not "the business side" in the abstract; the rollback line for phase three of the gradual rollout is quantified.
> Stuck: the degradation path — "fall back to manual" — is written down, but has never been rehearsed. A rehearsal date got scheduled on the spot before sign-off proceeded. The on-call roster only covers the first two weeks after go-live; after that, it's blank.
> Disputed: responsibility for incidents caused by user error. The vendor argues it's entirely the customer-side administrator's fault; the customer argues the interface should have been designed to prevent the error in the first place. After a standoff, it went in as "shared responsibility, plus a joint post-mortem rule."
> One-third passed, one-third stuck, one-third disputed — that's roughly what a real review meeting's minutes look like.

One last look at the two ways this degrades when responsibility is left hanging — both happen

exactly where nobody drafted a contract. One is "**run it free for three months first**" — responsibility has no owner; everyone's happy while it runs well, and the relationship blows up the first time something breaks, because no page anywhere ever said whose fault it was. The other is "**the results are great, just keep using it**" — the vendor is effectively refusing the handoff, and the system stays permanently stuck at "trial," never growing into a production system. The two degrade in opposite directions, but the root cause is the same: an absent contract.

V. Both Sides Have Homework to Do

This contract shouldn't wait to be remembered until the go-live review. Both sides have homework due earlier than that.

The customer's self-check list. Flip the vendor's own customer-screening filter over, and it becomes the customer's self-check list: is your project, internally, the kind of "no-man's-land" a vendor dreads — pushed down from a group-level directive, led by an IT department, with no business department willing to put its name on it? Does your procurement process structurally force "look, but don't touch" — demanding the vendor prove its capability first while handing over exactly zero real data? Does what you wrote into the RFP amount to a universe-scale demand — "AI-empower the entire group"? The verdict is straightforward: that last kind of requirement scares off exactly the vendors who know what they're doing, and attracts exactly the ones bold enough to bluff. If any one of these three is a "yes," fix your own house before you talk about going live.

The vendor's homework. A good proposal is written backward: the first layer has to be the **business outcome**, stated in one paragraph, in the customer's own language — "cut average anti-money-laundering investigation time from 4 hours to 15 minutes within eight weeks." The second layer is the **value-validation path** — acceptance metrics, measurement method, baseline data, an exit mechanism for each phase — and the signal it sends is "we're not afraid to be checked." Only the third layer is the **delivery method**, spelling out what the customer needs to contribute. Last comes **risk and countermeasures** — "here's how we've thought this could die" [1] . That second layer is the shadow this chapter's three-piece set casts back into the pre-sales stage: the acceptance sheet isn't homework you make up at go-live — it's a commitment planted on page one of the proposal. This playbook has a precedent a full decade older than the AI era: a data-analytics company has done heavyweight pre-sales implementation

[1] Fan Bing, Forward Deployed Engineer (FDE) (XDash open-source book)

during free trials since 2013, on exactly one principle — **the demo is the proof of concept.** Always ask a prospective customer for a real dataset. Never perform on a carefully prepared sample.

The reality at the negotiating table is: when the customer side doesn't understand metrics, they tend to fall back on one of two postures — "we need 100% accuracy" (which turns the acceptance sheet into an impossible task) or "just run it free for three months" (which leaves responsibility hanging). The fix for both postures is the same sentence: turn "results" into a measurable number. This is also the vendor's own sales logic — **trade the acceptance sheet for trust**: the more clearly the acceptance terms are written, the more confidently the customer signs, and the faster the deal closes. In an enterprise market that's been burned by over-promising before, the willingness to be checked is the scarcest form of sincerity there is.

VI. Going Live Is a Ramp, Not a Switch

The contract's signed — but going live still isn't a switch you flip. It's more like climbing a ramp: release traffic or user groups in stages, and every stage has a **rollback line** — if the metric falls below the agreed value, drop back a stage, find out why, then try again. Go-live day isn't the finish line. It's the first step of the ramp.

Buried inside the length of that ramp is exactly the stretch of time most schedules leave out. A bank-project post-mortem circulating in the Chinese community gives a strikingly lopsided pair of numbers: the technical work was done in six to eight weeks, and the pilot phase plus building trust took another four months [4] . The first half is progress the vendor controls; the second half is time the customer's organization needs — and most schedules only draw the first half. So the full answer to "when can this go live" has to set aside an **explicit** time budget for building trust: the stretch it takes front-line users to go from "afraid to use it" to "can't work without it" is exactly as real as development time, and how to earn it is the subject of the chapter on adoption.

Closing out this stretch of the book's road: an open-source methodology document once marked reference paces for this kind of deployment — the Minimum Viable Deployment validated in weeks (2—6 weeks), the full deployment measured in months (1—4 months), and one warning attached — **time-to-value that keeps getting longer is the first sign that the methodology or the platform itself is broken** [1] . Check it against the three-piece set: the

[1] Fan Bing, Forward Deployed Engineer (FDE) (XDash open-source book)

[4] "FDE Engineer 101: Bridging the Gap Between AI and Business" (Chinese-language compilation video)

acceptance sheet is signed, the readiness checklist has passed review, the responsibility table is up on the wall, and the ramp has climbed its last stage — only then can the words "ready to go live" actually be said. And what those words really announce isn't that the system succeeded. It's that the contract just took effect.

But the contract taking effect is only the ticket to get in the door. Turning "a validation that runs" into "a production system you can actually trust" still means getting past eight walls — data, permissions, evaluation, approval, cost... Every one of them is real, and every one of them can be climbed. That's what Part Three is about.

Part Three · Engineering Foundations

> This part has six chapters, and together they answer one question: **what does it actually take, in engineering terms, to turn "a proof that runs" into "a production system worth trusting"?**

This is where the line Part Two ended on gets cashed in: the acceptance contract is signed, and plenty of obstacles are still standing.

Chapter 9 opens by breaking "going to production" into eight walls, each definable and each subject to acceptance — and makes one recommendation: count the walls before you quote a price and a schedule. A quote that hasn't counted the walls is a landmine buried at the contract stage.

Then the book tackles the walls one by one: how to wire up data and permissions (Chapter 10, Walls One and Two); how to constrain what an agent can do so it's safe to put in production (Chapter 11, Walls Three and Four); how to prove results haven't regressed (Chapter 12, Wall Five); how to earn the customer's trust in the results (Chapter 13, Wall Six); how to keep the bill and the system's stability under control (Chapter 14, Wall Seven). Wall Eight — transferable responsibility — gets its full payoff in Chapter 17.

The six chapters form one chain: only once the data is wired up can an agent actually act; only once its actions are governed does it make sense to talk about proving results; only once results are provable does it make sense to talk about trust; only once results are trusted do money and stability still need holding down — loosen one link, and the whole chain goes slack.

The primary reader for this part is **whoever is on the hook for production**: the technical lead for delivery, the engineer leading the on-site team, the customer-side technical representative who has to sign the acceptance form. The eight walls get translated into engineering language, but every one of them starts with something a customer actually says — "the data can't leave our domain," "whose fault is it if this runs wild at 2 a.m.," "where does this number come from," "why is the bill this big." Organization and adoption are Part Four's business; pricing and the business model are Part Five's. This part answers exactly one question: what makes this system worth trusting with the job.

Chapter 9 · How Many Walls Stand Between PoC and Production?

> The general who wins a battle makes many calculations in his temple before the battle is fought. The general who loses a battle makes but few calculations beforehand.
> — Sun Tzu, The Art of War, "Laying Plans"

"The PoC ran in a week — why does going to production still take another six months?" Every customer asks this sooner or later, and most FDEs fumble the answer.

The answer isn't actually hard. What holds up a launch was never the amount of code.

I. Between PoC and Production Stand a Lot of Walls

Start by taking apart the word "productionize." Different people mean completely different things by it — one means "add monitoring," another means "add more machines," a third means "pass the security audit." None of them is wrong. Each is just a piece.

A system that can actually do work inside a real enterprise splits into three layers: the **model**, the **interface**, and the **harness**. The model can reason, but it can't act — the capacity to act comes entirely from the harness. In one insurance company's deployment, the harness comes down to five things: **tools** (the model can't calculate premiums reliably, so give it a calculator; it can't read a handwritten form, so give it a recognition router); **integration** (wiring into each insurer's wildly different API — and where there's no API, driving the browser programmatically instead); **a file system** (a process that runs for days needs its state stored somewhere, so an interruption doesn't mean starting over); **task orchestration** (route the easy classification work to a cheap model, the ambiguous clauses to a stronger one, with a task graph deciding what runs when); and a **supervision mechanism** (low-confidence outputs go to a human review queue, irreversible actions wait for human approval, and everything leaves an audit trail) [1] .

The value in this split is honesty: a company on the front lines gave a name to "everything that sits between the model and production," and admitted it's half the product, not some cleanup

[1] Inside an Applied AI company (Pace long-form post)

[2] enterprise_agent_platform (open-source enterprise agent platform engineering project)

task tacked on at the end. An open-source book on enterprise agent platform engineering breaks it down further still — it lists a dozen-plus capabilities a production-grade deployment has to deliver, from the runtime and tool contracts to evaluation and cost governance, each paired with a minimal code reference [2] .

This book follows the same direction, breaking down what stands between a model and production into **eight walls** — translating "productionize" into a checklist you can sign off item by item. All eight walls have to come down before delivery is complete.

II. The Eight Walls

Wall One, data and permissions. The question on the ground: which tables can be read, who can see the results, how does masking work, what's the process for exceptions? The customer's security review isn't a formality — there's an actual form to fill out. What gets you over the wall: a Data Contract. That's the subject of Chapter 10.

Wall Two, integration. The question on the ground: the system the agent needs to write back to is one the customer has been running for twenty years — no API, and nobody can quite describe the data format anymore. Box's CEO put it bluntly: integration is the wall, and an agent won't fix your integration problems for you — it'll just run straight into them [3] . What gets you over the wall: a path that actually connects the old system to the new.

Wall Three, tool boundaries. The question on the ground: which tools can the agent call, and who validates the parameters? An agent that can send an email on its own and one that can only draft an email sit at two completely different risk levels. What gets you over the wall: a Tool Contract. That's the subject of Chapter 11.

Wall Four, controllable execution. The question on the ground: if a job runs wild at 2 a.m., whose fault is it? If it's stuck waiting on human approval, can it pick back up the next day? What gets you over the wall: a Run State Machine with checkpoints. Also covered in Chapter 11.

Wall Five, provable results. The question on the ground: after a prompt change, did quality go up or down? If you can't answer that, every change is a gamble. What gets you over the wall: an eval set and a regression gate. That's the subject of Chapter 12.

Wall Six, trustworthy results. The question on the ground: the customer points at the screen and asks "where did this number come from, why should I believe this conclusion" — what can you produce besides an apology and a rerun? What gets you over the wall: a chain of evidence.

[3] a16z: Box CEO on AI agents and why enterprises can't keep up (YouTube)

That's the subject of Chapter 13.

Wall Seven, cost and stability. The question on the ground: a call cost pennies in the demo, and the monthly bill hits five figures after launch; the system worked fine yesterday and is glitching today. What gets you over the wall: cost attribution and a stability commitment. That's the subject of Chapter 14.

Wall Eight, transferable responsibility. The question on the ground: who's responsible when something goes wrong, and once the FDE leaves, who does the system get handed to? What gets you over the wall: a responsibility handoff table and a runbook — Chapter 8 already sketched an early version of that table; Chapter 17 unpacks the full handoff.

A concrete example. Jove Zhong, who heads AI Agent FDE at Cresta, has described just how complex a voice AI agent can get: as many as twenty models can be running behind the scenes at once — ASR, interruption detection, noise isolation, retrieval, tool calls, guardrails spanning multiple concurrent models — on top of compliance audits (PCI for payment cards, HIPAA for U.S. medical privacy) that routinely take six months to a year. Getting it running end to end takes a week; writing the thousands, sometimes tens of thousands, of tests takes a month after that — and these aren't traditional unit tests, but small models trained on historical call data to actively simulate the edge cases a real deployment will hit (more on this technique in Chapter 12) [4] .

This example puts a concrete face on both Wall Four (controllable execution — which of twenty models runs when, which one gets to interrupt which) and Wall Five (provable results — where thousands of tests come from) at the same time: the excitement of "end to end in a week" during the PoC stage and the reality of "a month of testing" afterward are two sides of the same project.

Looked at together, the eight walls aren't all the same kind of thing: Walls One and Two are the **foundation** — without them, nothing else is worth discussing. Walls Three and Four are **risk** — without them, the system can still run, but something can go wrong at any moment. Walls Five through Eight are **trust** — without them, the system stays stuck at "trial" forever. Different natures call for different fixes, and different acceptance criteria (see Figure 9–1).

[4] Tencent Research Institute, AI Lens Roundtable, Episode 6: "FDE Non-Consensus Views and a Field Guide from Silicon Valley Founders"

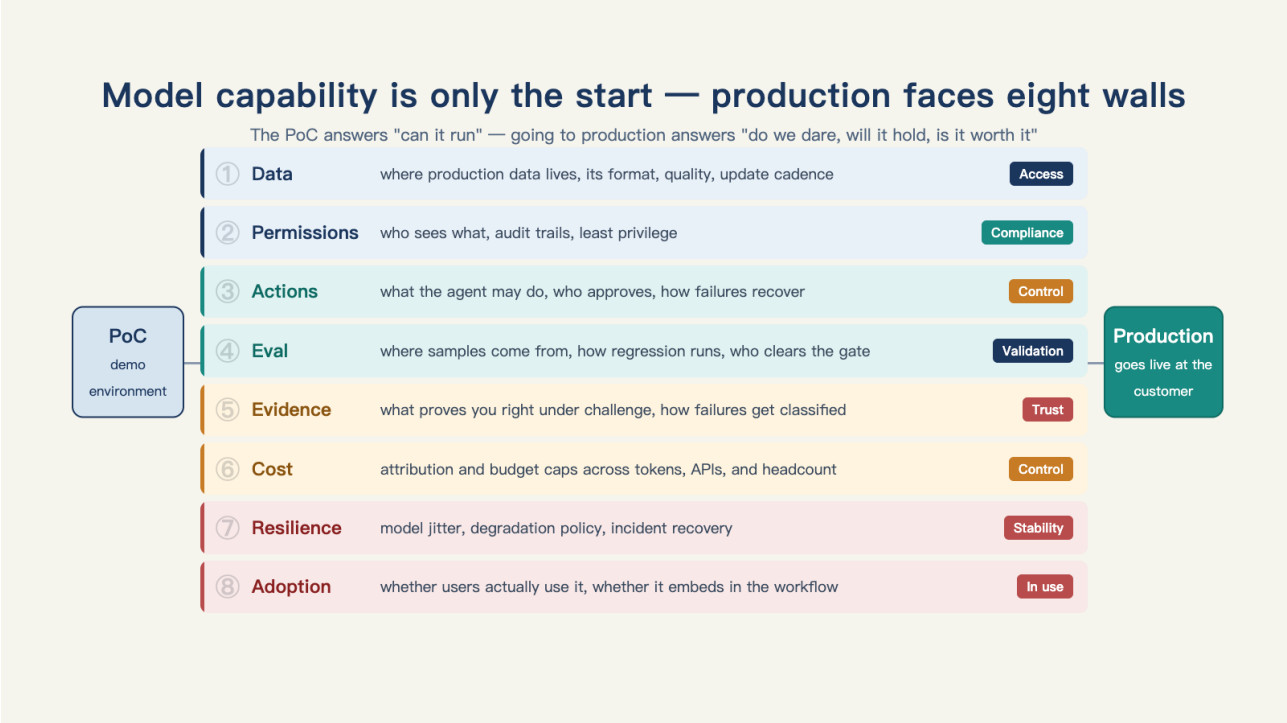

Figure 9–1: Model capability is only the starting point — a production system still has to pass through eight kinds of engineering and organizational constraints.

III. Map the Walls When You Sign the Contract — Not After You've Already Hit Them

Why map the walls out? Because the engineering effort behind each one has to go into the quote and the schedule.

Two different approaches to the contract lead to two completely different endings.

In a contract that has mapped the walls, every wall comes with an effort estimate, an owner, and acceptance criteria — behind the quote the customer sees is a map, both sides move forward by milestone, and hitting a wall just means it was on the plan all along.

In a contract that hasn't, "AI capability delivery" gets bundled into one line item — the applause from the demo is still ringing, and delivery starts running into walls one after another. Every wall becomes "an additional requirement," every wall means renegotiating money and schedule from scratch, and three months of wrangling later the project quietly dies. Same project — call the first outcome a delivery and the second a write-off. The fork in the road is only whether the walls got mapped on the day the quote was written.

There's another cost to not mapping the walls, and it lands on the vendor's own books.

[5] @deployengineer (Bhaulik Patel), essay series

A delivery lead with hands-on industry experience has warned about this: take a $2 million contract that hasn't priced in the engineering effort behind the walls, staff it with six engineers for twelve months, and the gross margin can come out negative — profitable on paper, eroding enterprise value in reality [5] . The full arithmetic behind that belongs to Part Five; for now, here's the warning worth flagging: **failing to map the walls is a direct cause of the losses that follow.**

The popular claim that "AI projects have a high failure rate" sounds like a verdict on the technology. Take "failure" apart and most stalled projects die of the same cause: at quoting time, the walls never made it into the contract, and never made it into the price. A wall that never got priced doesn't stop existing — it just surfaces later, at the worst possible moment: mid-delivery, under the customer's gaze. The technology doesn't deserve the blame.

IV. As Models Get Stronger, Do the Walls Disappear on Their Own?

A natural question: as each model generation gets stronger, will these eight walls just disappear?

Half of them shrink. The first four walls — data access, integration, tool generation, run orchestration — are engineering work that code can solve, and the stronger the model, the faster that work gets done. Some have gone even further and turned FDE work itself into a product: an agent that takes over the repetitive labor behind the first four walls [6] — pointed in the same direction, just packaged differently.

But the last four walls are hard to shrink.

Ali Ghodsi, Databricks' co-founder and CEO, gave an example in a guest lecture at Stanford that draws this line clearly: building a production-ready application connector used to take about nine months on their old process — the first quarter alone went to requirements gathering and dozens of pages of documentation, and only then came development, staging, security verification, and customer feedback. The team lead wrote a working version himself with AI in two days, but the team agreed it was still just a demo — it hadn't gone through full testing and couldn't be guaranteed safe and stable for a customer. The more sobering estimate: even bolting AI straight onto the old process only shrank the cycle from nine months to about seven and a half — the time saved got eaten right back up by the process itself. The real turning point wasn't a stronger model. It was rewriting the process: requirements went from a full quarter down to a week, staging environments were handed to a team better suited to building them in

[6] Palantir Foundry, "AI FDE" product documentation

[7] Ali Ghodsi's guest lecture in Stanford's MS&E 435 course

parallel, and the way people worked changed from "one person owns one connector" to "a team collectively covers a group of connectors." After the rebuild, they shipped seven production-grade connectors in a single quarter [7] .

The AI that wrote a demo in two days ate into the engineering effort behind the first four walls. Nine months becoming seven and a half — when only the model changes and the process doesn't — shows that the last four walls won't give an inch. Seven connectors in one quarter shows that once the last four walls come down, the speedup from the first four finally converts into productivity. **Testing, security, how people collaborate, how responsibility is divided — the height of these walls is set by organization and trust, not by model capability.** This is the engineering version of the judgment from Chapter 5: the reason an FDE won't be replaced by AI is hiding inside these last four walls.

V. Take This Table With You

This chapter doesn't hand you the specific method for taking down each wall — that's the job of the next five chapters. What it hands you is a reference table: take any project that's "passed the PoC, awaiting production" and ask three questions about it — **has this wall come down? What's the deliverable? Who signs off?**

Wall	The question on the ground	What gets you over it	Full treatment
One — Data and permissions	Which tables can be read, who can see the results	Data Contract	Chapter 10
Two — Integration	How a new system writes back to a twenty-year-old one	A connected integration path	Chapter 10
Three — Tool boundaries	What the agent can call, who validates the parameters	Tool Contract	Chapter 11
Four — Controllable execution	Whose fault if it runs wild, can it resume once stuck	Run State Machine and checkpoints	Chapter 11
Five — Provable results	Did a prompt change make it better or worse	Eval set and regression gate	Chapter 12
Six — Trustworthy results	What answers "where did this number come from"	Chain of evidence	Chapter 13
Seven — Cost and stability	What to do about a runaway bill or a jittery system	Cost attribution and a stability commitment	Chapter 14
Eight — Transferable responsibility	Who's responsible when it breaks, who inherits it after the FDE leaves	Responsibility handoff table and runbook	Chapter 17

If any one of the three questions goes unanswered, that wall is a hole in the schedule — and the

most expensive kind of hole. The full version of this table is Appendix A: the delivery checklist, organized across four stages — discovery, minimum viable deployment, productionization, and launch and adoption. This chapter's eight walls sit concentrated in the "productionization" stretch, grouped into foundation, risk, and trust.

Starting next chapter, the walls come down one by one. First, the one you hit earliest: the customer's data and permissions.

Chapter 10 · How to Wire Up Data and Permissions — Usable Without Losing Control

> Trust, but verify.
> — Russian proverb, popularized by U.S. President Ronald Reagan, who invoked it repeatedly at the 1987 signing of the INF Treaty

On an FDE's first day on-site, the first question for the customer is usually "where's the data?" The most common answer: "it's all in the system."

In real AI deployments for new customers, merging seven or eight scattered data sources is routine — email, spreadsheets, CRM, ERP, structured and unstructured data all mixed together. Companies that have been rolled up through private equity acquisitions are the extreme case — sometimes running four ERP systems at once, and the customer itself can't say what its own data actually looks like [1] . So the accurate translation of "it's all in the system" is: nobody knows where it is yet.

The pitfalls of data access start with that very first question — and something more dangerous than "where" is "who can see it."

I. On Day One, Ask Five Questions First

Turning data access into schedulable work starts with five questions: **Where is the data? Who owns it? Who can see it? How does masking work? What's the process for exceptions?**

None of the five can be skipped. Miss "who owns it" and permission requests land on the wrong desk — one round trip alone costs a week. Miss "who can see it" and the agent's output can leak straight to someone who shouldn't see it. Miss "the exception process" and the first time data needs to cross a boundary, work stops while everyone waits for an executive to make the call.

Any project that can't answer these five questions gets two weeks added to the schedule,

[1] Silicon Valley 101, Episode 240: "The Hottest New Job in Silicon Valley — FDE" (YouTube)

automatically. That's not a penalty — it's just realism: pulling data together is usually the first wall a team hits after arriving on-site, and it gets hit before any permission design even begins [1] . Chapter 6 already asked "can the data actually be obtained" once, during discovery — this section is that question's sequel during the engineering phase. Discovery asks whether the data exists at all; here the questions are how to get it, how much to get, and who gets to see it.

Once you've mapped out where the data lives, there's another reality to accept: the system the agent needs to write back to is very likely one the customer has run for over a decade. Chapter 7 already gave the fix — read from the old, write to the new: read data from wherever it already lives, interact through an interface, and never wedge code into the legacy system.

II. Contract First, Data Second

Once the five questions are answered, the answers can't just sit in the FDE's notebook — they need to be written up as a Data Contract before any data gets touched (see Figure 10-1).

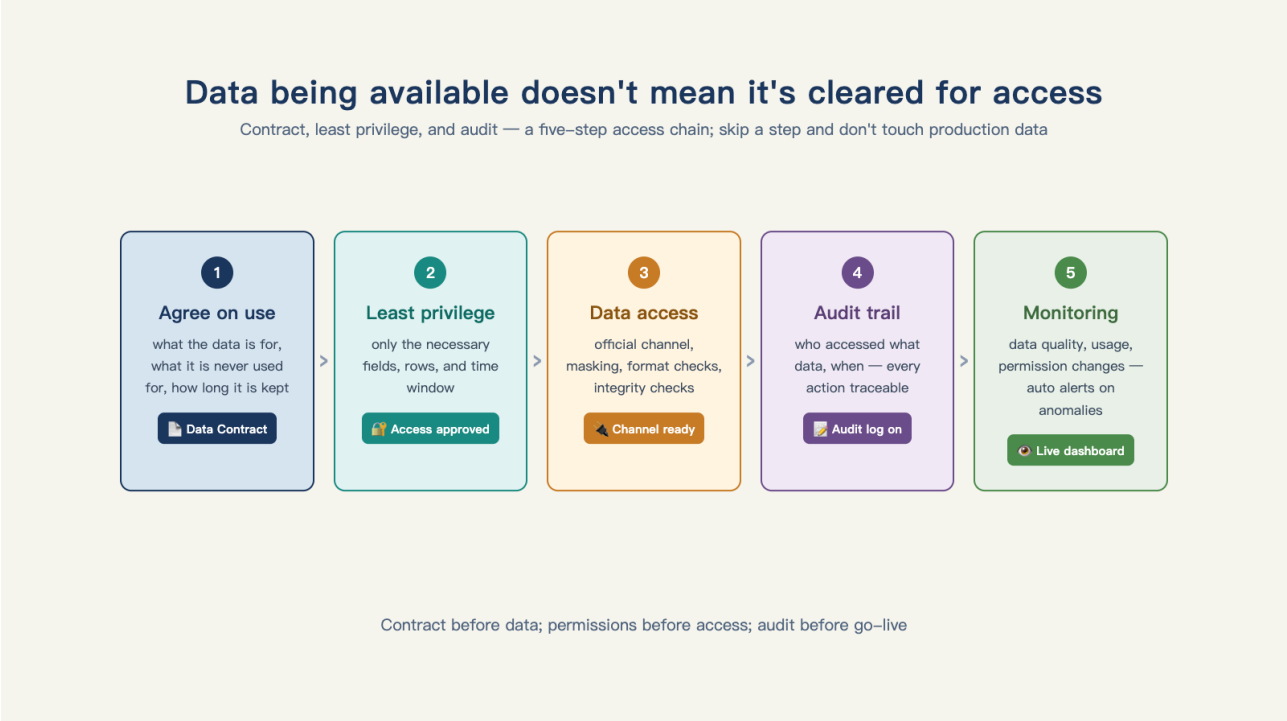

Figure 10-1: Data being available doesn't mean it's cleared for access — the contract, least privilege, and audit trail are all required.

The contract spells out four things: **access scope** (which tables, which fields, precise down to the column); **result visibility** (who can see what the agent produces, and can it leave the

[2] enterprise_agent_platform (open-source enterprise agent platform engineering project)

building); **retention and destruction** (how long the vendor holds the customer's data, and how it gets destroyed once the project ends); and **audit requirements** (who reviews the access logs, and how often) [2] .

This contract has three users. **The customer's compliance review** uses it: the security department's sign-off is looking at exactly this piece of paper, and without it the project can't clear the process. **The FDE uses it to constrain their own implementation**: if the contract says read-only access to three tables, the code only gets connections to three tables — this is a boundary for the engineer, not a posture for the customer. **Post-incident accountability relies on it**: if a real data incident happens, both sides sort out responsibility against the contract's terms, instead of pointing fingers at each other.

Why can't the order be reversed? Because a PoC without a contract is running naked through the customer's data warehouse — if something goes wrong, there's no basis for accountability, and even if nothing goes wrong, it still won't clear the customer's security review. A lot of people think "get it running first, write the contract later" counts as flexibility. It's actually just deferring the engineering effort behind two walls — data and permissions — to the most expensive possible moment.

III. An Agent's Permissions Are Inherited, Not Assigned

The contract governs data; permissions govern action. And here's the most common mistake in the industry: requesting a broad-permission service account for the agent — "convenient, set it up once and done."

The correct approach is this: **the agent inherits the permissions of the user who triggered it, with a full trail left behind.**

There's already a product-level example of this. Palantir turned its own AI FDE into a product, and the documentation states plainly that every action the agent takes follows the user's existing permissions — whatever the user can't see, the agent dispatched on their behalf can't see either [3] .

Box's CEO supplies the other half of this real-world constraint: an agent only holds the human's existing permissions, and it won't route around the permission wall — being stuck on permissions just means being stuck, and that doesn't change no matter how much stronger the

[3] Palantir Foundry, "AI FDE" product documentation

[4] a16z: Box CEO on AI agents and why enterprises can't keep up (YouTube)

model gets [4] .

Put the two together and you get one conclusion: an agent's permissions equal the triggering user's permissions, plus a full trail — never request superset permissions at the service-account level. The problem with superset permissions isn't that they "might be abused" — it's that they can never be sorted out. On the day something goes wrong, when the audit log shows an action from that all-powerful account, you can't prove it was a human, and you can't prove it was the agent.

IV. Permission Configuration Is Something You Do Together with the Customer's IT Team

In engineering terms, permissions don't just gate data reads — they also have to govern **the consequences of an action**: who can trigger an outbound send, who can write to a production table, who can run privileged actions in production. Front-line teams control tool calls across three layers — permission, role, and scenario: the same person can do different things in staging than in production; the same tool carries different permissions for reading data versus writing it [5] .

How this actually gets done deserves its own callout: a junior FDE at a Japanese AI company documented their own configuration experience — working item by item alongside the customer's administrator to configure production permissions for M365 and Entra ID [6] . Notice the picture here: **the FDE and the customer's IT sitting side by side in front of the same screen.** Permission configuration isn't something a vendor delivers unilaterally — it's work both sides do together, because every boundary permissions draw takes away some convenience from a real person inside the customer's organization.

This also answers a common complaint: "the customer's IT won't cooperate." Flip the perspective: if an outside team wanted to punch a hole in your own permission system, your first instinct would be to scrutinize it too. Turn permission configuration into shared work, and the customer's IT team shifts from gatekeeper to partner — every change after that goes a lot faster.

V. Data Sensitivity Determines the Deployment Model

[5] "AI Job Sense: Stop the Agent Rat Race"

[6] "I Thought FDE Meant Fighting Alone at the Customer Site" — What Two Months on the Job Actually Looks Like (note.com · AI Shift)

There's one last decision in data access: where does the model run — a cloud API, a private deployment, or fully on-premises?

This decision isn't a matter of technical preference. It's set by the data's sensitivity level.

Split the data into four tiers: **public** (marketing materials, published reports); **internal** (process documentation, non-sensitive operating data); **sensitive** (customer personal information, transaction records); and **regulated** (data under industry regulation, such as core medical or financial records). Each tier maps to a different deployment model: public and internal data can go through a cloud API; sensitive data leans toward private or on-premises deployment; regulated data can often only run on-premises.

One front-line lesson walked this whole path start to finish: a customer-service agent project sailed through its first two versions, and only when it came time to run experiments on real customer data did a data-security problem surface — budget constraints left the team no choice but to deploy a small model on-premises, and quality took a hit as a result [5] . What makes this case valuable is its honesty: **data sensitivity determines the architecture, not the other way around.** When sensitive data forces the deployment down to on-premises, both the model choice and the ceiling on results get constrained right along with it — that's not an engineering preference, it's a constraint propagating downstream.

So sensitivity tiering needs to happen back in discovery — the fourth filter in Chapter 6 (data reachability), which asks "what data is needed, who owns the permissions," already includes this question. Get sensitivity mapped out during discovery, and there's room in the quote to budget for it; miss it, and "suddenly we need on-premises deployment" becomes a rework during the production phase.

VI. Every Access Leaves Evidence Behind

The last piece of the puzzle: the audit log.

Every data access — who initiated it, which table it read, when, for which task — gets recorded as a log entry. Its value doesn't show up on an ordinary day; it's the only answer available the moment a compliance inquiry lands. When the customer's security team asks "exactly what data has your agent touched," two kinds of answers carry completely different weight: one is "we've never had an incident," the other is exporting the log — every access documented and traceable.

The first is a promise; the second is evidence. Enterprise trust doesn't run on promises — it only

[5] "AI Job Sense: Stop the Agent Rat Race"

runs on evidence. This is Chapter 13's theme (trustworthy results), making an early appearance here at the data layer: **logging is an architectural decision, not something written up after the fact.** The audit log runs from day one — that's the engineering payoff of the "audit requirements" line in the Data Contract.

The signal that the data-and-permissions wall has come down can be written up as an acceptance sheet: the Data Contract is signed, the permission-inheritance model works, joint configuration with the customer's IT is done, sensitivity tiering has settled the deployment model, and the audit log is running. Only once all five are in place does it make sense to talk about the next thing: letting the agent actually get to work. And the moment it can act, the next question shows up immediately — where exactly is the boundary on each of its actions?

Chapter 11 · How to Constrain What an Agent Can Do — Enough to Trust It in Production

> You must first enable the government to control the governed; and in the next place oblige it to control itself.
> — James Madison, The Federalist Papers, No. 51, 1788

"We have a human review step." This line comes up in every presales meeting, but it's worth pushing back with three questions: is what gets approved the **action** or the **result**? Who's the approver, and **is there a record**? After a rejection, does the process **rerun from scratch**? If those three questions go unanswered, "human review" is just a slogan.

Turning the slogan into engineering is this chapter's job — and it's also the line that separates a demo agent from a production one.

I. Start by Making the Customer's Worries Concrete

A customer's IT team doesn't worry about the agent in some vague "it's not safe" way — every worry is specific:

"Will it pass garbage parameters and blow a hole through the production system?" — if nobody validates the model-generated parameters, one misspelled table name can corrupt data.

"Will it go delete data or send emails on its own?" — an agent with delete and outbound-send permissions turns a single hallucination into a real incident.

"If it runs off the rails at 2 a.m., who notices?" — a scheduled job gets stuck in a loop at 2 a.m., and by 9 a.m. when people show up for work, it's already run a few thousand times.

"If something goes wrong, can we trace which step it was?" — a black box that just spits out a result at the end gives you nothing to localize a failure to; all you can do is start over from scratch.

Notice: what they care about isn't "is the model smart enough." They're asking whether **actions have boundaries, whether the process leaves a record, whether there's a way out when**

something goes wrong. Four worries map to four mechanisms — the four sections of this chapter.

An agent you're willing to put into production has to pass four tests on every single action: **validatable, pausable, resumable, accountable.** These mechanisms already have mature, field-level designs in the open-source engineering community [1] ; what follows here is why each one is needed and what it actually looks like.

II. Validatable: If the Parameters Don't Pass, the Action Doesn't Leave the Gate

The first gate sits at the entrance to the tool.

Every time the agent calls a tool, the parameters have to pass a **structural validation**: what's the tool called, which version, what types of parameters does it accept, what's the legal range for each — all written up as an explicit contract. Every call is checked against the contract first; it only executes on a match, and gets rejected with an error otherwise.

Take the smallest possible example: a "query order" tool whose contract accepts exactly two parameters — an order ID (string, fixed format) and a query reason (an enum with three options). If the model's generated call slips in an extra "customer phone number," validation fails and the action doesn't execute. This one gate alone blocks the most subtle kind of overreach — the model quietly grabbing a bit of extra data along the way.

Every tool also needs its **call boundary written down explicitly**: read-only, write, or outbound — pick one and state it. If a read-only tool breaks, the worst case is a bad read; if a write tool breaks, it changes state; if an outbound tool breaks, it can send data across the boundary — three completely different risk levels. Declaring this up front is what gives the tiered controls in Section V something to work from.

When self-checking on-site, ask one question: "at which step does parameter validation happen for each of your tools?" If the answer is vague, this gate is fake.

III. Pausable and Accountable: Approval Is a Ticket, Not a Pop-Up

High-risk actions have to wait for human approval — "add a human review step" is a phrase

[1] enterprise_agent_platform (open-source enterprise agent platform engineering project)

everyone knows how to say. Where's the engineering? It's in modeling approval as a **ticket with fields**, not a confirmation box that pops up on a screen.

A proper approval request carries at least five fields: **which run it belongs to** (never approve an action in isolation — you need to know what comes before and after it); **which step it's stuck on** (where the whole process currently stands); **what it wants to do** (the specific action plus parameters, at a glance); **who approves it** (a named person, not just "the business side"); and **what the annotation says** (the reasoning behind approval or rejection, kept on file).

The difference between a pop-up and a ticket is the difference between a slogan and engineering. Close the pop-up and it's gone; a ticket is a **record that can be archived and audited** — six months later, when the customer asks "who signed off on that outbound send," pull up the ticket: who, when, on what grounds, approved what. Four questions answered on one sheet of paper. This also sets up the handoff later: on the day the customer takes over running the system, the "who approves" field on the approval ticket is exactly where responsibility transfers (more in Chapter 17).

IV. Resumable: A Night's Delay on Approval Doesn't Mean Starting Over

Approval has one built-in property: it's slow. The approver is in a meeting, on vacation, in a different time zone — waiting overnight is completely normal.

What happens if the process can't resume? The morning after approval finally comes through, the system reruns from the beginning — the dozen-odd steps already completed, the API costs already spent, the hours already consumed, all repeated. Worse still is the behavioral distortion: a business side that can't afford to wait starts looking for ways around approval — either forcing the action to be reclassified as approval-free, or simply abandoning the process altogether. **An approval nobody can wait for is the same as no approval at all.**

The fix is checkpointing: after every key step, save a snapshot of the current state; once human approval clears, **resume** from the step where it was stuck, with all prior progress preserved exactly as it was. In open-source engineering implementations, this mechanism is called a state machine with checkpoints — every run has an ID and a lifecycle (waiting, planning, executing, waiting on a human, succeeded, failed); when it's stuck at "waiting on a human," state freezes, a snapshot lands, and it can resume at any time [1] .

[1] enterprise_agent_platform (open-source enterprise agent platform engineering project)

It also happens to solve the third worry from Section I: a job running wild at 2 a.m. — the state machine has a timeout and a retry ceiling, and once a loop hits the threshold, it suspends automatically and hands off to a human. "Running a few thousand times at 2 a.m." turns into "suspended at 2 a.m., handled by a human at 9 a.m."

V. Tiered Authorization: Not Every Action Needs a Human Sign-Off

Once the four tests are in place, the easiest way to turn this into theater is the opposite failure: making every action wait for human approval. Approval is a scarce resource — approving everything is functionally the same as approving nothing (people get fatigued, and start clicking "approve" with their eyes closed).

The right answer is to tier by risk:

Action tier	Examples	Handling
Read-only	Query, retrieve, summarize	Auto-execute
Reversible write	Write intermediate results that can be rolled back	Auto-execute + post-hoc spot check
Irreversible write	Write to a production table, delete, overwrite	Approval required
Outbound	Send an email, call a third-party API	Approval required

The basis for tiering is exactly the **call boundary each tool explicitly declared** in Section II — the contract comes first, the tier comes second, one link locked into the next.

There's actually an even earlier question than tiering: **should this step be handed to the agent at all?** One delivery team offered a concrete way to slice this: break a workflow into roughly ten steps, and hardcode about five of them into deterministic logic — math calculations, compliance checks, the kind of thing that can't afford to be wrong, gets no room for the model to improvise and absolutely no room to make a mistake; hand three or four steps to the agent, allowing flexibility; and leave two for human review. The counterexample is just as blunt: something like financial reconciliation shouldn't be left to an agent's judgment at all — what it needs is a deterministic result, not intelligence [2] . The first question in risk tiering isn't "does this need approval" — it's "does this step need intelligence at all."

Now let's walk through this chapter's mechanisms end to end, using a near-disaster that got caught in time (details anonymized).

An expense-approval agent, at the "report the outcome" step, generated an outbound action:

[2] Silicon Valley 101, Episode 240: "The Hottest New Job in Silicon Valley — FDE" (YouTube)

send the full quarter's expense details — employee names and bank card numbers included — as an attachment to a third-party mailbox. That mailbox came from an email disguised as a finance-system notification, which the model had treated as a legitimate configuration. What happened next, second by second: **structural validation** flagged it first — the outbound tool's contract stated "attachments are limited to summary reports; itemized data is prohibited," the parameters were invalid, the action never executed, and the process stopped where it was; **risk tiering** classified the outbound send as requiring approval, and an approval ticket was generated — which run, which step, what it wanted to send, to whom; the customer's finance lead saw the ticket and **rejected** it, with the annotation "that address isn't on the whitelist"; the system **resumed** from its checkpoint, switched to an in-app notification instead, and the eleven steps already completed stayed exactly as they were. No data ever left the boundary, and the customer's IT department saw the full record in the next day's logs. The disaster didn't fail to happen — it was stopped at the door by engineering.

VI. Four Tests, One Sentence for the Customer

Pulling the four sections together: **validatable** (executes only once parameters pass the contract), **pausable** (high-risk actions wait on a ticket), **resumable** (resumes from the breakpoint once approved), **accountable** (who approved what is on paper) (see Figure 11-1).

Figure 11-1: An action fit for production must be validatable, pausable, accountable, and resumable.

[3] Inside an Applied AI company (Pace long-form post)

[4] "AI Job Sense: Stop the Agent Rat Race"

Together, these four tests are the last of the five elements of the harness a front-line company described — the **supervision mechanism**: low-confidence outputs go to a review queue, irreversible actions wait for human approval, and an audit trail follows every step [3] . Different names, same substance: turning "a human watching the agent" into "the architecture watching the agent." Front-line teams build this down to the tool layer, by permission, role, and scenario — the same tool can trigger different consequences depending on who's using it, and in what environment [4] .

For the customer, none of this jargon is necessary. It comes down to one sentence: **every action it takes has a boundary, a record, and a stop.** Every worry the customer had is now answered.

Boundaries on its actions only answer "can it do this." The moment a prompt changes, a model gets swapped, or a tool gets added, a different question shows up right away: does it still do it correctly? If that question goes unanswered after every change, all the boundaries built so far still can't hold delivery quality in place. The next chapter turns that question into a reproducible evaluation system.

Chapter 12 · How to Build an Evaluation System from Zero

> What gets measured gets managed.
> — a maxim widely circulated in management literature; authorship disputed

On Monday, the system classifies a refund as "can be auto-processed." On Tuesday, an engineer swaps out one line of the prompt, and the same order gets routed to a human instead. The business owner asks: "did this get better, or worse?" If yesterday's sample wasn't kept, there's no way to tell — the team is left arguing on gut feeling.

"Did changing the prompt make quality regress?" — a system that can't answer this question doesn't belong in production.

That sounds severe, but take it apart and it's all common sense: without an eval set, every change is a gamble — the bet placed on an engineer's gut. Every customer complaint turns into a he-said-she-said — "it got worse," "no it didn't, you just got pickier" — and nobody can convince anybody. Evaluation isn't a testing step tacked on before launch. It's the floor a production system stands on.

This chapter covers how to build one from zero to production-ready, in four steps: **first build a Golden Set, then a Regression Set, then automated judging, and finally wire it into the change process.**

I. Crossing from 70% to 98%

Chapter 7's case again: on day one in the real environment, the score came in around 70%; the contract promised above 98%. What closes those 28 points?

Not by tweaking the prompt — that's the fine-tuning at the very end. It's a loop: **evaluation surfaces a failure pattern, you fix it specifically, regression testing verifies it, and evaluation surfaces the next one.** The insurance practitioner from before calls this the evaluation loop, and the quality bar is the same testable standard from Chapter 8 — right 995 times out of 1,000 in a week [1] .

[1] Inside an Applied AI company (Pace long-form post)

The same climbing curve has a fully documented public version too. A customer-service software company's first-generation product resolved only 23% of conversations on average; swapping the underlying model pushed that to 51%; deep customization for specific customers pushed the ceiling up to 86%. The vendor itself deployed the same product for its own customer service, tuned it continuously for two years after launch, and pushed the resolution rate to roughly 79%, resolving about 560,000 conversations a month.

What separates those three numbers isn't three model upgrades — the stretch from 51% to 86% was ground out one round at a time against an evaluation set. Another software company walked the same road, tuning its own customer-service agent for a full year before pushing the "couldn't answer" rate down from roughly 30% to under 10%.

Why can evaluation drive all of this? Because the quality of an AI system is continuous, probabilistic, and context-dependent — the same answer that dazzles in a demo can be a disaster in a specific business context, and the authority to define "good" sits with the business side, not the engineering side [2] . Iterating without an evaluation metric is running blind: two weeks of tuning go by, and nobody can say whether quality went up or down. An evaluation system is exactly what turns the phrase "what counts as good" sitting inside a business expert's head into a scoring standard the system can run every single day.

II. Step One: The Golden Set — Standards Grown from Real Cases

A Golden Set is a batch of **real cases carrying explicit judging criteria**: what the input is, what the correct output is, why it counts as correct. It's the foundation of the whole evaluation system, and it only has two sources: **real cases accumulated during the Minimum Viable Deployment (MVD)** (things the system has already answered, that users have actually used) and **disputed cases flagged by front-line users** (answers that looked questionable, or drew a complaint). You can't use cases an engineer invented at their desk — a desk can't invent the mess of the field, and it can't invent the exact line a business expert cares about.

How many sample cases does a cold start need? Generally, fifty to a hundred usable ones are enough to get going. Chasing several hundred in one go is a common detour — more cases don't beat a precise standard.

More critical than the count is **who does the judging**. On the same output, a front-line user might call it a pass, an FDE might call it good, and the customer's business owner might call it a fail — all three standards are correct, because the three people are looking at different things:

[2] Fan Bing, Forward Deployed Engineer (FDE) (XDash open-source book)

the front-line user cares whether it's easy to use, the FDE cares whether it's technically right, the business owner cares whether the company can afford it if it's wrong. The judge has to be pinned down at the moment the eval set gets built, and written into every sample: a Golden Set isn't a pile of test questions — it's a pile of **questions a specific role has already judged against a standard**. This is also where the "who judges" field in Chapter 8's acceptance sheet comes from — the acceptance standard starts taking shape the day the eval set gets built.

The best example of a cold start comes from a financial scenario: hundreds of implicit business rules at one institution — how a loan gets approved, which field anomaly needs to be escalated — lived only in the memory of a few veteran employees, never written down. The approach was to walk those employees through several rounds of real-case simulation: take one real piece of business, have them work through it while explaining why they judged it that way, record it as a rule item on the spot, and feed it to the model as an eval set. The first round is bound to miss things — fill the gaps, run it again, and a few cycles later it settles down. Once it's stable, the veteran employee spot-checks the system's judgments, and it only counts once they sign off [3] . This whole exercise lays the essence of an eval set bare: **a Golden Set is a standard grown out of someone's head** — taking a veteran's implicit judgment and making it explicit, one executable question at a time.

III. Step Two: The Regression Set — A Mistake Log That Only Grows

Once the Golden Set is standing, the evaluation system starts growing on its own: **every real failure becomes a new sample added to the set.**

The system gets a case wrong — it goes in. A user complains about an output — it goes in. An online audit spot-check turns up a problem — it goes in. The set grows the longer the system runs; it's really a mistake log — every entry is a real, costly error, and the point of recording it is to make sure it never happens again. The Regression Set only grows, never shrinks — the one exception being a standard change (say, the judging criteria get revised, and old samples get rejudged against the new standard instead of just getting thrown out).

Two disciplines guard against contamination. **Demo data doesn't get in**: the clean data from a demo environment would dull the eval set's ability to discriminate against the mess of production. **Samples with labeling disagreement get archived separately**: a case where two people can't agree isn't a bad sample — it's actually the most informative one. The disagreement itself is a signal that the standard isn't aligned; set it aside and revisit the standard

[3] "100 Questions About FDE" (open-source ebook)

on a regular cadence.

At this point the division of labor between the Golden Set and the Regression Set is clear: the Golden Set is the **foundation** (the standard gets defined here, small and precise), the Regression Set is the **safety net** (it keeps growing, large and messy). The first guarantees that "good" has a definition; the second guarantees that a past mistake never repeats.

There's an even more effective way to make the Regression Set "meaner": train a dedicated model on historical data — in a voice agent, for instance, train a small model on historical call recordings to specifically simulate the speech patterns a real deployment runs into, an impatient customer, a patient who slurs their words [4] . This kind of "adversarial test set" hugs the edge cases production will actually hit much more closely than a standard QA set does, and it's also a technique for evolving the Regression Set from "recording errors that already happened" to "actively manufacturing edge cases."

IV. Step Three: Automated Judging — Three Tiers, Plus a Human Calibration Pass

With samples in hand, you can't have a human rejudge every one of them after every change. Automated judging splits into three tiers, ascending by cost:

Tier one, exact match and rules. Where an answer can be compared exactly — classification, extraction, structured output — write a rule-based judge: near-zero cost, fully deterministic results. Whatever a rule can judge, never hand to a model.

Tier two, LLM-as-Judge. For open-ended tasks with no single correct answer — is a summary well written, is a reply appropriately toned — a model acts as the judge, scoring against a rubric written into its prompt. This is the mainstream approach in current engineering practice, but there's a discipline that's easy to overlook: **the judge's rubric is itself a "prompt change."** Change it, and the judging standard changes right along with it. So the judge's rubric also has to go under regression management — changing the judge and changing the system go through the same release gate.

Tier three, human spot-checking. Periodically pull a small batch of samples for a human to rejudge, and compare against the model judge's conclusions — not because automation can't be trusted, but because automation **drifts**: a judge model can develop a systematic preference for a

[4] Tencent Research Institute, AI Lens Roundtable, Episode 6: "FDE Non-Consensus Views and a Field Guide from Silicon Valley Founders"

certain writing style, or a systematic blind spot for a certain class of error, and only human spot-checking catches that kind of drift. A judge that's never recalibrated is just wrong, automatically, forever.

Cost awareness runs through all three tiers: not every sample needs the most expensive judge. Whatever a rule can judge goes to a rule; the LLM judge only handles open-ended tasks; humans only spot-check — the evaluation system's own running cost is still a cost (Chapter 14 puts it on the books). These mechanisms have already settled into standard components in open-source engineering practice — offline evaluation to guard against regression, online auditing to test against reality, and a cost-aware change gate [5] .

V. Step Four: The Release Gate — No Deployment Without Passing the Regression Set

Everything built up over the first three steps cashes out at this step: **any prompt change, model swap, or tool modification must pass the Regression Set before it can be deployed.**

This turns "I don't think it regressed" into "regression pass rate: 99.2%." Don't underestimate that shift in wording — the first is a feeling, and it can't enter any decision; the second is a number, and it can walk into an acceptance meeting, a change order, a customer's trust ledger. The line planted back in Chapter 7 pays off here: the one thing an MVD absolutely must build from day one is the eval set — because it's the release gate for every change, and change is the normal state of production.

The gate is also the interface between evaluation and Chapter 8's acceptance sheet: where do the launch-acceptance metrics come from? From the eval set's standard. The baseline is the historical level the eval set produced; the target is the commitment both sides agreed to on the evaluation standard — every number on the acceptance sheet has a reproducible evaluation process standing behind it. Any project where the two documents don't line up has one side making it up as they go.

VI. Online and Offline: The Loop Is the Evaluation Cycle

The gate handles the moment of change; a system already running needs a second set of eyes. Evaluation splits into two tracks:

[1] Inside an Applied AI company (Pace long-form post)

[5] enterprise_agent_platform (open-source enterprise agent platform engineering project)

Offline evaluation guards against regression — it runs against the Regression Set before a change ships, holding the line at "no worse than before." **Online evaluation tests reality** — production stays under continuous sampled audit, and user feedback (upvotes, corrections, complaints) flows back into the Regression Set. Put the two tracks together and you get the full version of the loop this chapter opened with: offline blocks a regressing change from shipping, online discovers a new failure pattern, the new failure becomes a new sample, and the sample makes the next round of offline evaluation stricter. Once it's turning, this is the evaluation loop that practitioner described earlier [1] (see Figure 12–1).

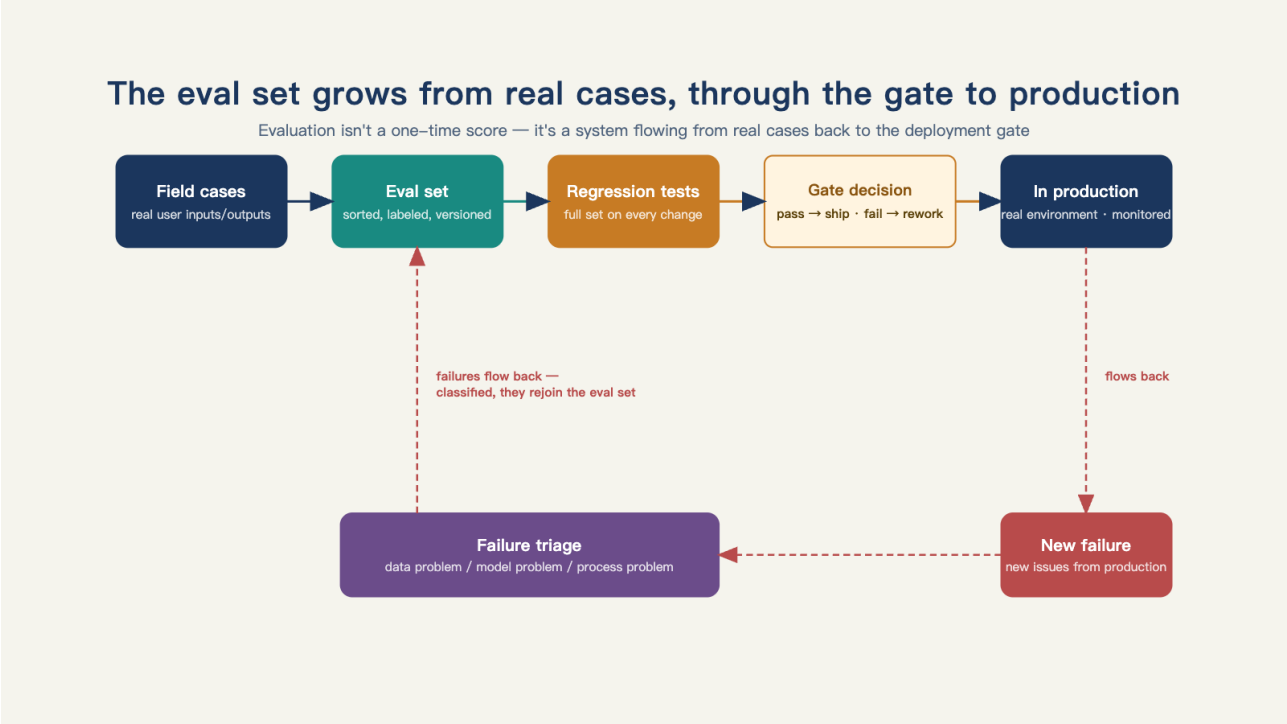

Figure 12–1: The eval set grows out of real–world cases, passes through the regression gate to production, and brings new failures back into the samples.

This methodology has a fully documented public counterpart worth using as a mirror for this chapter. A leading team's practice settled into three steps: **the evaluation set grows out of real cases** (never invented by engineers); **the business side sits as judge** (evaluation is shaped so business experts can actually take part — side–by–side output comparisons, simple better/worse labeling, regular review meetings; as the evaluation set gets more accurate, the business side watches the system get better on its own cases round after round, and trust accumulates from watching it happen with their own eyes); and **evaluation gets wired into the production loop** (launch isn't the finish line — continuous sampling, with an alert the moment scores dip). This book's four steps don't conflict with those three: the three steps describe direction (real, business–owned, closed–loop), while this chapter's four steps are the operating sequence for a cold start (Golden Set first, then Regression Set, then judging, then the release

gate).

The most complete public example of evaluation-first is in agriculture. A farm equipment maker set out to solve herbicide waste: traditional sprayers blanket the whole field, while the new system uses 36 cameras and machine vision to spray only the weeds while moving. But what farmers wanted wasn't technology — it was advice they could trust. The AI team's engineers flew out to the farms and followed the agronomists into the field, first reviewing hundreds of real operating cases together with the experts to build a custom evaluation system, and only then iterating on the model — all of it racing against the planting season. In the end, chemical usage dropped by as much as 70%, and farmer engagement frequency went up sixfold [2] . Those two numbers only exist because "good" had already been defined by an evaluation system.

VII. The Eval Set Is an Asset the FDE Can Take Along

One last piece of arithmetic, pulling the lens back from a single project to the whole team.

When a project ends, the code might get thrown away, the data has to go back to the customer, and the integration solution won't carry over to the next customer — so what actually accumulates? **The eval set, and the process knowledge that grew out of building it.** One practitioner put it clearly: data is the customer's moat, so it can never simultaneously be the vendor's moat too. What actually accumulates is the tacit knowledge and the evaluation loop that each deployment leaves behind [1] .

On the next deployment, the model might be a whole generation newer, and the integration solution gets rebuilt from scratch — but the failure patterns, judging standards, and judge criteria accumulated at the last customer carry straight over as the starting point for the next project's Regression Set. **The eval system is one of the rare engineering assets that doesn't lose value when the model gets upgraded** — the more the model changes generations, the more you need a ruler that doesn't change with it. How this compounding value cashes out at the organizational level is the subject of Chapter 22 (turning field experience into product capability); here, the job is just to build the asset itself.

Three items on a negative checklist, to self-audit against: **building the evaluation platform before the eval set** — that's the order backwards; the platform exists to serve the eval set, and a platform with no eval set is just spinning its wheels. **Running the eval set only once, right before launch** — that's not an evaluation system, it's a launch ceremony. **A judge that's never**

[1] Inside an Applied AI company (Pace long-form post)

[2] Fan Bing, Forward Deployed Engineer (FDE) (XDash open-source book)

recalibrated — with nobody watching for automation drift, automation just becomes an amplifier for its own errors.

Evaluation establishes "provable results." But there's a moment in production that evaluation alone can't reach: the customer points at yesterday's result on the screen and asks "where did this number come from, why was it judged that way at the time" — what that calls for isn't a score. It's evidence.

Chapter 13 · What Proves You Right When the Customer Pushes Back

> Res ipsa loquitur.
> "The thing speaks for itself."
> — an old maxim of Anglo-American tort law, commonly invoked in findings of negligence

At a quarterly business review, the customer's CFO points at a number on the screen and asks: "where does this accounts-payable conclusion come from?" The room goes quiet. If the system can pull up the source documents and pinpoint the exact passage — which invoice, which statement, which contract, and which page of each — the challenge gets settled on the spot. If the only answer is "let us go check and get back to you," then even if the number turns out to be right, trust has already taken damage in the waiting.

A moment of challenge is a moment of reckoning for trust, and what a reckoning calls for isn't attitude — it's evidence.

I. Trust Is a Three-Story Building

A customer's question of "can I trust this result" breaks down into three layers: **What is this answer based on? What was the process behind this judgment? What caused this particular failure?** — a citation you can trace back, a decision you can replay, an anomaly you can diagnose (see Figure 13-1).

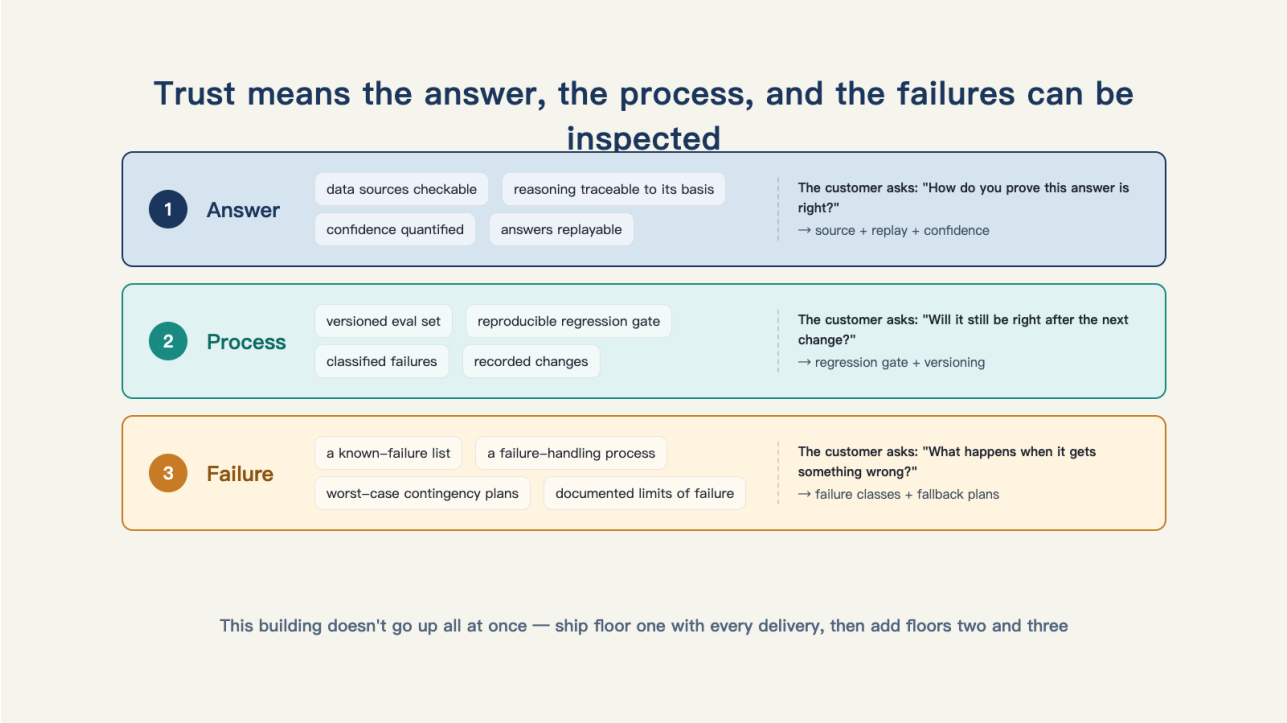

Figure 13-1: Trust isn't a line about "the model is accurate" — it's the answer, the process, and the failure all being inspectable.

All three layers share one property, and it's the most important sentence in this chapter: **they're not reporting material assembled for the customer to see — they're natural byproducts left behind by the system as it runs.** A chain of evidence isn't a packet compiled after the fact once someone raises a challenge; it's a recording capability built into the architecture from the start — logging is an architectural decision, not damage control after the fact. Open-source enterprise agent engineering practice has already turned all three layers into standard components, and the supervision mechanism among the five harness elements a front-line company described likewise requires an audit trail to follow every action — recording what the agent did, and why [1] .

II. Layer One: Every Answer Carries Its Source

The most basic layer: every conclusive answer the agent gives carries an **EvidenceRef** attached to it — which document it's based on, where exactly, and how confident it is [2] .

The full version of the question the last chapter closed on lives right here: the customer asks "where does this number come from," and the answer comes in three parts — what the source is (file name, system, timestamp), where exactly it points (paragraph, page number, field), and how

[1] Inside an Applied AI company (Pace long-form post)

[2] enterprise_agent_platform (open-source enterprise agent platform engineering project)

confident it is. With all three parts present, the customer can follow the citation and check it themselves — every check earns a little more trust.

This layer also hides the other half of hallucination governance. Everyone hopes a stronger model just stops making things up, but the engineering discipline is harder-edged than that: **when retrieval fails, the answer has to say "not found" — it is never allowed to make something up.** Saying "not found" is just a failure; inventing a plausible-looking answer is an overdraft against trust. So the architecture has to enforce this: a conclusive output with no EvidenceRef attached is not allowed to go out. If retrieval quality falls short, the citation chain breaks first, and any reasoning or explanation built on top of it never gets off the ground.

III. Layer Two: Every Decision Step Has to Be Replayable

The second layer goes one level deeper: it's not just the answer that carries a source — **the entire process of arriving at that answer can be replayed.**

In engineering terms this takes the shape of a run trace: every run the agent makes gets recorded event by event — what data it read, what tool it called, what parameters it passed, what result it got, how it reasoned along the way. One timeline, start to finish [2] .

The value of replay pays off in three scenarios. **When the customer challenges a result**: you can give them not just "where the number came from" but "exactly how, step by step, it got reasoned into this number" — the EvidenceRef from Section II is a snapshot; the run trace is the full recording. **During incident review**: when something goes wrong, an event-by-event replay pinpoints exactly which step things started to drift — Chapter 11's approval ticket recorded "who approved it"; the run trace adds "what the system was looking at right before the approval." **When evaluation catches a failure**: for a sample that failed in the Regression Set, open the trace and see exactly which step it failed at — this connects right back to Chapter 12's diagnosis, and leads into this chapter's final layer.

IV. Layer Three: Failures Have to Be Classified, or You Can't Fix Anything

With a trace in hand, an anomaly can finally be diagnosed. But diagnosis has one precondition: **failures have to be classified, because different classes of failure get fixed in completely**

[2] enterprise_agent_platform (open-source enterprise agent platform engineering project)

different ways.

Failures in production split into at least four classes:

- **Retrieval failure** — it didn't find what it should have. The fix lives in the data: fill in the index, tune recall, adjust the chunking. No matter how smart the system is, it's useless if you can't feed it the right material.
- **Reasoning failure** — it found the right thing, and used it wrong. The fix lives in the model and evaluation: improve the prompt, add a judging rule, put this case into the Regression Set.
- **Tool failure** — the tool was called, and the tool failed. The fix lives in the engineering implementation: timeout, retry, graceful degradation (Chapter 14's four resilience primitives).
- **Data failure** — the source itself was wrong. The fix lives on the customer's side: upstream data-quality governance, which may mean going back to Chapter 10's Data Contract.

Four classes of failure, four different directions, four different owners of responsibility. Lump them together — the most common way is calling everything "the model isn't good enough" — and there's no telling where to even start fixing it: throwing a model upgrade at a retrieval failure, throwing a prompt rewrite at a data failure — plenty of money spent, the same errors keep showing up. The run trace is what classification runs on: replay to the failing step, see whether it got stuck fetching material, judging, executing, or at the source, and the class sorts itself out.

V. The Playbook for Proving Yourself: Four Standard Moves

The architecture provides three layers of capability — how do you use them to actually face a challenge? There's a standard four-move playbook:

Move one, make the challenge concrete. Don't say "our system is reliable overall." Say "which specific case are you referring to — let me pull it up." Pin a vague challenge down to a specific record.

Move two, put the evidence on the table. EvidenceRef plus the run trace, produced right there on the spot — source, location, process, all three laid out on the table, not "we'll check and get back to you."

Move three, attribute and classify. If there really is a problem, classify it on the spot — which of the four failure classes it falls into, and where the line of responsibility sits (a problem at the

data source belongs to the customer's governance; a reasoning problem belongs to our fix).

Move four, commit to the fix. This failure case goes into the Regression Set (Chapter 12's mistake log), and the fix has to pass the release gate before it ships — give a deadline with acceptance criteria attached, not a vague "we'll optimize it."

These four moves turn "proving yourself" from a defense into a live demonstration of quality improvement. Facing the same accusation of "you got this wrong," a customer who's been walked through the four moves doesn't see a company admitting fault — they see a system operating. In a lot of long-term relationships, handling one challenge well is worth ten smooth status reports. This playbook echoes forward and back with Chapter 8's responsibility handoff table and Chapter 17's handoff checklist: the responsibility table answers "who do you go to when something goes wrong"; this answers "once you've found them, what do they actually show you."

VI. The Cost of Leaving a Trail, and the Payoff of a Single Audit

A chain of evidence isn't free. A full, event-by-event trace means real storage and logging overhead — record the complete replay for every single run, and the bill arrives before the value does. So recording needs its own tiers: **fully record critical runs** (anything touching money, approval, or an outbound send), **sample-record routine runs** (with a sampling rate aligned to evaluation's own spot-check rate).

The basis for tiering is the exact same one from Chapter 11's risk tiering — a high-risk action gets a high-specification trail [2] . This storage bill ultimately lands in Chapter 14's cost governance; here, the principle to plant is this: the specification of the trail follows risk, not a compulsion for completeness.

One last scenario, to see the commercial payoff of getting this wall down:

A customer's audit department ran a routine spot check, randomly pulling 20 historical outputs and demanding a documented source for each one. The vendor came back the same day: 18 of the 20 had a fully traceable chain of evidence — source file, location, and run trace all present; 2 were missing due to a recording-policy change that month, with an explanation and a backfill plan attached. The audit department didn't ask a second question. On the renewal negotiation table, that 18-out-of-20 record later became a slide in the vendor's deck — **an evidence-completeness rate turned into leverage for renewal.**

[2] enterprise_agent_platform (open-source enterprise agent platform engineering project)

At this point, the two hardest walls of the eight to quantify — provable results, trustworthy results — are both standing.The last set of walls in Part Three are two sides of the same coin: the cost bill and stability. That's what the next chapter takes up.

Chapter 14 · What to Do About Runaway Bills and Jitter

> Hence the victories of a skillful fighter bring him neither reputation for wisdom nor credit for courage.
> — Sun Tzu, The Art of War, "Tactical Dispositions"

A call cost pennies in the demo; the monthly bill hits five figures after launch. The demo ran smoothly a hundred times in a row; the system starts glitching on day three in production. These two things usually get treated as two separate problems — one goes to finance, the other to ops.

They're actually one disease: **the demo architecture never had a place for cost or failure in the first place.** A demo environment naturally runs short chains, low concurrency, clean data, and a human watching the whole time — cost and failure never get fully exposed. Production strips all of that protection away, and the root cause finally shows itself. The fix is turning both cost and stability into **explicit engineering objects** — cost attributed by run, with a settable circuit breaker on the budget; stability managed by commitment, with graceful degradation and rollback built in.

I. Dissecting the Bill

The bill arrives at month's end, five figures. The project manager goes through it line by line, and not one charge is wrong — every single one is the system just "doing its job normally." Cost blowing out is never a one-off glitch. It's structural.

Cause one, context bloat. A demo runs a single round of Q&A; production runs a chain dozens of steps long — every step carries the entire preceding conversation forward, the context snowballs, and billing follows the context. The chain is an order of magnitude longer than the demo, so naturally the bill is too. The fix: split the task chain by step, carry only the necessary context at each step, and write intermediate conclusions to storage instead of stuffing them into the conversation.

Cause two, no caching. The same question gets asked twenty times over the course of a day, and every single time gets recomputed at full price. The fix: cache and reuse answers to

high-frequency questions, and recompute only when something changes — what gets saved is repetition, not quality.

Cause three, model mismatch. Using the most expensive flagship model for a simple task is like using a Rolls-Royce Phantom to deliver takeout. The fix is Section III's subject: routing. For work like classification, extraction, and format conversion, a small model is faster, cheaper, and just as good.

Cause four, retry storms. A failed call retries automatically with no ceiling — a downstream service jitters for ten minutes, and the retries multiply the call volume several times over, with the most expensive minutes being exactly the minutes of the incident. The fix is in Section V: retries have to be bounded, back off, and have a circuit breaker underneath as a last resort.

These four causes make up the most common structural sources of a runaway bill, but look at them one by one and not a single one is "waste": every charge is compliant, every charge is necessary, every charge is doing real work.

Lesson one on cost: attribute first, optimize second — if you don't know where the money is going, optimization is just flailing blind.

II. Cost Attribution

The engineering form of attribution: every dollar spent gets traced back to **which run, which tenant, which tool** [1] .

Attribute by run, and the bill stops being a lump sum and becomes a map: what kind of task drove this month's bill up? Which customer's usage is climbing? Which tool's call volume is ballooning? A cost dashboard isn't a finance report — it's part of the runbook. The engineer on duty scans it every day, and an abnormal trend gets handled that same week, not left until month's end.

Above the dashboard sits the brake: a **budget circuit breaker**. Once a single run's spend crosses a threshold, it pauses automatically and hands off for human confirmation — not stinginess about saving money, but a fuse to stop the bleeding: a task chain that's run off the rails (the third worry from Chapter 11's list) becomes "paused, awaiting confirmation" with a circuit breaker, and "ran all night, see you at the bill" without one. The circuit breaker reuses exactly the mechanism from Chapter 11: crossing the threshold triggers an approval ticket, and once a human clears it, the run resumes from its checkpoint — the same mechanism, just one more

[1] enterprise_agent_platform (open-source enterprise agent platform engineering project)

scenario for it.

Run one set of numbers through a bill diagnosis: a $48,000 monthly bill, broken apart — $21,000 from context bloat (a thirty-step process chain carrying the full context at every step), $9,000 from no caching (high-frequency repeat questions recomputed at full price every time), $8,000 from model mismatch (extraction tasks running entirely on the flagship model), $10,000 from a retry storm (one downstream jitter amplified by unbounded retries). Each of the four gets its own treatment: split the chain by step, cache the high-frequency queries, route the tasks, cap the retries. Bring these structural sources down one by one, and next month's bill can drop to $13,000, with no loss of functionality.

The room for cost optimization is in the structure, not in the features.

III. Model Routing

The direct fix for mismatch is model routing: **tier tasks by difficulty, and route each one to a model that's just good enough** — classification and extraction go to a small model, complex reasoning over ambiguous clauses goes to a large one; for any critical write-back, beyond weighing capability and price, route it onto a verified, controlled execution path — validate first, write second, and wait for approval where needed [2] .

In real-world work, choosing a model under cost constraints is the norm, not the exception. The key to routing design is that it goes **into the architecture, not patched on afterward** — routing bolted on after the fact is one patch stacked on another; routing designed in from the start is a natural byproduct of task decomposition.

This lines up with Chapter 9's judgment: model orchestration (which task goes to which model) is itself one of the five harness elements from Chapter 9 [2] — it isn't a cost-optimization trick, it's a high-priority piece of production architecture.

IV. Where Jitter Comes From, and How to Fix It

On to stability. First, get clear on where "glitching" actually comes from.

A production system's jitter comes from four sources: **dependency jitter** (a downstream API or model service timing out); **output jitter** (the same input producing two different outputs — the nature of a probabilistic system); **data-source jitter** (the customer's data updating, a field

[2] Inside an Applied AI company (Pace long-form post)

drifting); and **load jitter** (a concurrency spike — everyone in the company using it at once when the books close at month's end).

None of the four is the kind of thing you "fix once and it never happens again" — jitter is the normal state. What turns it into engineering is how you commit to it.

The standard practice in this field is to give a Service Level Objective (SLO), and the correct way to write one is **to commit to three dimensions separately** [1]:

- **Availability** — what fraction of requests return normally (99.9% and 99% describe two completely different systems);
- **Latency** — what fraction of requests return within a given time (a fast average is meaningless; what matters is the long tail);
- **Quality** — what fraction of outputs clear the passing bar.

The third is the easiest to leave out, and the most important: **the quality commitment's standard comes directly from Chapter 12's eval set** — "passing" isn't an adjective, it's a bar defined by the Regression Set. The reason the three dimensions get written separately is that their cost curves are different: every extra nine of availability multiplies redundancy cost; taming the latency tail takes architecture, not raw compute; and the quality bar is held up by the evaluation loop. Collapse all three into one line — "the system runs stably" — and none of the three has actually been committed to.

V. The Four Resilience Primitives: What to Do When Something Breaks

The SLO is the commitment; resilience is the mechanism that makes good on it. Four primitives, each aimed at one way things die:

Timeout — don't wait yourself to death. Every external call gets a time limit; once it's hit, count it as a failure and hand it to the next step. In a system with no timeouts, one stuck service can drag the entire chain down with it.

Retry — bounded, with backoff. Retries need a ceiling (to prevent the storm from Section I's cause four), and an exponential backoff between attempts (to give the downstream room to breathe) — retrying blindly and immediately is like pouring gasoline on a burning house.

Idempotency — a retry shouldn't charge twice. Sending the same operation a second time has to produce the same effect as sending it once. Retrying without idempotency is dangerous: after

[1] enterprise_agent_platform (open-source enterprise agent platform engineering project)

a timeout, you don't actually know whether the other side succeeded, and sending it again might mean two charges instead of one.

Graceful degradation — a tiered fallback plan. For minor trouble, a smaller model stands in (routing steps down); for a moderate failure, a rules engine takes over (falling back to deterministic logic); in the worst case, **fall back to a human and a spreadsheet.** This was already flagged in Chapter 8: only a plan that dares to write down "fall back to a human" dares to go live — a degradation path isn't a fig leaf for failure, it's a safety net designed on purpose. What the customer feels from this tiering is exactly where their sense of security comes from: when the system breaks, there's somewhere to go, and the sky doesn't fall (see Figure 14–1).

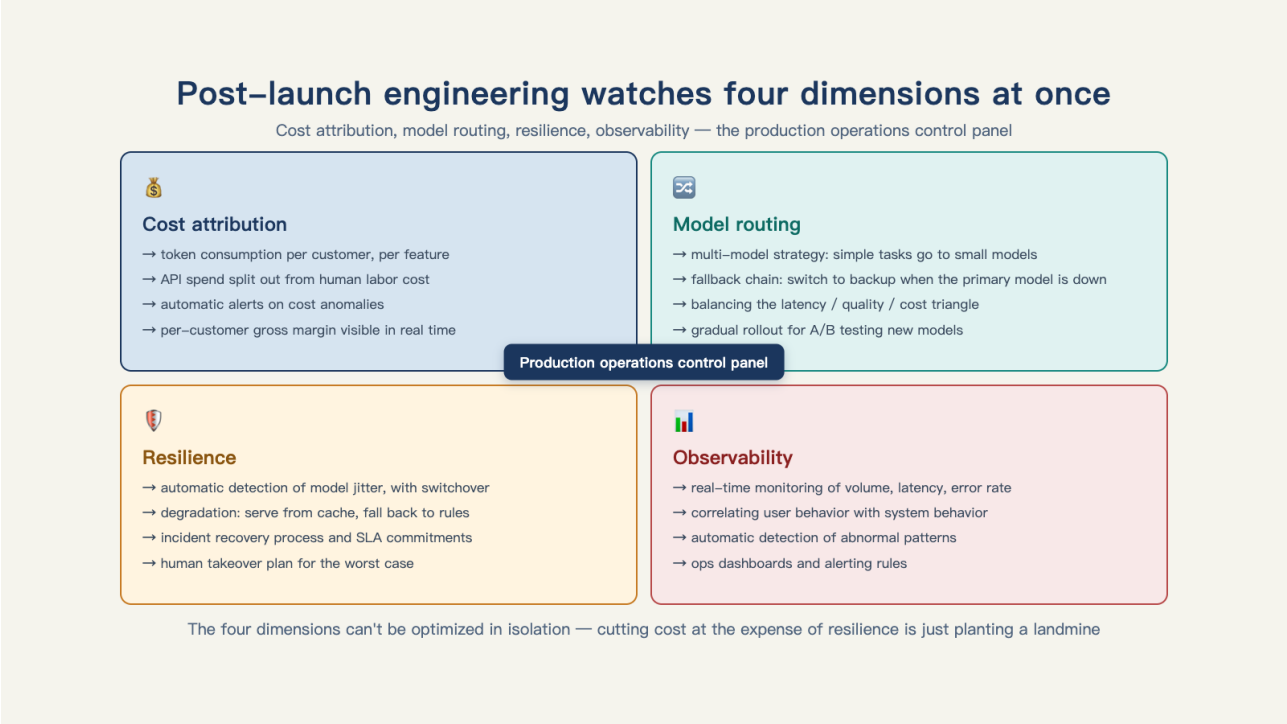

Figure 14–1: Post–launch engineering has to watch cost attribution, model routing, and recoverability during failure all at once.

VI. A One–Page Runbook

Pull everything in this chapter together into one page for the customer — the runbook: who watches which alert (Chapter 8's four questions), which page to flip to for which incident (the four failure classes, Chapter 13), when to hit the fallback button (the three degradation tiers), where the cost dashboard lives, what the circuit–breaker threshold is. **Write it for whoever's on duty on the customer's side, not for the engineers** — a runbook that only exists on the vendor's laptop is the same as no runbook at all (Chapter 8's discipline, restated here).

One last question about cost: after squeezing the structure and writing the commitments, the customer still thinks it's expensive — what then? Look first at a maintenance–dispatch scenario: an agent costs two thousand dollars a day, and what it's replacing is the decision of "which engineer to send out for the repair." If the cost of sending the wrong person is far higher than two thousand dollars, then the math can't just compare the API bill against zero. This scenario doesn't call for a better pitch — it calls for cost's first–principles question: **compared to what?** Compared to the human decision cost it's replacing, compared to the cost of getting it wrong — never compared to zero.

That closes out all eight walls of Part Three: data and permissions, integration, tool boundaries, controllable execution, provable results, trustworthy results, cost and stability, and transferable responsibility (the last wall gets its full treatment in Chapter 17).

Looking back, these eight walls all stand on one shared principle — **the first design constraint on a production system isn't "smarter." It's "what happens when it breaks."** A predictable small glitch beats unpredictable brilliance; the skilled fighter wins no glorious battles, and a well–run system has no firefighting stories to tell.

But laying the engineering foundation only answers "can the system stand on its own." Once it can, the real test is just beginning — if nobody uses it, everything before this was just cost. That's what Part Four takes up.

Part Four · The Real Work Begins After Go-Live

> This part has four chapters, and together they answer one question: **once a system has "shipped" in the technical sense, how do you make it "live" in the organizational sense — used by people, backed by people, and still running once you walk away.**

Part Three wrapped up with the data connected, actions under control, outcomes provable, and bills accountable. But a production system's real enemy isn't only engineering — it's the organization. A system that scores full marks at acceptance can die quietly within three months, and the cause of death is never in the code. It's in the incentives.

Chapter 15 dissects this silent death first: going live is not the same as activation, and all three failure modes happen only after "successful launch." The antidote is three questions about incentives, embedding change gradually instead of all at once, and cultivating two kinds of allies.

Chapter 16 hands the key to its rightful owner: AI adoption is inherently cross-departmental, and budget, process, data, and benefit are four cards only an executive sponsor can hold at once. Executive sponsorship isn't a slogan — it's three kinds of backing (budget, incentives, failure), two gates the project has to pass through, and a flywheel that gets heavier the more it spins.

Chapter 17 defines where delivery actually ends: after acceptance and adoption there's a third stage — handoff. The three exit conditions, the handoff ladder, three anti-patterns — all boiling down to one line: a system that can't survive without its FDE was never delivered. It was parasitic.

Chapter 18 turns the lens back to China: procurement structure, trust structure, and who defines the last mile all work differently there. Four structural differences give rise to two homegrown forms — and because China's constraints are tighter, they make an ideal stress test for the methods in this book.

The four chapters form a single progressive chain: first get the system used (adoption), then make sure an organization is backing that use (executive sponsorship), then, once it's stable, hand it off (handoff) — and how hard that handoff turns out to be is finally decided by the soil it's planted in (China). The primary reader for this part is **whoever is on the hook for "the system is actually alive"**: the delivery lead running the project, the customer's IT or informatization owner, and any manager stuck staring at a system that shipped and nobody

uses. Engineering takes half a step back here, and the organization takes half a step forward — because every step in this part starts from the same line, said in a conference room: "Why won't anyone use this thing?"

Chapter 15 · It Shipped — So Why Is No One Using It?

> Show me the incentive and I will show you the outcome.
>
> — Charlie Munger

The system finally went live.

The acceptance review was a picture of decorum: every technical metric checked off one by one, the project leads from both sides shaking hands for a photo, and once the customer's IT department read out "the system is now officially in operation," the applause came right on cue.

Three months later, a follow-up visit turns up a different picture: the last login is still stuck in week two after go-live, the business team's backbone users still have the same six-year-old spreadsheet open on their desktops, and when you ask about the system, the answer is "oh, that thing — I open it once in a while."

No incident, no complaint, no pushback — the system simply died, quietly and with perfect decorum.

This chapter takes up the question: once a system is live, has passed every technical acceptance check, and shows green on every metric, why does the business side still refuse to use it?

I. Go-Live Is a Process. Activation Happens in Behavior.

Start by pulling apart two words people conflate. "Going live" completes an administrative process — procurement and acceptance: the contract is fulfilled, deployment is finished, the acceptance form is signed. Whether a system is actually alive is judged by exactly one standard: has the target user group built a stable habit of using it in their day-to-day work — not applause at the demo, but the system still getting heavy use three months later, with nobody having to chase anyone.

The enterprise software industry has never been short on projects like this: every item on the feature checklist ticked off, and only five percent of people actually using it; the system hits four nines of availability, and the business side still prefers its spreadsheets.

[1] Fan Bing, Forward Deployed Engineer (FDE) (XDash open-source book)

Why do enterprise deployments so consistently die at activation? Because the three gates it has to pass are none of them in the code:users find it awkward — one extra step over the old habit, and nobody uses it;users don't trust it — one mistake, and trust resets to zero;users see it as none of their business — and then nobody touches it. [1]

None of the three gates can be cleared back at headquarters. They can only be passed one at a time, at the customer — in their conference rooms, on their shop floors, at their desks.

This is exactly the watershed between an FDE and traditional "deliver and leave" implementation: traditional implementation treats the acceptance sheet as the finish line; an FDE treats a change in the customer's behavior as the finish line.

II. Dying Quietly After "Successful Launch"

A frontline FDE practitioner, on a podcast, boiled the causes of failed projects down to three:

First, the executive sponsor's expectations of what AI can do exceed reality. They assume that once AI is switched on, the company simply takes off — they treat it as a superhero, not a tool. Expectations set that high are guaranteed to crash, and the crash ends in abandonment.

Second, the project gets initiated by the technical department instead of the business department. The business side treats itself as the client and the technical team as the vendor — a system initiated this way is almost guaranteed to fail, because the business team is the one who understands the business, and the technical team can't supply the knowledge and judgment behind product selection, forecasting, or negotiation. The system ends up as the technical team's achievement, not the business's tool.

Third, and most fundamental: the organization's incentive structure never changed to match. In the guest's own words — "Don't think of this as a piece of software going live. Think of it as a batch of new employees going live — a new wave of productive capacity has arrived, and the relations of production have to change." [2] In plain terms: the system isn't just one more piece of software, it's a batch of "new hires" — and when new hires come on board, performance metrics, division of labor, and reporting lines all have to be rearranged accordingly. Keep managing it the way you'd manage a tool — leaving performance metrics and division of labor untouched — and this batch of "new hires" naturally can't perform.

Notice what these three failure modes have in common: not one of them happens before go-live. All of them happen after "successful launch." At the moment of going live, the project is still

[2] Crossroads × Rolling AI, "A Conversation on FDE" (podcast)

scoring full marks; the death happens quietly, day by day, over the following three months. Of the three causes, the first two point at the executive sponsor (unpacked in full in the next chapter); the third — incentives — is what this chapter focuses on. First, a look at more real feedback from the field.

One engineer who ran four iterations on a customer-service project — the four versions evolving from an orchestration framework all the way to a locally deployed system with a full engineering shell — was technically a success. Asked what the project actually did for the business, his answer was blunt: [3] without headcount cuts, there's no way to show a return. Not one person was let go from their customer-service team; the time saved just went toward "creating more value" — but that made the ROI impossible to pin down, because in most bosses' arithmetic, time saved doesn't count, only headcount saved does. Here, the ROI math and the employees' interests collide directly.

Another practitioner complained: "With AI we're on 997 [9am–9pm, 7 days a week]; without it we were only on 996." — the tool added workload instead of cutting it: when the system crashes, someone has to firefight; when its memory gets wiped, someone has to re-feed it; when a process jams, someone has to unblock it by hand. [4]

None of this feedback is a complaint about "the project failing" — the technical capability was never the problem.

What's really stuck is structural: the time saved never converts into the ROI ledger, and the extra work created never gets reassigned to anyone.

The project was built right, and the people using it are no better off for it — that's the actual problem.

III. What Training Can't Fix, Incentives Can

Unpack the third failure mode into a usable diagnostic tool. For frontline users, whether they use a system comes down to three questions — nobody asks them out loud; users run the math silently, in their own heads:

First: does using this system change how I'm evaluated? If performance metrics stay the same old ones, every minute spent learning the new tool is unpaid overtime.

[3] "AI Job Sense: Stop the Agent Rat Race"

[4] Yilu Tongxing (一路瞳行), Episode 134: "From AI Assistant to Digital Employee"

Second: who gets the time I save? If the entire benefit of the efficiency gain goes to the company while I bear the cost — learning the new tool, absorbing its new mistakes — nobody's going to sign up for that deal.

Third: whose fault is it when the system errs? If mistakes made by the new system count against me while mistakes in the old process are treated as "just normal," then the rational choice is to stick with the old process.

Get any one of these three answers wrong, and no amount of polished training will help — training solves "can I," incentives decide "will I." The practitioner quoted earlier offered a positive counterexample: when a sales team adopted a new tool, its scorecard was rebuilt from 100% results-based to 80% results plus 20% process — process data generated by the new system converted directly into performance points, and only then did adoption of the tool actually stick.

There's an even more thorough positive example of incentive design. A housing-rental platform has a "property manager" role that handles everything a tenant needs, big and small — a barking dog next door, a leaking air conditioner, all the small, emotionally charged stuff. When the AI team came in, they didn't build "auto-reply." They redrew the division of labor first: the machine took over all the routine responses, and the humans moved to the warm, high-touch care work — paired with an agent that surfaces sales opportunities, prompting the manager that "this tenant has a cat and is traveling this week — worth pitching a pet-sitting add-on."

Over a year, output per person went from one manager per five hundred tenants to one per twelve hundred, with a target of two thousand the following year. The key detail: nobody in that department was laid off. [2] When efficiency gains aren't tied to headcount cuts, employees don't treat the system as an enemy — "no layoffs" here isn't a moral gesture. It's the incentive design itself.

IV. Employee Hostility: Fear Curdles into Non-Cooperation

Over the past couple of years, "teach the AI and you've just made yourself redundant" has become the office worker's unspoken dark joke. "Distillation" started out as a model-training term, and "distilling a coworker" is now catching on too — shorthand for feeding a veteran employee's experience into an AI, and then that employee getting "optimized" out of a job.

Behind the joke is a real backdrop: a steady drumbeat of AI-related layoff news has turned "is my experience even worth anything" into an alarm that can go off in any frontline employee's

[2] Crossroads × Rolling AI, "A Conversation on FDE" (podcast)

head at any moment.

Self-preservation is instinct, and once that instinct fires, almost nobody says no to your face. What's far more common is quietly folding that wariness into everyday cooperation: they'll teach it, but they'll hold something back; they'll cooperate, but they'll stall wherever they can and nitpick wherever they can.

Incentive design solves "is this worth it" — but however clean the math, it can't stop an employee from distrusting the system at a gut level. That distrust is harder to deal with than "not worth it," because it usually goes unspoken.

At a company that delivers enterprise agents and pairs a frontline veteran with an "AI apprentice," the veteran will say: "you're distilling my experience — once you've learned everything I know, I've got nothing left to be worth to this company." [2] That line names precisely the other half of what the earlier sections never quite spelled out: everything before this was about "is the math worth it"; this is about "even if the math checks out, I still don't want to use you."

That company's response has two layers. The first: reciprocity comes before extraction — "before you learn from him, you have to give him something first." Their AI apprentice doesn't start by studying under the veteran; it starts by working — offering suggestions when the veteran builds a shift schedule, offering input when the veteran forecasts sales. Only once the veteran genuinely thinks "this apprentice is useful" does real teaching begin. The second layer: redefine the value — "a strong frontline employee whose experience is worth distilling is actually one of the company's biggest assets — you shouldn't be trying to get rid of him," because the business context is shifting every day, and AI can help think through a lot of things, but it can't replace frontline judgment and decision-making.

Push further, and this employee's value hasn't shrunk — it's grown. A top salesperson used to be impressive by selling two or three times as much personally, or, with more skill, by building a team that sold two or three times as much. Now, the judgment and methodology they've built up can be replicated across a thousand, ten thousand salespeople nationwide — their value to the company can only get bigger, and the incentive attached to it has to rise with it, not stay flat. The veteran's new identity in the system is trainer and authority, not the thing being replaced.

The same fear shows up in a client one enterprise AI adoption consultant worked with: "It's all fully automatic now — why does it still need me to click a button? What reason is there for me

[2] Crossroads × Rolling AI, "A Conversation on FDE" (podcast)

[5] Turning Point (破局点), Episode 31: "FDE, Chinese Style"

to even still exist?" [5] This isn't melodrama, it's genuine existential anxiety: if a person's entire responsibility in a workflow has been compressed down to "click to confirm," that role was already hollowed out to begin with. Adoption design has to answer head-on: what is this person's new role in the workflow? Even if the new role is nothing more than judgment and accountability, it has to be stated explicitly and written into the new job description — leave it unsaid, and the employee is left to guess in a fog of anxiety, and the guess almost always lands on the worst-case scenario.

Fear doesn't always get said out loud — more often it turns quietly into behavior instead of open opposition. Four patterns show up most:

Stalling — sitting on issues that should be flagged, and letting the system run straight into them;Hedging — cooperating on the surface while quietly keeping the old process running as a fallback, so that when something goes wrong, there's proof "I said this wouldn't work";Nitpicking — hunting for flaws, piling on edge-case requests and "this doesn't work well" complaints to slow acceptance down and make the system look less finished than it is;Bad-mouthing — describing the experience to coworkers in the most negative terms possible, pulling the undecided into opposition before they've even formed their own view.

What all four have in common is their killing power: they wear the costume of "cooperation," which makes them harder to spot and harder to dismantle than open opposition.

In practice, three moves reliably defuse this kind of non-cooperation: don't let the system present itself as a replacement for people (AI handles the repetitive work, people handle the judgment calls); go after the user's most annoying repetitive task first, so they get an early taste of the payoff; and let the people who benefit speak for you — one testimonial from a user who's actually gained something does more work than a hundred explanations from you. [6]

The "no layoffs in this department" promise from the property-manager case, and the gradual shift in the weekly-report example from "writing the report" to "just leaving your work trail behind" — both are, at bottom, about catching this fear first and talking incentives second. Leave the fear unresolved, and no matter how clean the math is, nobody will dare use the system — some people may even quietly root for it to fail. And the real cure for this fear usually isn't in the project team's hands: what employees are watching for isn't what some project owner says, it's how the very top of the organization positions itself. Who opens that door, and how, is unpacked in full in the next chapter.

[6] "100 Questions About FDE" (open-source ebook)

V. Gradual Change: Don't Go All the Way at Once

Beyond incentives, the second front in adoption is how the system enters the workflow.

There's a rule that keeps proving itself here: embedding into an old habit beats starting from a blank slate.

One investment firm's weekly-report automation went through a textbook three stages: stage one, a daily-report bot built into their collaboration platform, still filled in by hand; stage two, switched to voice — a few sentences on the way home about "what kept you busy this week, what's stuck," and the system turns that into a report; stage three, not even voice is needed anymore — the system is authorized to read the trail people leave behind at work directly — chat logs, documents, calendars — and auto-compiles the summary. People went from "writing the weekly report" to "just leaving their work behind properly." [4] Notice the direction of this path: it's not throwing the person out of the process all at once, it's having each stage ask a little less of them while leaving them a little more confirmation authority.

The counter-lesson cuts deeper. In one widely circulated case, a group of veteran workers who had run an eleven-step process for years were handed an agent that did the whole thing "all at once" — and they simply stopped using it. Adoption only came back once the design was changed to keep the original steps, have the agent complete each one incrementally, and let the workers see the intermediate output. [7] The logic isn't mysterious: intermediate output is a person's confirmation point, and a confirmation point is what trust rides on. Hide every intermediate step inside a black box, and you're effectively asking the user to put their authority on the line to vouch for that black box — nobody's willing to do that.

So when designing for adoption, start with a plain question: is what I'm building meant to remake how this person works, or is it something they'd be glad to pick up and use as-is?

This question gets pushed to its extreme in "fully automated" scenarios. One enterprise AI adoption consultant relayed a client's judgment on fully automated quality: "what comes out of full automation usually only rates a sixty or seventy." [5] The reason is plain: fully automated output is exactly as good as the model, and the model is shared by everyone — when the model improves, everyone's output improves together, which is the same as nobody having improved relative to anyone else. The mediocrity of full automation is therefore the ceiling on adoption: users quickly notice that the system's output doesn't match their own professional reputation,

[4] Yilu Tongxing (一路瞳行), Episode 134: "From AI Assistant to Digital Employee"

[5] Turning Point (破局点), Episode 31: "FDE, Chinese Style"

[7] "FDE Engineer 101: Bridging the Gap Between AI and Business" (Chinese-language compilation video)

and abandonment is just a matter of time. Flip it around, and the twenty points of judgment a person contributes on top is exactly the gap between the system's passing grade and an excellent one — which is a deeper reason for keeping "intermediate output" and "confirmation points" in gradual embedding: it isn't reluctance to change the process, it's that those twenty points can't happen without a person.

People should be the answer to the problem, not an obstacle in its way.

VI. Running the Organization as Adoption Infrastructure

Incentives and embedding solve "will it be caught" and "how it gets caught"; dealing with employee hostility solves "can it be trusted." Stretch the timeline out over the whole project, and adoption has one more organizational layer — not defending against fear, but cultivating allies. Two types of people are worth naming individually (see Figure 15–1). [1]

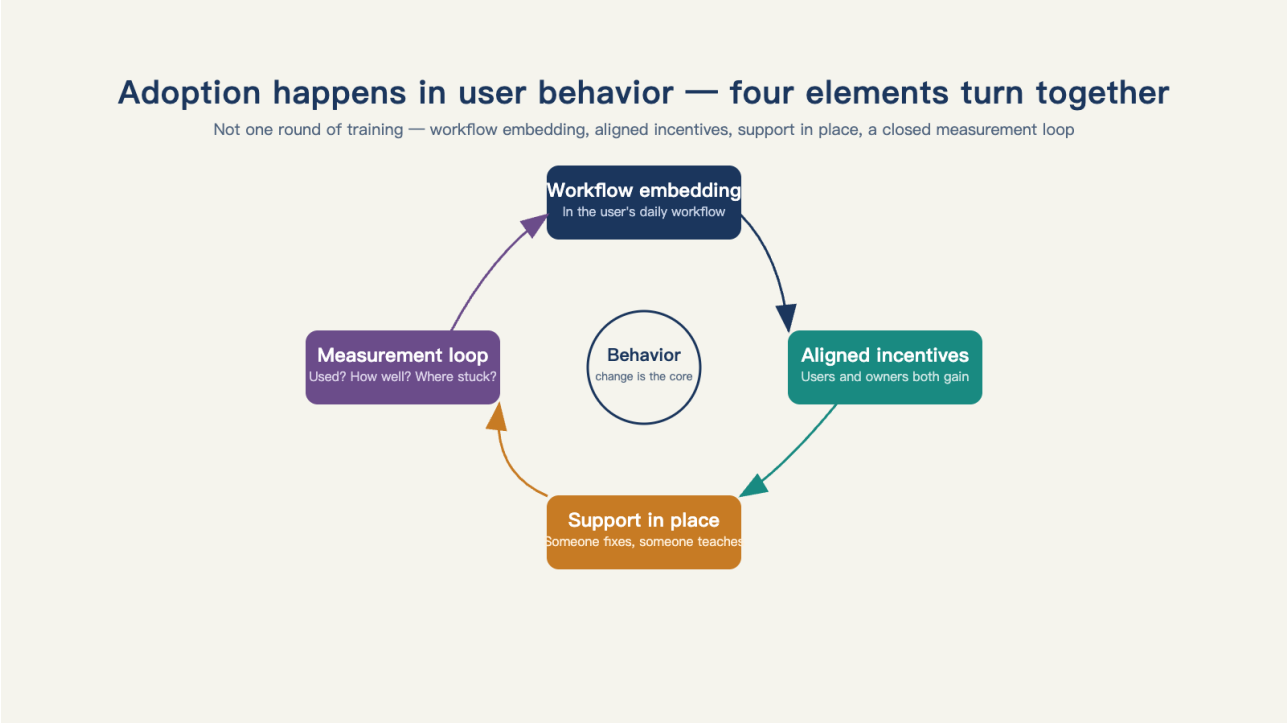

Figure 15–1: Adoption happens in user behavior — it needs workflow, incentives, support, and measurement all turning together.

Sponsors: people inside the customer's organization who genuinely believe in this and are willing to stake their own credibility to clear the way for you. Cultivating a sponsor has three parts — give them material they can actually take and present, turn their foresight into their own track record, and stand in front of them when something fails. One well-tended sponsor is worth ten

[1] Fan Bing, Forward Deployed Engineer (FDE) (XDash open-source book)

product pitch meetings.

Influencers: the uncrowned kings of any organization — the veteran on the shop floor, the person in a department everyone says "just ask them." They don't hold formal power, but they hold trust; one "this thing actually works" outweighs three all-hands emails from management. Letting them try it first, taking every complaint from them seriously, and making their input visible ("this field was added because Master Wang suggested it") is the shortest path around the "I didn't build it, so I won't use it" reflex.

Last is pace. Iteration during the deployment window has to run on "days," not "versions": a business user says a field is missing from the output in the morning, and the field is added by the afternoon; a shop-floor supervisor says the button's too small to hit accurately with gloves on today, and the button is doubled in size tomorrow. This response speed matters far more than the feature itself — users form their judgment within the first month: is this team actually here for real, or just another one that delivers and disappears. The prioritization rule flips too, relative to product-cycle norms: don't rank by feature importance, rank by "adoption-blocking severity" — whatever is standing in the way of tomorrow's use is today's top priority.

The same goes for training: training isn't a lecture, it's running alongside someone — a week spent shadowing real tickets does more than a single class.

VII. Adoption Needs Measurement, and the Definitions Come First

Improving adoption needs a dashboard, and two metrics are enough.

Activation rate: what share of target users genuinely use the system each week — with three definitional elements to pin down: the denominator (who counts as a target user, using the headcount set in the Chapter 8 acceptance agreement), the window (weekly or monthly), and the criterion (does a login count, or does it require a real completed task).

Diversion rate: of a given category of task, how much of it has moved from the old path onto the new system — it's more honest than activation rate, because it measures not "was it opened" but "did the work actually get taken over." [1]

The discipline around definitions is the same as in Chapter 8: a pretty number under a vague definition is worth less than an ugly number under a clear one. "80% of users have used the system at some point" versus "60% of target users complete at least one real task in the system every week" — the second number is smaller, but it's the only one that can guide improvement.

[1] Fan Bing, Forward Deployed Engineer (FDE) (XDash open-source book)

This chapter has gone from causes of failure to incentives to gradual embedding, but it's really been saying the same thing the whole way through: a system entering an organization is a contest of interests and inertia.

And there's exactly one person who can change the rules of that contest: the executive sponsor. Which is why the next chapter is about executive sponsorship.

Chapter 16 · Why AI Adoption Has to Be an Executive-Sponsored Project

> He will win whose army is animated by the same spirit throughout all its ranks.
>
> — Sun Tzu, The Art of War, "Attack by Stratagem"

Start with a controlled comparison. The same AI adoption project, at two nearly identical companies, launched two different ways.

Company A had the technical department initiate the project: the proposal was tidy, the technology choices were sound. The project died in month four — the post-mortem traced the cause to needing sales-department data; the sales director's response was, this is your IT department's project, why would I grant you access.

Company B had its top executive initiate the project: at the kickoff meeting, the boss spent fifteen minutes explaining why this mattered to next year's business, and six months later the system was running inside the daily work of three departments.

The two companies' technical solutions were identical. The only difference: Company A's project had a good architect. Company B's project had someone who could mobilize the whole company with a signature.

The phrase "executive-sponsored project" has been worn smooth from overuse in China's enterprise software circles — enough that many practitioners flinch on reflex when they hear it. It sounds like a slogan for dodging blame: a project dies, and "there was no executive sponsorship" wraps the whole thing up. This chapter wants to restore those words from slogan back to structure: why is AI adoption almost inevitably an executive-sponsored project? What does the phrase actually mean? What does it look like when sponsorship is misapplied? And when doesn't it have to be the top executive at all?

I. The Cross-Departmental Problem

AI adoption has a trait other technology projects don't share: it is cross-departmental by nature.

Pull apart any real project and you'll find a map scattered across departments — data sits with Department A (customer information lives in the sales system, historical records with

operations), process sits with Department B (the approval workflow that needs reworking belongs to finance), budget sits with Department C (the project has to clear the IT budget, but the benefit shows up on the business side's books), and benefit sits with Department D (the efficiency gains get recorded in the frontline team's own report).

Can a mid-level project owner command all four at once? Not easily — not even one of them. Data access needs Department A's sign-off; reworking the process needs Department B to give up some of its authority; budget ownership needs Department C to accept the bill; confirming the benefit needs Department D's cooperation on measurement — every single one of these is a cross-departmental negotiation, and a mid-level manager only has "negotiate" in their toolkit, not "rule." The only position in the organization that holds authority over all four at once is the top executive.

This isn't theory — the field already has examples.

First, a counterexample: a major tech company's delivery team — a customer's business unit wanted a business agent but had no budget; the budget sat with the IT team, and IT and the business unit were in different departments — so the vendor had no choice but to design a solution that could "satisfy IT's technical sophistication and the business's outcome at the same time." What they found in the end was that "the sophistication IT wanted wasn't necessarily the outcome the business needed, and the outcome the business needed wasn't necessarily the most sophisticated approach" — no way to reconcile the relations of production, whichever way they turned it. [1]

A positive counterexample comes from a manufacturer: the chairman himself made a clear commitment to embracing AI, picked the sales function and a target region, and drove it top-down — finance, IT, and the sales line all moved together. The pilot started in the company's weakest-performing province and, six months later, that province had become the most advanced in the entire company, ready to be rolled out nationwide. Same equation, but whether the top executive is in it changes the answer completely.

It runs deeper than that. Inertia and self-interest are what have to be overcome, and the authority to change how people work — to overturn that inertia — likewise rests only with the top executive: performance metrics, headcount, the boundaries of departmental responsibility are all matters of corporate governance, not something any single business or technical department can move unilaterally. So this isn't a matter of posture — a plea to "get leadership to take it seriously" — it's a math problem about authority: the sum of authority required only comes together at the very peak of the management pyramid.

[1] Silicon Valley 101, Episode 248: "China-Style FDE, with Alibaba Lingyang's Peng Xinyu"

II. Fall Short of the Priority List, and the Road to Adoption Stops

So is it enough for any project to simply get an executive attached, and everything falls into place? No. Bob McGrew, an early Palantir executive, gave a sharper filter: your entry point has to be a problem leadership has already flagged as critical — ideally, one of the CEO's top five priorities. His reasoning is entirely practical — only a problem at that scale gives you enough force to grind through an organization's internal bureaucratic resistance; fall short of that priority level, and the customer organization has no reason to stay with you through the hard parts of getting the system adopted. [2]

The flip side of this standard matters just as much. As the vendor, the person you can actually reach is often an IT lead or some sponsor a few levels removed from the CEO — and that distance changes the nature of the problem. Whoever sits between you and the top has their own performance metrics to answer to, and would rather you solve something easy for them than touch something risky for them. So the project gets "downgraded" right at the source: what should have addressed the business's core pain point ends up as a safe, marginal, nobody-gets-offended demo project instead. This is exactly the organizational failure mode mentioned in Chapter 2 — it isn't that the problem wasn't worth solving, it's that you couldn't reach it.

A more complete chain of failure comes from the CEO of an enterprise software company: the board goes to the CEO and says "we need more AI," and the CEO says, fine — "I'll get a consultant to do more AI." The result is some centralized project nobody understands — detached from any specific business unit, operations never aligned. "Those things will fail." [3] Note the irony in this chain: it isn't a failure of the top executive's absence at all — it's a failure of the top executive showing up only at the start and being absent for the entire middle. The signature gets signed, and then what?

III. "None of the Three Understand": An Empty Intersection

Why do AI projects depend so heavily on the top executive to pull people to the same table? Because there's a structural hole inside the organization, summed up in a phrase that circulates through the industry: whoever understands the business doesn't understand AI, whoever

[2] Fan Bing, Forward Deployed Engineer (FDE) (XDash open-source book)

[3] a16z: Box CEO on AI agents and why enterprises can't keep up (YouTube)

[4] "Anyone Out There Actually Doing FDE Work?" (linux.do thread)

understands AI doesn't understand the business, and whoever understands both has no authority. [4]

This isn't a matter of capability — it's a matter of how the organization divides labor. A business expert's knowledge grew out of years on the ground; an AI team's ability grew out of a technical stack; two different languages, two different scorecards, two different circles — and the people who straddle both sides tend to sit low in the org chart in most organizations. They're too new, not yet risen to a position where they can command resources. With the intersection of all three empty, every project ends up running a "translation relay": the business explains the requirement to product, product translates it for engineering, engineering builds it and translates it back — and context is lost at every handoff (the same relay-loss from Chapter 6 replaying itself, unchanged, inside the organization).

Only the top executive can pin all three types of people to the same table at once. Pulling all sides together is what lets "what the business doesn't know" and "what the technical side doesn't know" cancel each other out inside the project — the business expert corrects course on the spot, the technical expert assesses feasibility on the spot, and questions of authority get settled on the spot. The reason an FDE is called a "forward-deployed engineer," the reason they have to be stationed at the customer's own site, is essentially to pull that translation-relay middle layer down onto a single table.

IV. What Executive Sponsorship Actually Means

Pull the slogan apart, and executive sponsorship in a real project takes the shape of three kinds of backing, none of them optional. [5]

Budget backing: only the top executive can rule on cost ownership for a cross-departmental project. The department that benefits from an AI project and the department that funds it are almost never the same — the money comes out of the IT budget, and the efficiency gains get credited to the business department's ledger. On any single department's books, a project like this looks like a loss; it only balances on the company's overall books. Who looks at the company's overall books? The top executive.

Incentive backing: the incentive problem discussed in Chapter 15 has no solution at the department level — the answer only exists at the top-executive level. Whether performance metrics change, whether headcount moves, whose fault an error is — these are acts of corporate governance, not something a project manager can promise. Without the top executive

[5] Crossroads × Rolling AI, "A Conversation on FDE" (podcast)

publicly standing behind it, the three failure modes from Chapter 15 will come true one by one.

Failure backing: allowing the first project to die meaningfully. An AI project's first delivery is, to a large degree, an exercise in organizational learning — learning how to work with an agent, learning how to measure, learning how to set acceptance criteria. Someone has to absorb the tuition for that first project on the organization's behalf; otherwise every department learns exactly one lesson — don't touch it, because touching it means the mistakes are yours alone.

There are real examples of all three kinds of backing.

One is a shift in how the demand side approaches things: enterprise top executives' attitude toward AI has changed — "night and day — it used to be all about how to use it, whether it could even be used, and now people skip straight past that to the next stage: when do we start actually putting it to work for me." The inflection point came once agents capable of executing tasks arrived — the nature of the question shifted from "should we" to "how fast." [1]

Another form of backing is the most concrete and closest to the ground: the chairman from the manufacturer in Section I personally taught more than fifteen sessions to the full workforce and management within a single year — "I learn it myself first, then I teach it to the employees, and only then can they understand what I'm thinking and actually use these systems well." [1] Shifting how people think isn't something a single kickoff rally can accomplish — it takes the top executive turning himself into the first instructor, and that single act covers all three kinds of backing at once: real, non-trivial time invested (budget), the most public form of standing behind it (incentives), and paying the tuition first, in person (failure).

Xiaopeng Lu (co-founder of Ventus AI, serving the North American healthcare industry) has shared his own observation: bosses who can genuinely drive AI adoption share exactly one trait — they're AI-native themselves, meaning they're heavy users of AI in their own right. His strategy is "win over the benchmark case first, then push it downward." [6]

This isn't an abstract question of attitude — in execution, it translates into real, measurable friction cost.

Lu compared two similarly sized clients: one boss's reasoning for driving adoption was "everyone else is pushing it, so I want to push it too"; the other's was "I'm certain AI needs to start now, and I'll invest whatever it takes." The difficulty of change management was completely different between the two — with firm, CEO-level support, things move far faster. [6]

[1] Silicon Valley 101, Episode 248: "China-Style FDE, with Alibaba Lingyang's Peng Xinyu"

[6] Tencent Research Institute, AI Lens Roundtable, Episode 6: "FDE Non-Consensus Views and a Field Guide from Silicon Valley Founders"

This adds a mechanistic layer to "three kinds of backing": budget backing and failure backing matter not because they're ceremonial gestures, but because how certain a boss is about "whether to do this" translates directly into the friction coefficient of the organization's execution — a boss following a trend can hand over a budget, but can't hand over the conviction it takes to pay the tuition and fail in public first.

Test "showing up at the kickoff rally" against these three, and it doesn't count as executive sponsorship. Showing up is a fifteen-minute act; the three kinds of backing have to run through the entire months-long span of adoption.

The test for telling real backing from fake is just as plain: when the project runs into the departmental wall in month four, does the top executive step in and rule on it, or say "you two go work it out among yourselves"?

V. A Catalog of Misaligned Sponsorship

Projects where the top executive is under-involved, or involved in the wrong way, tend to grow into a few typical shapes, each with its own tell. [5]

- **Technical-department-driven**: the system is the IT department's achievement, not the business's tool. Tell: the weekly project report is all technical milestones, not a single business metric. This is the second of the three failure modes from the last chapter, already unpacked.
- **Nominal sponsorship**: the top executive showed up at the kickoff meeting and never appeared again. Tell: when the project escalates to something needing cross-departmental judgment, the meeting minutes have no signature above department level. The matching tell on the vendor side: whoever you could reach during the sales pitch is unreachable by the time delivery starts.
- **Siloed**: the project closes its loop inside a single department; data, authority, and process can't reach beyond it. Tell: the system works great, but only for a thirty-person team, and expansion plans are perpetually "next quarter." Siloed is the most respectable of the four shapes — it's at least alive — but it never proves the project can scale.
- **Consulting-dependent**: exactly the "board → CEO → consulting firm → centralized big project" failure chain. [3] Tell: the project documentation keeps getting thicker while frontline usage records keep getting thinner.

[3] a16z: Box CEO on AI agents and why enterprises can't keep up (YouTube)

[5] Crossroads × Rolling AI, "A Conversation on FDE" (podcast)

This catalog isn't for labeling other people's projects — it's for checking the one in your own hands: if you're the customer-side project owner, which of these does your project look most like right now? If you're the vendor, which one is your contract actually signed against?

VI. Two Gates and a Flywheel

Executive sponsorship isn't a one-time act of "winning over leadership." It has two gates — clear the first and there's still a second — and once both gates are cleared, it can spin up into a flywheel (see Figure 16-1).

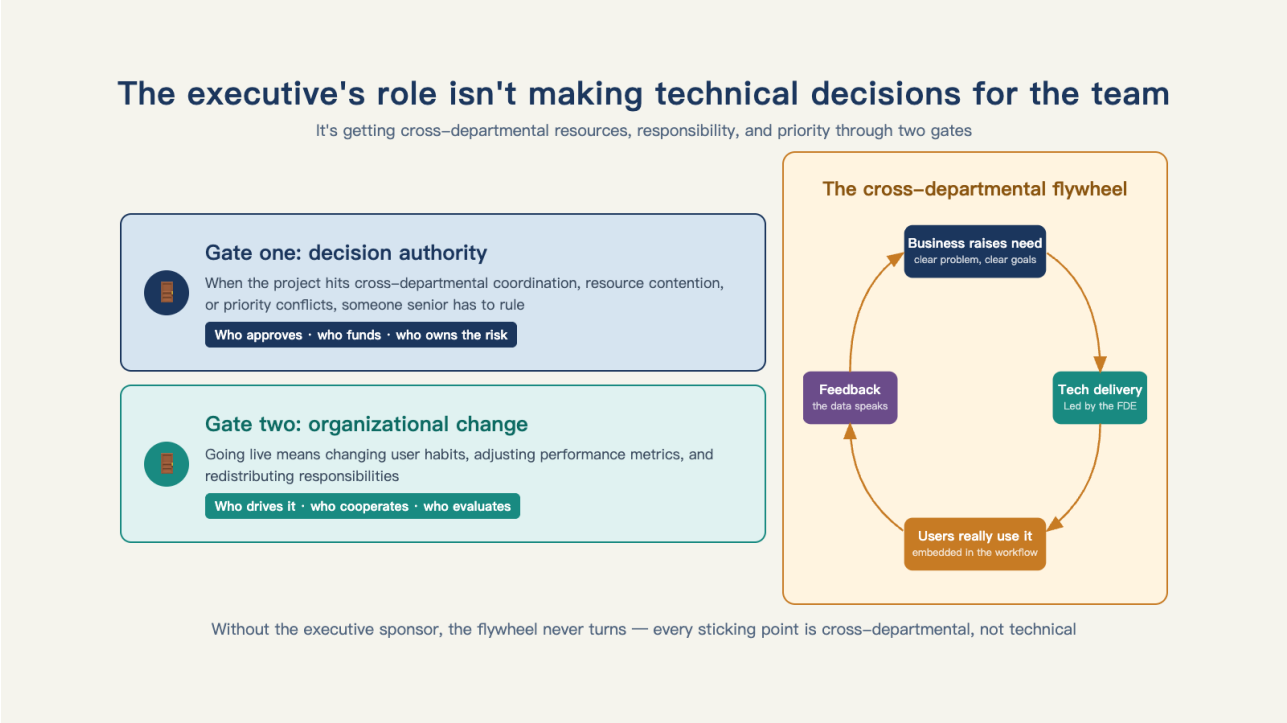

Figure 16-1: The top executive's role isn't making technical decisions for the team — it's getting cross-departmental resources, responsibility, and priority through two gates.

The first gate: **calibrate the top executive's expectations.** Going in, four things need to be laid out clearly: what AI can do, what it can't, how long before results show, and what has to be invested. Get expectations wrong, and once any single piece falls short after go-live, the boss's reaction is "I was misled" — one of the three failure modes is already buried at the door. A common calibration scenario: the boss opens with "cut headcount by thirty percent" or "replace three thousand people," and the conversation has to be pulled back to value right there on the spot — what AI replaces today is tasks, not jobs; a role only becomes replaceable once eighty

[7] "100 Questions About FDE" (open-source ebook)

[8] "AI Job Sense: Stop the Agent Rat Race"

percent or more of its core tasks have been taken over, and reality is nowhere close to that yet; trading "promise to cut three thousand jobs" for "deliver seven verifiable efficiency gains" is what lets both sides keep going. [7] Why is this gate hard to clear? Because most bosses' judgment of AI comes from a short video they scrolled past last night or a viral case study floating around online, not from systematic understanding — one practitioner said plainly that the biggest hidden cost in the adoption process is the boss's own cost of learning. [8] Calibrating expectations essentially means spending that cost of learning up front, at the gate, instead of letting it explode at acceptance.

The second gate: **defuse employee fear.** Chapter 15 devoted an entire section to unpacking this fear — it has no solution at the department level; what employees are watching for isn't what some project owner says, it's how the very top of the organization positions itself. Without the top executive stepping forward, this gate stays bolted from the inside. Three moves: don't let the system present itself as a replacement for people; let the first cohort of users become beneficiaries, then let those beneficiaries win over the ones still on the fence; and the most critical move of all — the top executive has to stand behind it publicly. Approving a budget privately does nothing; it has to be stated publicly that "this is so everyone can spend their time on more valuable work." [7] The difference between private support and public backing is one employees can tell apart perfectly well when they vote with their feet.

Once both gates are cleared, there's one more practical habit: pin a key-stakeholder map into the weekly report — who's a sponsor, who's the point of contact, who's potential resistance, and what each of them actually cares about, updated every week, turning the org chart from a static stack of business cards into a living map.

Does the work of clearing these gates have to start over from scratch on every project? One practitioner's approach turned it into compound interest: getting the first AI-illiterate boss on board is the most exhausting; every boss won over after that adds another layer to your material — training decks, alignment scripts, a list of common misconceptions — ready to reuse next time; and real problems worked out in the field also feed back into the head-office product, making the next customer easier to win over. So "the bosses keep turning over, and the playbook in your hands keeps getting thicker" isn't consolation — it's compounding: individual instinct accumulating, one project at a time, into an organizational asset. This skill itself is one of the assets a forward-deployed engineer carries away.

VII. A Two-Sided Tool

[7] "100 Questions About FDE" (open-source ebook)

This chapter closes with two checklists — one for the customer, one for the vendor.

A checklist for managing upward, for the customer-side project owner: ask the top executive for three things — a clear ruling on budget ownership (who pays, how it's split), explicit authorization (a written designation of the project owner and their cross-departmental authority), and incentive backing (a public statement, ideally paired with a change in performance metrics); report progress using the Chapter 8 acceptance vocabulary — business metrics, time window, judgment criteria — instead of a technical Gantt chart; sync progress with the top executive once a quarter: where does this project currently rank among their five priorities? If it falls out of the top five, either escalate the project back into rank, or wind it down early and gracefully.

A pre-sales due-diligence checklist for the vendor: what signals of executive involvement show up in the contract — who attends the kickoff meeting? Who signs off on the acceptance criteria? Does the authorization path for cross-departmental data access actually work? A contract negotiated only with the technical department should be priced with "this project will probably die in month four" baked into the risk (Chapter 20's pricing and Chapter 27's deal-screening will pick this thread back up). One more piece of well-meant advice: if due diligence turns up that the customer's top executive just wants "AI" printed in the annual report, the best deal you can make is not making this deal.

VIII. When It Doesn't Have to Be the Top Executive

Back to the instinctive flinch from the opening — when "executive sponsorship" gets sold as a cure-all, it really does obscure a truth one level down: the core of this rule isn't reverence for rank. It's **aligning authority with benefit**. [7]

When a project can close its loop entirely inside one department — data never leaves it, process never touches anyone else's territory — "top executive" can scale down to "process owner": whoever holds both the authority and the benefit for that one process. A shop-floor supervisor over the shop's scheduling system, a finance director over an invoice-review agent — both can be effective "department-level top executives."

So this chapter's conclusion comes down to one sentence: the minimum organizational condition for AI adoption isn't "there's a top executive" — it's "the reach of the authority is no smaller than the span of the problem." A problem crossing four departments needs authority that stands above all four; a problem crossing a single process only needs a process owner. Which one is

[7] "100 Questions About FDE" (open-source ebook)

the exception, and which the norm? So far, the four-department case remains the norm — because the problems worth an FDE's time are almost always cross-departmental.

The top executive gets you through the door, and keeps the project alive through adoption. But a living system eventually has to answer one last question: where does delivery actually end — at the signature on the acceptance form, or at the point where you can walk away with a clear conscience? Handoff is next.

Chapter 17 · How to Hand Off So the Customer Can Run It Themselves

> Unhappy the land that needs heroes.
>
> — Bertolt Brecht, Life of Galileo

When can an FDE actually leave? There are two ways to exit, and the outcomes couldn't be more different.

The first is a graceful exit: acceptance passes, the celebration dinner wraps up, the engineer withdraws. Three months later, nobody at the customer mentions the system anymore, and the monitoring dashboard is a wall of green — not because the system is running, but because nobody is using it anymore.

The second is also a graceful exit: the same process, but six months later, in the customer's internal quarterly meeting, the discussion is "let's have the agent take over two more steps next quarter" — nobody calls it "the AI project" anymore. It has grown into the work itself.

The two exits look identical on the day they happen. The difference only shows up later, and every action that decides which one you get happens before the exit itself. This chapter takes up delivery's final stretch: what does handing a system over for the customer to run on its own actually have to accomplish before delivery counts as done?

I. Delivery Has Three Stages — the Acceptance Sheet Only Covers One

Traditional project management defines "delivery complete" as one signed acceptance sheet. But delivery actually runs in three stages: **acceptance** — the system meets the agreed-upon criteria (Chapter 8); **adoption** — real users are actually using it (Chapter 15); **handoff** — the customer can run, maintain, and improve it on their own (this chapter). Only once all three stages are cleared does the capability actually stay inside the customer's organization; clear only the first, and what's left behind is a piece of software nobody is responsible for.

One long-time observer of this industry once wrote a line that sounds self-contradictory: the

[1] @deployengineer (Bhaulik Patel), essay series

best forward-deployed organizations work to eliminate their own jobs — building templates, building tools, turning one-off customer code into configurable platform capability; the goal isn't to erase the role, it's to move it toward a more valuable problem. [1] The line was originally about productization (unpacked in Chapter 21), but it holds up just as well applied to handoff — maybe even better: a delivery team that can never make itself unnecessary has turned every project into its own long-term meal ticket. That isn't dedication. It's failing to wean the customer off you.

The test for whether the customer has actually been weaned is simple: six months after you leave, do the system's small changes, small breakdowns, and new requirements still have to come back to you? If they do, it means the judgment and operational knowledge were never handed over. If they don't, then delivery is actually done.

II. The Three Exit Conditions

One practitioner summed up the conditions for a completed FDE exit as three things: business integration, knowledge governance, and system integration — "only once all three are done can he actually leave." [2]

The first: business integration. The system has to grow into the business process, stop being "that AI project." Three verifiable actions here: process documentation updated — the step the agent participates in gets written into the department's standard operating procedure, so replacing the system would require changing that document, and organizational inertia now protects the system instead of resisting it; embedded into daily use — the system gets used inside the interface users already work in (the embedding principle from Chapter 15), not as an island requiring a separate login; and metric ownership landed — the system's business metrics get recorded in the business department's own report, not in the IT department's project summary. The test for this item is the flip side of the line that opened this chapter: six months on, nobody says "this is the AI project" anymore. It's just part of the work.

The second: knowledge governance. The knowledge for running the system has to move out of the FDE's head and into the customer's organization. What has to move? Three assets, all of them outputs from earlier chapters: the operations manual (the one-page document from Chapter 14 — who watches which alert, which page to flip to when something goes wrong, when to hit the rollback); the eval set (that "compounding asset you can carry with you" from Chapter

[2] Crossroads × Rolling AI, "A Conversation on FDE" (podcast)

[3] Yilu Tongxing (一路瞳行), Episode 134: "From AI Assistant to Digital Employee"

12 — plus who maintains it: who adds new failure cases, how often regression gets run); and the incident playbook (the four-category failure triage from Chapter 13 — the first-response action for each type of incident). All three of these are line items on the handoff checklist, not the three words "we discussed it." The counterexample already showed up in Chapter 15: "the memory gets wiped and someone has to re-feed it" — operational knowledge that never gets written down degrades the moment the person who held it leaves. [3]

The third: system integration. Operations responsibility has to plug into the customer's own systems: monitoring alerts routed into the customer's own operations platform (not visible only to your company); upgrade paths made explicit — who approves changes, who executes them, where the maintenance window sits; and the vendor's boundary of responsibility written into the contract — who to call for what severity of incident, how fast the response time is, and under what conditions it's billable. This third item is the easiest one to skip when relations are good — and it's usually discovered the hard way, at 2 a.m. during the first real incident, that it was never agreed on at all.

What all three have in common: none of them is a technical action. All three are transfers of responsibility and knowledge. The technology was handed over long ago. The responsibility wasn't.

III. Delivery Is a Staircase Down, Not a Cliff Jump

The three exit conditions can't be checked off all at once on exit day — delivery has to descend a staircase, and every step has a duration and an exit criterion (see Figure 17-1). [2]

[2] Crossroads × Rolling AI, "A Conversation on FDE" (podcast)

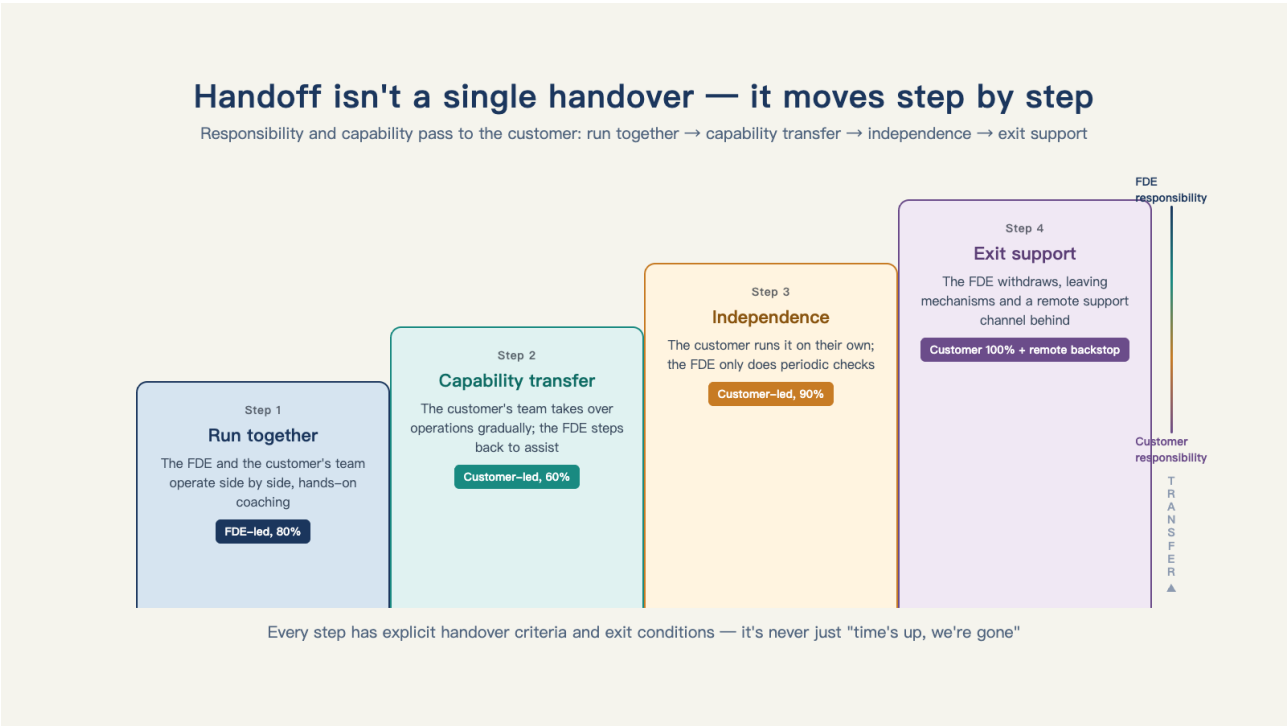

Figure 17–1: Handoff isn't a single handover — it's responsibility and capability moving to the customer one step at a time.

Step one: run alongside. The customer's team starts operating the system, with the FDE watching close by — errors get fixed, but the fixing happens where the customer can see it. By the end of this step, the customer can handle routine operations on their own. Exit criterion: a run of several consecutive weeks with no human intervention needed.

Step two: shadow. The customer operates, and the FDE only observes — recording, not touching, meeting once a week to compile a list of issues the customer missed on their own. What's really being delivered at this step is judgment: what counts as normal, what should raise a flag. Exit criterion: the customer's own decisions on handling anomalies match the FDE's after-the-fact judgment, several times running.

Step three: exit. Only escalation support remains — an agreed response tier, an agreed channel. Everything else belongs to the customer.

The staircase has a pitfall on each side: on one side is "never actually leaving" — every step gets extended indefinitely, shadowing turns into a permanent babysitter, and delivery has failed without anyone admitting it; on the other is "vanishing overnight" — jumping straight from acceptance to step three, so the customer faces their first real incident alone, with abandonment counting down from that day. How long to stay on each step, when to move to the next — that sense of pacing is exactly what separates a novice from a veteran.

IV. Three Wrong Turns

Handoff failure tends to take a few recurring shapes, each with its own correction.

Wrong turn one: handoff means sending documents. Package up the docs, get a signature, exit — three months later nobody has read past page two. Enterprise documentation that nobody reads is the default outcome, unless it does two things: gets organized around tasks rather than features ("how to handle an anomalous refund," not "refund module feature specification"), and gets embedded at the point of use rather than sitting on a shared drive (surfacing right where the user gets stuck). [4] A small test during the handoff window works better: have the customer's incoming owner walk you through incident handling out loud, while you watch — they talk, you listen, and wherever they can't get through it is exactly where the documentation has a hole.

Wrong turn two: withholding the eval set. The operations manual can be handed over verbatim, but the eval set often gets kept back — it's an asset, and it's leverage for the next deal. This choice has to be settled at contract time, not haggled over at exit: hand over the eval set along with the responsibility for maintaining it, and the customer gets the ability to keep improving on their own while the vendor gets a reputation for "handing off cleanly" — this trade gets tallied again in Chapter 20's pricing models. The position here is worth stating plainly: a team that clings to the eval set is, at bottom, valuing "the right to keep charging" over "the obligation to finish delivery" — and the customer will eventually feel it.

Wrong turn three: handing over operation without handing over judgment. The customer can push the buttons, but can't adjust the parameters: whether to move a threshold, how to attribute a new failure case, whether to trigger a rollback — every small question ends up as another purchased service call. This one is the most hidden, because it doesn't show up during the contract term at all, and can even look great on revenue. There's a ready-made example of how to break it: a partnership between an AI company and a fintech firm wrote "train the trainers" into its core design — identifying the motivated people inside the customer's organization and developing them into internal instructors and internal go-to experts, giving them official certification, a dedicated support channel, and visibility in front of executives; the partnership's stated goal in one line: "transfer knowledge so the customer can build and extend their own agents independently." Delivered all the way to the end, that's teaching the customer to teach themselves. [4]

V. After Handoff: Five Mirrors

[4] Fan Bing, Forward Deployed Engineer (FDE) (XDash open-source book)

Exiting isn't the finish line. The quality of a handoff only becomes visible a year or two after you leave, looking back at it in the mirror. The reasons enterprise customers churn can be grouped into five categories, and every one of them traces back to a debt left unpaid at handoff.

- **Value evaporation**: the system is still running, but nobody remembers what problem it solved — the value narrative from when the project was launched has gone dark, and a newly arrived manager only sees a line-item for operations spend. The countermeasure belongs on the handoff checklist: hand over the value framework and the responsibility for measuring it along with the system, so there's always someone who can answer "what is this system actually worth."
- **Sponsor departure**: your internal ally gets promoted or transferred, their successor never lived through the original decision, and the system becomes "the previous guy's legacy project." Countermeasures: build a mesh of relationships (develop at least two more relationship lines beyond your one sponsor), and institutionalize the value (write the system into process documents and collective memory). The handoff window is exactly the window for building that mesh.
- **Quality drift**: the business changes, the data changes, the model gets updated, and output quality slides slowly downward until management notices too late. Countermeasure: this is where handing over the eval set really earns its keep — regression tests run by the customer themselves, so the moment quality starts to drift, the customer is the first one to catch it.
- **Cost backlash**: heavier usage means a bigger bill, and finance scrutinizes it harder. Countermeasure: hand over Chapter 14's cost dashboard and budget circuit breaker along with the system, so the customer can read it themselves and pull the brake themselves.
- **Vendor withdrawal syndrome**: the customer's own wariness about being "locked in" — the consulting industry has even forecast that most enterprises will eventually abandon this kind of solution because of high cost and insufficient internal skill. [4] The countermeasure here is, precisely, a thorough handoff: what the customer fears is "being unable to live without you," not "you being present."

All five mirrors reflect the same conclusion: handoff isn't the tail end of delivery. It's delivery quality's own acceptance sheet — every debt you left unpaid while you were on site comes due, with interest, after you leave.

VI. When You Genuinely Can't Exit

[2] Crossroads × Rolling AI, "A Conversation on FDE" (podcast)

[4] Fan Bing, Forward Deployed Engineer (FDE) (XDash open-source book)

Honestly, some engagements can't be exited: the customer either has no will to take over, or no ability to. What to do here isn't to force an exit (the system dies) or to linger indefinitely (a dependency on billable days) — it's to write that ambiguous state explicitly into the contract: managed operation, priced openly as its own line of service. The customer knows clearly they're buying ongoing managed operation, not completed delivery; the vendor knows clearly they're being paid for operations, not for delivery; and neither side has any illusion about who actually owns the system. [2]

The difference between managed operation and a faked handoff is, commercially, one contract, and ethically, one honest sentence. Under managed operation, the customer can decide at any time to "build our own team and take it back"; under a faked handoff, the customer thinks they own it when they're really only renting it — and on the day they find out the truth, what they lose isn't just one system, it's their trust in everything else you've ever promised.

One last note, from a business angle, bridging toward Part Five: the ability to hand off is itself a prerequisite for productization. Only a team that can hand off cleanly earns the right to talk about scaling and replicating — because every item on the handoff checklist (templates, manuals, eval sets, connectors) doubles as launch material for the next customer. A team that can't hand off is stacking billable days on top of billable days on every project, and it stops the moment the team hits its headcount ceiling. [1] This logic gets built out into the full "declining customization" argument in Chapter 21.

Exit day can be a short meeting with a three-line agenda: run through the checklist for all three exit conditions item by item, with a customer-side signatory for each; finalize the weekly-report template for the shadow-run period; and schedule the next "escalation-only" meeting for six months out. Only once that meeting ends does the person actually leave.

This chapter has described the ideal case: the customer wants the system, and the organization cooperates with handoff. But turn the lens toward the Chinese market, and procurement structure, trust structure, and who defines the last mile all look different — adoption and handoff grow in a different soil altogether.

The next chapter takes up China's different playbook.

---

[1] @deployengineer (Bhaulik Patel), essay series

Chapter 18 · Why China Plays by Different Rules

> The map is not the territory.
>
> — Alfred Korzybski

Start with two people doing the same FDE job, and look at where a day's working hours actually go.

The first comes from Silicon Valley. An FDE at a model-platform company spends about 75% of his time on software engineering and model optimization, with consulting and customer relationships splitting the rest (the detailed time ledger is in Chapter 1). [1]

The second comes from China. A consultant doing enterprise AI adoption inside China ran the same accounting: roughly 60% of his time goes to helping customers untangle business problems, or simply to building relationships — maintaining connections, chatting, dining out, digging for requirements. Technology gets the small sliver that's left. [2]

Put the two side by side, and the point isn't which is better. It's that the same job title, FDE, is two different jobs in two different markets.

The Silicon Valley FDE assumes a world that is already in place — requirements are clear, data is online, the payment mechanisms are mature — so three-quarters of his time can naturally go into engineering. The practitioner in China faces a different world: requirements get dug out over dinner, data gets hand-copied out of notebooks, payment gets counted by head. Transplant the Silicon Valley playbook unchanged, and the more faithfully you copy it, the faster you die.

This chapter takes up three questions: which structures the differences actually sit in, how the playbook has to change, and what native forms the Chinese market has grown of its own.

I. Different Structure, Different Playbook

[1] Baseten, "What I Learned as a Forward-Deployed Engineer Working at an AI Startup" (Het Trivedi)

[2] Turning Point (破局点), Episode 31: "FDE, Chinese Style"

[3] "100 Questions About FDE" (open-source ebook)

The difference isn't technology — the model is the same model. The difference is structure, and it comes down to four things (see Figure 18–1). [3]

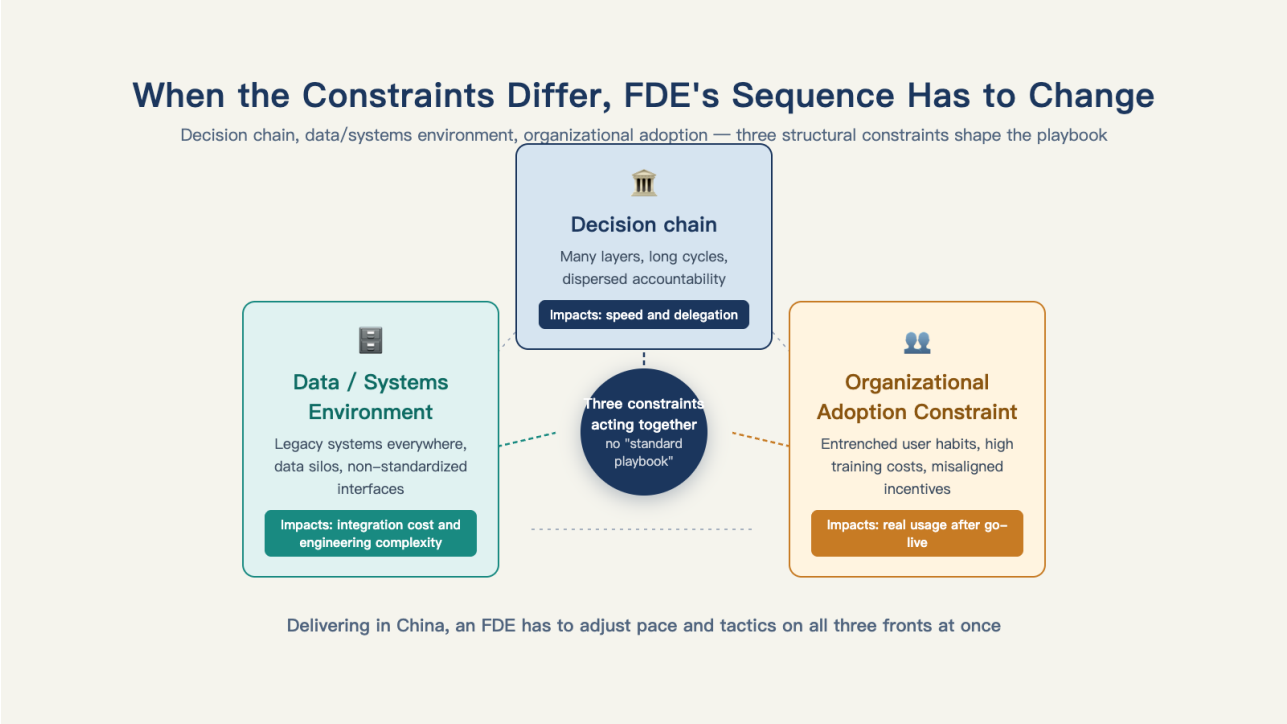

Figure 18–1: When the constraints on decisions, the data environment, and organizational adoption differ, FDE's sequence has to adapt too.

First: procurement and budget. The US market accepts subscriptions, usage–based billing, even outcome sharing; the corporate muscle memory of Chinese enterprises is "buy it outright and accept it as a project": annual budgets, project initiation and tendering, acceptance payments in installments. How the playbook adapts: slice the Minimum Viable Deployment into the acceptance milestones that already exist — the acceptance contract from Chapter 8 (business metrics, time window, judgment criteria) happens to look exactly like a project–style acceptance sheet, so push with the current rather than against it. Copy the subscription model as–is, and you die first in the procurement process.

Second: data constraints. "Data never leaves the building" is not a preference for Chinese customers — it is often a hard constraint, especially in state–owned and sensitive industries. How the playbook adapts: the architecture decisions around private and on–premises deployment have to move up into the design phase, not be left to the remediation phase. Field evidence: one team, constrained by budget, cold–started with a locally deployed open–source model plus retrieval — localization under cost constraints isn't a China–specific sideshow; it's the norm. [4]

[4] "AI Job Sense: Stop the Agent Rat Race"

Third: payment habits. Paying by the day is a market that has been mature for a century; paying for results has no line item in the procurement catalog. How the playbook adapts: the education cost of outcome-based pricing has to be built into your pricing (Chapter 20 covers how to run that calculation), or you transition through a hybrid of "stage acceptance plus value metrics written into the contract."

Fourth: talent supply. For hybrid talent who understand both the business and AI, the market consensus is "they're either expensive or they're not good; most companies have to grow their own." [5] How the playbook adapts: internal transfers plus on-site practice — the big tech companies routinely move product managers and frontend engineers into forward-deployment roles and send them to customer sites to gather requirements and produce prototypes.

Dig one level beneath the four, and there's a shared foundation problem. Peng Xinyu, CEO of Alibaba Lingyang, whose team serves several hundred to a thousand-plus mid-sized and large customers, offers this judgment: American companies came through their informatization era with companies like Salesforce and SAP, which "laid down many of the workflows and set many of the data standards"; China lacks that layer, and "not one of them has a reasonably stable foundation" — so delivery in China often means "driving piles and raising the building from day one, then doing the interior finishing — the whole string of things has to get done." [6] This one judgment explains several of this chapter's observations at once: data constraints are tight because the data foundation has to be poured on the spot; relationship-building eats so much of the calendar because there are no standard workflows to lean on, so requirements can only be mined out of relationships; payment recognizes person-days because "every large enterprise in China has an IT team" — pure feature-integration delivery is something the customer could do in-house, so the vendor has to create value beyond the internal team before anyone will pay for it.

II. Who Defines the Last Mile

Above and beyond the four differences hangs a more fundamental question: **who defines the last mile?**

On the same project, some people think the last mile runs from demo to go-live, while others think it doesn't end until frontline employees actually use the thing — a gap of several times over. Who gets to define the finish line directly decides whether the FDE is doing technical work

[5] "Anyone Out There Actually Doing FDE Work?" (linux.do thread)

[6] Silicon Valley 101, Episode 248: "China-Style FDE, with Alibaba Lingyang's Peng Xinyu"

or wrangling disputes. In the Silicon Valley narrative, the FDE defines that mile himself: he's stationed at the customer's site, watches people work, connects the broken links with his own hands, then carries the experience back to headquarters. In China, that defining right often isn't in the FDE's hands at all.

One real case reported by practitioners: at a state-owned enterprise, the contract was signed by the top boss, but the people executing on the ground couldn't articulate requirements — so they let the FDE find them, propose them, and build them himself. Seven agents got built. Nobody used any of them. The last mile had been stretched into a swamp, because from start to finish nobody ever said clearly where the finish line was. The chain often has middlemen in it too: plenty of domestic projects are taken on by a cloud vendor's resellers, and requirements reach the work site through several layers — each one amplifying and distorting the promises. [3]

The play for the fight over definition isn't confrontation. It's turning "definition" from something spoken into something written: use the Chapter 8 acceptance contract to put "what does finished look like" onto a single page both parties sign — and the question of who decides where the finish line is becomes a question of what the finish line says in black and white. Until the signature, you're working toward someone else's expectations — ones that don't match your own.

III. Two Soils, Two Logics of Advance

When the structural differences land on a specific customer, another feature of the Chinese market shows up: it runs on two tracks. Private companies and state-owned companies are two entirely different soils.

The private-company logic is "get the thing done." The owner cares about exactly one question: I spend a few million — does revenue go up, or does cost come down? Whether AI is involved, he doesn't care. The play: go straight to the person who signs off, let results do the talking, and don't take detours.

The state-owned logic is "get the people sorted." The contest over authority and accountability inside the organization is ten times more complicated than the technical solution, and fear of making a mistake is the background color. The play: first map out who is responsible and who takes the blame; translate project progress into language your point of contact can carry upstairs and report with — give him material to show for himself, rather than leaping over him and making the decision for the boss; and at the same time, list the small things that can move

[3] "100 Questions About FDE" (open-source ebook)

this week, banking whatever inch you can. One line above all: **push the process forward, but never make the final call for someone else** — unclear authority and accountability is the customer's own ailment, and you're in no position to operate. Step outside your lane and absorb that responsibility, and if the project dies, the blame is entirely yours.

Mixing the two logics is where things break. One move works on both sides: get in front of the person who signs the contract — the closer to the source, the truer the requirements. [3]

Ten minutes spent judging the soil before you enter is worth more than three months spent remediating afterward.

IV. Field Observations from the Chinese Ground

The practitioner who described spending "60% of his time on relationships" has three first-hand observations. [2]

Observation one: build the data first, talk contracts second. "In a lot of under-the-radar businesses, there is no data — customer information lives in a notebook." The Data Contract in Chapter 10 assumes "the customer has data to connect to" — on the ground in China you often have to take one earlier step: first transcribe what's in the notebook into a system, first draw the veteran's in-head process into a diagram. Only then is there a contract to talk about.

Observation two: the last mile jams in the most mundane, unglamorous places. One store customer was still running an old XP machine that couldn't install new software; a fully configured bot "went on strike" one day, and after half a day of digging, the cause turned out to be that someone had switched the computer off. The first technical decision in a deployment plan often isn't which model to pick — it's which year's operating system you need to be compatible with.

Observation three: relationships are infrastructure. Look again at that 60% of time spent on relationships: connections aren't lubricant, they're load-bearing walls — requirements are mined out of relationships, and trust grows across the dinner table (this time cost has to be priced into cost estimates and quotes from the start; Chapter 27's deal screening and pricing will put it to work). Where your trust comes from — the endorsement of a major company, past cases, a personal brand, or referral by people who know you — directly determines what kind of practitioner you are.

[2] Turning Point (破局点), Episode 31: "FDE, Chinese Style"

[3] "100 Questions About FDE" (open-source ebook)

[7] Tencent Research Institute, AI Lens Roundtable, Episode 6: "FDE Non-Consensus Views and a Field Guide from Silicon Valley Founders"

Xiaopeng Lu is one of the very few who has done FDE work on the ground on both sides — in the US he founded a company doing North American healthcare AI adoption (Ventus AI), and before that he worked back in China for two years, where he saw the domestic version of the role up close. His summary of the domestic customer's state of mind stings: "two or three people tinkering could put something like this together too" — customers will even ask point-blank: "you're giving me two people on the project — why shouldn't I just hire two people myself and manage them?" [7]

That rhetorical question and this chapter's third structural difference (payment habits) are saying the same thing: the Chinese customer assumes by default that an external team is selling "person-days," and if person-days are affordable, there's no reason to outsource. Paying for results, for sustained judgment — that itself requires educating the market first. This isn't Chinese customers being irrational; it's a perception gap between the two markets over "what an external team is actually selling" that hasn't been filled in yet.

Looking at the same question across both the US and China actually makes it easier to see clearly: this isn't one market's defect. Low-teachability complexity — complicated problems that are very hard to hand over for the customer to run themselves — is a line of business that has to clear this exact same first gate in any market where the trust structure isn't yet mature.

V. The Answers from Those Who Went First

The Chinese market is not without its answers. Line up the pioneers with public records, and two clear paths appear: how a service provider productizes itself, and how the big companies turn this into an institution.

Productization for a service provider works the same problem on two levels: first set boundaries, then define stages. One domestic provider in the real-estate and facilities-management industry draws its line against on-site outsourcing in three sentences on its own website: delivery by stage, acceptance by result, not settlement by hours worked; engineering done with a formed product in hand, not written and patched from scratch on site; we leave when it's done, with the capability deposited in the system and in the customer's team, not staying ever longer. It has even built a mechanism for turning away the wrong business — no unvalidated scenarios, no engagements where the data could be exported in a single spreadsheet, no customers who just want to learn about AI. Three sentences plus a set of turn-aways is just about Part Two and Part Four of this book, turned into advertising copy.

[8] Fan Bing, Forward Deployed Engineer (FDE) (XDash open-source book)

Once the boundaries hold, the next step is turning the methodology into replicable stages: one financial-services provider has settled its delivery into three phases — scenario diagnosis, one to two weeks (identify the three to five highest-value scenarios, quantify the return on investment) → embedded delivery, eight to sixteen weeks (from model selection to production integration) → production deployment and continuous evolution (iterating with the business) — while making "data never leaves the domain" explicit. [8] Notice the structure: the diagnosis phase is Chapter 6's discovery and the five filters, the delivery phase is Chapter 7's Minimum Viable Deployment, the evolution phase is this chapter's after-delivery — the methodology carries straight across; it just grew on China's procurement rhythm.

The big companies take a different road: not persuading customers with boundaries and stages, but with institutionalization itself. Volcano Engine has stood up dedicated FDE teams; its president, in a public interview, described the role in words that nearly match this book's definition verbatim — "not sales, and not pre-sales; they must have a strong ability to land technology in production" — and team members are deliberately staffed with diverse industry backgrounds. [8] The head of efficiency engineering at the e-commerce company Dewu broke an AI transformation ten thousand people strong into three steps: first, company-wide consensus, lowering the threshold so people actually start using it; then, sorting scenarios into quadrants by tolerance for error, handing the higher-tolerance ones to agents first; and finally, standing up a knowledge-operations group to move experts' tacit experience into space the agents can see — consensus, scenarios, knowledge, in that order and no other.

The deepest sample of institutionalization may be the team led by Peng Xinyu, CEO of Alibaba Lingyang. As the founder of Alibaba's data middle platform methodology, his team was called CSM (customer success) before the word "FDE" became fashionable, and had done it for five years, first serving several hundred to a thousand-plus mid-sized and large customers on the strength of its data-platform business — in his own words: "before the term FDE existed, this is what we were already doing... we've had a team called CSM for five years now." [6] They have condensed the practice into three principles: oriented by business outcomes (the problem must come with a business goal, business numbers, and historical records, such that an eval set and a metrics system can be built), founded on enterprise data as the base layer, and defaulting to the benchmark performer — the starting persona for an AI customer-service agent is "a rep who has worked the seat for five to ten years." [6]

Down to the concrete case: the "shipment-chasing" after-sales flow at a consumer-electronics customer came apart into more than 260 steps — three kinds of chasing (a whole unit

[6] Silicon Valley 101, Episode 248: "China-Style FDE, with Alibaba Lingyang's Peng Xinyu"

[8] Fan Bing, Forward Deployed Engineer (FDE) (XDash open-source book)

under-shipped, an accessory missing, a gift not yet arrived), internal and external systems tangled together and spanning JD, Tmall, and several other platforms, covering over 95% of scenarios; meanwhile the human agents answering today respond "on the order of hours or days," and customers who can't wait simply cancel the order. [6] They also give a copy-ready three-question standard for choosing scenarios: where is the most people-time burned (phone calls made and taken, complaints handled), where the most money (payouts and loss control, marketing spend), where the most clock time (shipment-chasing inquiries) — those three questions are saying almost exactly what Chapter 6's value-density filter says. Organizationally, their answer is "our approach isn't one person, it's an organization": business analysts oriented by business outcomes (industry experts in beauty, appliances, automotive — responsible for orchestrating the workflow and setting the evaluation standards), AI architects (translating business problems into which model to use and which steps a human intervenes in versus which the machine takes over), and a coaching team drawn from the customer's own best service reps and salespeople — speaking directly to Chapter 24's capability model and Chapter 26's team building.

Behind all of these answers presses one shared financial fact: in their 2025 annual reports, Yonyou's revenue per employee was about 480,000 yuan and Kingdee's about 620,000; in the same year, Palantir's revenue per employee was about a million US dollars — an order-of-magnitude gap in per-person productivity. [8] This is what "the curse of the project model" looks like on a financial statement: the big customers demand customization, the vendor loses money on every engagement, the code gets abandoned when the delivery team walks, and the very best engineers are consumed by heavily customized, non-standard delivery. The same set of numbers also contains the Chinese market's biggest upside for this business: whoever rewrites the delivery engineer's output from "person-day" pricing to "results" pricing rewrites, at the same stroke, the market price of hundreds of thousands of people and the financial structure of the business itself.

The demand side deserves a hearing too. A targeted survey of enterprise CIOs put three numbers side by side: 73.3% of the CIOs surveyed endorse the forward-deployment model and are willing to pilot it; at the same time, 60.0% worry that vendors don't understand their industry well enough, and 53.3% believe the market lacks quantifiable proof of input-output return. [8] Translated into one sentence: the endorsement is real, and so are the reservations — customers aren't hesitating over whether to buy a new job title; they're evaluating an entire value-realization mechanism that hasn't been proven yet. And that list of reservations — industry understanding, quantifiable input-output — is exactly what this book's Chapter 6 and Chapter 8

[8] Fan Bing, Forward Deployed Engineer (FDE) (XDash open-source book)

take up.

VI. Two Native Forms

Under these structural constraints, forward deployment in China has grown two forms of its own — both differ from the Silicon Valley original's responsibility structure, and both genuinely run inside their own structure.

Form one: the big-company on-site prototype engineer. As a frontline practitioner in the community describes it: the company sends product-plus-frontend engineers "rolled into one" to the customer's site to gather requirements and produce prototypes on the spot with AI — frontend pages on pure demo data, revised on-site to the customer's requests; once requirements are settled, they go back to the in-house backend developers for implementation and integration. [5] Run it against the three tests from Chapter 2, and it covers the "discovery + prototype" stretch of the spectrum — production and the feedback loop live somewhere else. Under China's procurement structure (reseller layers in between, vague requirements, acceptance-driven sign-off), this is a rational division of labor — the prototype is the fastest lever for prying a requirements confirmation out of the customer, and it happens to route around the fights over who defines the last mile: put the prototype on the table, revise it three rounds, and the requirements have defined themselves.

Form two: the independent FDE and independent service providers. In another community practitioner's own account: three years building custom agents, most of it taking orders online, with a self-built case library and reusable modules — but "work for yourself and you have no endorsement; work for someone else and it's misery." [9] That sentence puts its finger on the two real constraints of the independent form: the credibility cost of personal endorsement (no big-company halo to open the first door for you), and the incentive drain of the employment form. Its mature shape is the "enterprise AI adoption services" business run by people whose main axis is consulting and sales ability, focused on mid-sized companies, replicating a case library up and down one industry — engineering ranks third, behind consulting and sales. It isn't that engineering doesn't matter; it's that in the Chinese market, digging out a problem worth solving and closing the deal are scarcer than standing up the system. [2] Chapter 27 develops this form in full (whether an individual can do FDE independently).

[2] Turning Point (破局点), Episode 31: "FDE, Chinese Style"

[5] "Anyone Out There Actually Doing FDE Work?" (linux.do thread)

[9] "Doing FDE Work in Wuhan, Looking for a Partner" (linux.do thread)

Put the two forms back against the responsibilities from Chapter 1, and the conclusion is this: the Chinese sample is generally narrower than the Silicon Valley original — but "narrower" doesn't mean "wrong." Form is a function of constraint: if the procurement structure means the only requirements that reach you are prototype requests, then take the prototype to its highest polish; if the trust structure requires relationship-building first, then turn relationship-building itself into a methodology. Before judging any form better or worse, see clearly which constraint it is answering.

VII. A Stress Test Under Tighter Constraints

On that premise, this chapter's conclusion comes in three sentences.

First: the Chinese market isn't FDE at lower specs — it's a different way of surviving, forced out by a different set of constraints. The procurement structure means you can often only start with a prototype; the trust structure means that without a big-company endorsement, you accumulate through relationships and cases, bit by bit. The on-site prototype engineer and the independent FDE are not "settling for second best" — they are plays that can stand upright under the constraints each of them actually faces.

Second: China's constraints aren't "one more item" — they're "tighter." The procurement rules, the state of the data, the payment habits, who owns the definition of done, the investment in relationships — everything this chapter has walked through adds up to one sentence: the same thing takes several more passes of work in China before it holds up. The methods in Part Two and Part Four are bonus points in a market with loose constraints; under constraints this tight, they are mandatory questions you can die for leaving unanswered.

Third, and to every reader: treat the Chinese sample as a stress test. If a delivery method still holds under the constraints of the Chinese ground — still finds problems worth solving, still pulls the acceptance criteria into a defensible definition, still manages to exit cleanly — then in markets with looser constraints, it will very likely hold even better.

Organization, adoption, handoff — all discussed. But hanging above all of it is a more fundamental question: with a delivery burden this heavy, does this business actually hold up as a business model? Part Five runs the numbers.

Part Five · How Does This Business Actually Work?

The first three parts covered getting the system built, stable, and adopted. This part comes back to the coldest question of all: what makes any of this a business, rather than a high-end outsourcing team?

Four chapters work four sides of the same ledger: Chapter 19 works the master ledger — hiring FDEs is a company-level P&L decision, not a staffing decision, and if the big numbers don't add up, don't proceed; Chapter 20 works the per-deal ledger — behind the three pricing models (by the day, fixed scope, by result) sits a question of who carries the risk, and all three pricing formulas seen in the Chinese market anchor to numbers the customer can already verify on their own books; Chapter 21 works the curve — the Nth customer's customization volume has to decline, or gross margin, speed, and quality all degrade linearly with customer count, and propping the curve up with heroics is the most expensive fix there is; Chapter 22 works the mechanism — turning field experience into product capability is an interface that has to be deliberately engineered, and the FDE's job is to carry that experience back to the product team, functioning as the product's second customer.

Written for managers deciding whether to build a forward-deployed team, and for founders already wrestling with the numbers: every chapter in this part might well conclude "don't do this" — that is the most valuable output these numbers can produce.

Chapter 19 · Why "Hire a Few FDEs" Is a Big-Ledger Decision, Not a Small One

> You should invest in a business that even a fool can run, because someday a fool will.
> — Warren Buffett

"This customer matters — let's hire two FDEs first."

In a project meeting, that sentence sounds almost unobjectionable. The customer needs a deep deployment, sales doesn't want to lose the deal, and the existing engineering team has no spare capacity. Two people who can sit on-site and put out fires look like the fastest fix.

But once the hiring decision is made, the question stops being just "two more salaries." When the next customer shows up, do you hire two more people again? Can the code they wrote, the processes they got running, and the experience they built up for this customer carry over to the next project?

"Hire a few FDEs" looks like a staffing question on the surface. What it's actually deciding is what the company will grow on going forward. This chapter works exactly that ledger.

I. Sort the Small Ledger from the Big Ledger Before You Write Anything Down

The small ledger is the hiring lens: what's this person's background, how capable are they, what salary do they need, can they handle being on-site. The big ledger is the input-output lens: what's this team's input-output structure? How does it change as the customer count grows? What does it turn the organization into?

Lined up side by side, the difference is obvious. The small ledger asks, "what does this person cost per year"; the big ledger asks, "for every additional customer, how much does cost rise and how does gross margin change." The small ledger asks, "can he handle the customer"; the big ledger asks, "can what he built for this customer be used by the next one." The small ledger asks, "how many people does the team need"; the big ledger asks, "does team size have to grow linearly with the number of customers."

This book's position is clear: answer the big ledger first, the small ledger second. If the big

ledger doesn't add up, no matter how well you answer the small ledger, you're just hiring people into the wrong organization. This is also the prerequisite for Chapter 26 (when and how to build the team) — every detail of organizational design grows out of the ledger in this chapter.

II. Six Metrics That Put the Big Ledger into Numbers

The big ledger can't just be a posture — it has to be calculable. Six metrics, each clarifying two things: how to calculate it, and what a healthy number looks like. [1]

First, Deployment Leverage: how much customer revenue can one FDE get into production? The numerator is customer revenue newly reaching production; the denominator is the number of engineers deployed on it. It answers this business's most basic question: how much can one person carry?

Second, Time to Value: how long from signing to the customer's first measurable business result? A reference range: validating a Minimum Viable Deployment runs in weeks (2—6 weeks), a full deployment in months (1—4 months); if this period keeps stretching out, that's the first sign something's wrong with the methodology or the platform. [2]

Third, Cost to Serve: how much engineering effort do the deployment phase and the post-launch period each require? This one has to be counted in full — hidden items like travel, support, and rework are the easiest to miss.

Fourth, Expansion Effect: do customers with an FDE involved grow faster than those without one? Of an existing customer's new revenue, how much requires no new delivery work? This one tests the logic that "deep service buys expanded trust."

Fifth, Product Leverage: how much of the work done for one customer turns into a capability that later customers can reuse? If a platform component saves two months on every future deployment, it's creating leverage at the company level.

Sixth, and the one to watch most closely: the Customization Decline Rate. The Nth customer's customization volume should be significantly lower than the first customer's — if it doesn't drop for three customers running, check the product feedback mechanism immediately. Chapter 21 works through this one in depth; for now, just get a first impression of it.

You don't need to track all six at once — watching the three to five most relevant to your current

[1] @deployengineer (Bhaulik Patel), essay series

[2] Fan Bing, Forward Deployed Engineer (FDE) (XDash open-source book)

stage beats a dashboard that covers a whole wall. [2] But an organization that watches none of them is doing the small ledger with its eyes closed (see Figure 19–1).

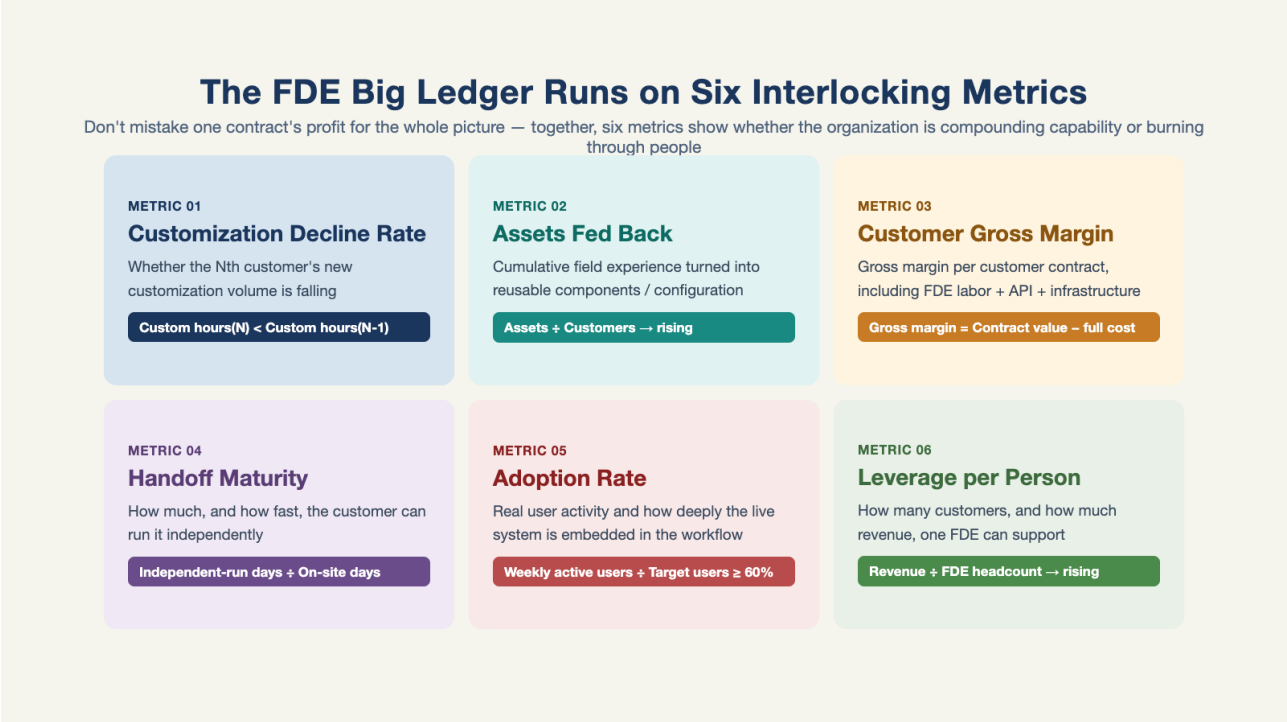

Figure 19–1: The FDE big ledger runs on six interlocking metrics, not just new revenue.

III. The Cost of Heroics

Of the six metrics, Deployment Leverage is the first to expose "heroics" for what it is.

What's a heroic FDE: extremely capable individually, carrying everything alone — give him the hardest deal, the hardest customer, the firefighting. At ten customers, the hero is an asset to the company: his name is the word–of–mouth, and his presence alone brings three deals back from the dead. But at a hundred customers, the hero is a disaster for the company — because organizational capability lives in him personally, not in process or product: he can't carry a hundred customers, he can't teach a hundred engineers (the knowledge is in his head), and when he leaves, a corner of the company caves in.

A deeper trap than the individual hero is hero culture. A forward–deployed team with no boundaries becomes the default dumping ground for "everything in the organization that doesn't fit anywhere else": sales shoves in features it already promised, product shoves in gaps it can't fill, engineering shoves in integrations it doesn't want to build, customer success shoves in escalated complaints — soon, every kind of work gets dumped here, and in the end it's all held up by a few heroes: a handful of standout engineers keeping customers alive through sheer

force, with the business depending on people instead of systems. It runs fine at ten customers. At a hundred, it seizes up.

The fix is in Chapter 22 — turning individual capability into product capability — but for now, note this down: **a hero is a linear-growth asset, and this business either compounds or decays; there's no linear gear.** One practitioner put it as clearly as it gets: the best forward-deployed organizations work to eliminate their own jobs — building templates, building tools, turning customer-specific code into configurable platform capability, and continuously pushing the role toward higher-value problems (see Figure 19-2). [1]

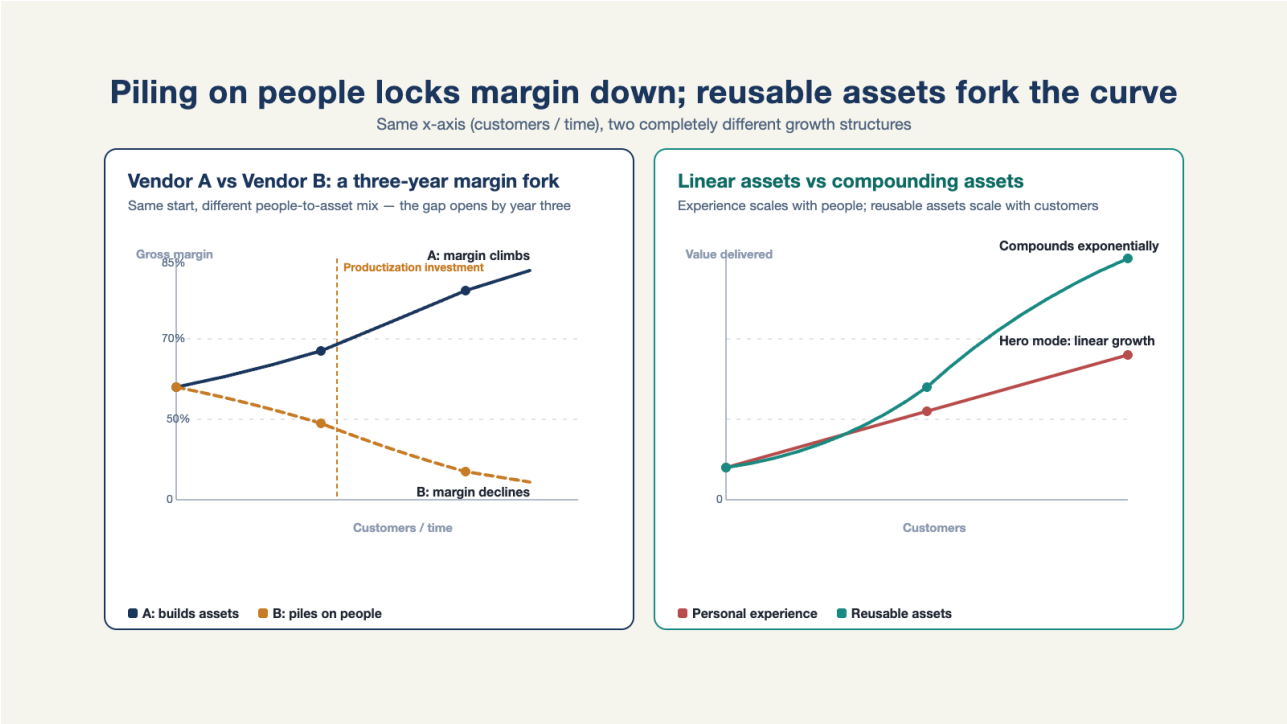

Figure 19-2: As customers grow, heroic headcount locks gross margin down; only reusable assets make the curve fork.

IV. McGrew's Two Ledgers

You can't talk about the big ledger without McGrew — he's the most successful person to have lived this model firsthand, and the most precise in walking through its numbers. On a podcast, he gave two separate accounts. [3]

The first, about KPIs. He drew a distinction between two strategies: in a product-market-fit strategy, you want to do less work per customer, lower costs, and keep contract size flat; the

[1] @deployengineer (Bhaulik Patel), essay series

[3] Y Combinator: The FDE Playbook for AI Startups (Bob McGrew)

FDE strategy is the opposite — you want to drive contract size up, doing increasingly valuable things for the same customer, and because the work is more valuable, it's acceptable to keep a certain level of customization per customer. So the internal yardstick is contract size, not necessarily work per customer.

At first hearing, this seems to contradict "declining customization" — wasn't customization supposed to go down? Hold on — the tension is real, and Chapter 21 takes it on directly: the right measure for the decline is customization's share of delivery, not absolute hours. When the contract grows and the unit of customization stays flat, the share still falls and gross margin still improves — the two readings are measuring the same health.

The second is about the path to margin. He said that a new deployment for a customer can actually lose money early on; the longer you stay with that customer, the more the product fits their needs through the discovery process — you no longer need a large group camped on-site figuring things out — and you've earned the right to work on bigger problems. So the cost per unit of value you deliver falls, and margin flips from negative to positive — it might take a year, it might take several. [3]

This ledger explains one thing: why a mature FDE organization is willing to "lose money" on the first few customers — that's not bad math, it's treating early losses as an investment in product discovery, a bet that the contract grows over time. It also adds a timeline to the six metrics: it's normal for every metric to look ugly on the first two customers — what matters is the direction, not the starting point.

V. A Three-Year P&L Comparison

Here's a demonstration table you can swap your own numbers into and recalculate.

Two similarly sized vendors start from almost the same place: three customers each, comparable contract value per customer, six-person teams each. Vendor A tags every customization request from day one, reuses connectors starting with the third customer, and by the fifth customer runs half its processes through configuration; Vendor B builds every deal from zero, because "the customer's in a hurry, reuse is too slow."

Year one, Vendor B's revenue is higher — no reuse means more person-days, and more person-days can be sold directly.

Year two, the fork appears: at Vendor A, the delivery cycle for customers six through ten

[3] Y Combinator: The FDE Playbook for AI Startups (Bob McGrew)

shortens by 40%, marginal cost compresses toward product cost, and delivery gross margin climbs from 30% toward 60%; at Vendor B, gross margin is locked below 30% by labor cost, and team size has to grow linearly with customer count.

Year three, Vendor A's output per person is more than double Vendor B's, and services added at renewal for existing customers cost almost nothing to deliver; Vendor B's three most senior engineers burn out and quit — the four customers they'd been carrying each need two new engineers to spend half a year finding their footing again.

The specific numbers above aren't real, but the mechanism behind the fork is: gross margin climbing from the 20—30% of the pure-labor phase to above 60% once platform reuse kicks in — "an FDE business whose margin doesn't climb is a consulting firm." [2]

VI. Build, Buy, or Blend

With the ledger settled, it comes down to a decision. A manager facing "should we have our own FDEs" is really choosing among three options. [1]

Build: fits when your differentiation lives in delivery (in your market, deep deployment is part of the product) and leadership accepts ugly metrics for the first year or two. Failure mode: a boundary-less team becomes a breeding ground for hero culture (Section III of this chapter), or you push ahead before platform capability has taken shape.

Buy: procure forward-deployed capability from an outside vendor, project by project. Fits when demand is intermittent and the scenario isn't core. Failure mode: knowledge never accumulates inside your own organization, and the second project is bought again from zero (none of the three exit conditions from Chapter 17 are present).

Blend: build for core scenarios, outsource for peak demand. This fits the widest range of conditions, but its failure mode is also the most hidden: the in-house team slowly gets "spoiled" by outsourcing and hollows out. The test is whether the in-house team keeps holding on to the hardest customers.

None of the three options is better or worse — the only question is fit with the big ledger. The one wrong option is the fourth one: hiring without ever having done the math (see Figure 19-3).

[1] @deployengineer (Bhaulik Patel), essay series

[2] Fan Bing, Forward Deployed Engineer (FDE) (XDash open-source book)

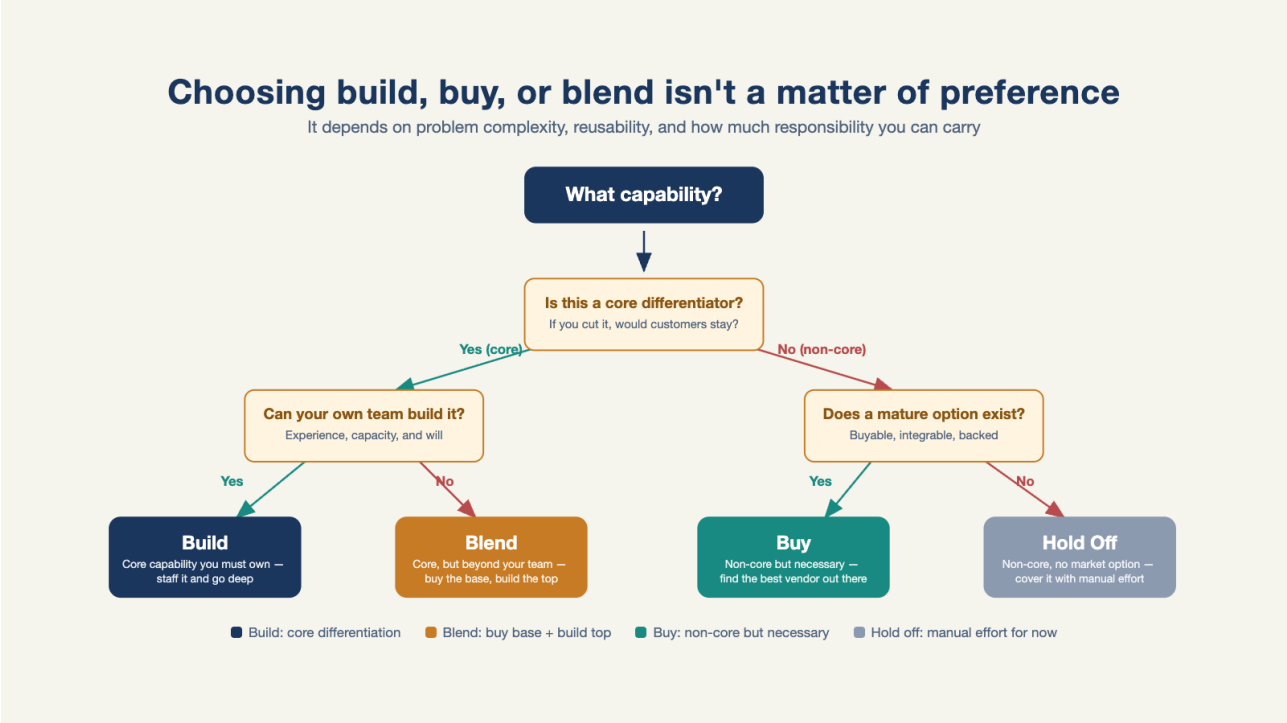

Figure 19–3: Choosing build, buy, or blend isn't a matter of preference — it's a matter of the problem, reusability, and the responsibility you can carry.

VII. Don't Do This at Home

Last is McGrew's warning–off: his first, second, and third pieces of advice are the same sentence — don't do this at home. [3] The weight of that line comes from who's saying it: the most successful practitioner to carry this model from Palantir to OpenAI is the one delivering the loudest warning in the room.

Put that line back inside this chapter's frame, and it turns out to be the last test of "big–ledger thinking": working the big ledger might, in the end, produce the answer "don't do this." That's not a failure — it's the most valuable output the ledger can give. Every company that wants to do FDE should first ask itself whether it can live with that answer; a company that can't bring itself to ask is probably already in the hole.

The big ledger settles the structure, but a finer–grained ledger is still waiting: for each individual contract, how does the money actually get collected? The next chapter takes up how to charge for it.

[3] Y Combinator: The FDE Playbook for AI Startups (Bob McGrew)

Chapter 20 · How to Charge: By the Day, by Scope, or by Result?

> The essence of strategy is choosing what not to do.
> — Michael Porter

"No results, no charge" is the single most tempting line you can put in front of a customer.

It makes something normally hard to pin down sound simple: pay once the project delivers results, don't pay if it doesn't. For a customer who's just been burned by a stalled project and doesn't want to foot another stack of day-rate invoices, that's almost impossible to turn down.

But the moment both sides try to write that line into a contract, the simple promise instantly sprouts details. How much does replenishment accuracy have to improve before it counts as "working"? When slow-moving inventory drops, is that the system's doing, or the combined result of the season, a promotion, and procurement strategy? If the payoff only becomes visible three months later, can it cover the staff time and cost the vendor has already sunk into those first three months?

A quote sheet that says "no results, no charge" isn't really opening a negotiation over price — it's opening a negotiation over these questions. Different charging models differ only in where they park this uncertainty: with the customer, with the vendor, or split between the two.

This chapter opens up the black box of pricing: the conditions under which each of the three billing models holds up, the preconditions for charging by result, one practitioner's pricing method, and the handful of things a contract must spell out.

I. Different Models, Different Risk

By the day. The oldest, most mature, easiest to sell — customers understand it, and procurement processes recognize it. The risk sits entirely on the customer's side: what they're buying is time, not results. Its failure mode is the collapse of Chapter 2's third test — someone selling time doesn't own the result, and the work quietly slides into high-end outsourcing. Bill by

[1] Crossroads × Rolling AI, "A Conversation on FDE" (podcast)

[2] Silicon Valley 101, Episode 240: "The Hottest New Job in Silicon Valley — FDE" (YouTube)

the day, and growth is permanently capped at person-days — none of the upside from results ever gets captured. [1] One enterprise customer-service AI platform vendor (Cresta) has gone further still: they say they've "never billed by FDE hours" — the FDE's labor is folded into the total of platform subscription and success fees rather than itemized separately; for a customer who's strongly interested but hasn't signed yet, they'll even front the labor and deliver results for free first — "it's like home delivery: you've already tried it on, and you don't want to be the one to say you're returning it" — treating FDE cost as a customer-acquisition investment rather than a pricing unit. [2] This isn't the same path as the result-based revenue share discussed later in this chapter (the customer still pays by subscription; internally, FDE hours simply aren't exposed to the customer) — it's a reminder that taking FDE labor out of the pricing unit is itself a way of allocating risk: the vendor absorbs the risk of front-loaded delivery cost, in exchange for a shorter sales cycle.

Fixed-scope delivery. The most common form in acceptance-based environments — it fits the rhythm of China's project-based procurement especially well. Risk is split roughly in half: the vendor owns delivery within scope, the customer owns the consequences of a poorly defined scope. It has two failure modes: scope creep ("just one more small change" piling up into a second project) and walls left unmapped — as Chapter 9 put it, a contract priced without first counting the eight walls is a contract-level cause of getting stuck.

Revenue share on results. This is the tightest alignment with outcomes: the customer pays the least for zero results, and the vendor shares in the upside. But its failure modes are also the sharpest: baseline disputes (how much of the money saved actually counts as the system's doing?) and delayed collection (delivery cost is front-loaded, revenue-share income arrives late, and cash flow gets crushed first). So it was never "more advanced" — it's just **a costlier precondition traded for deeper alignment** (see Figure 20-1).

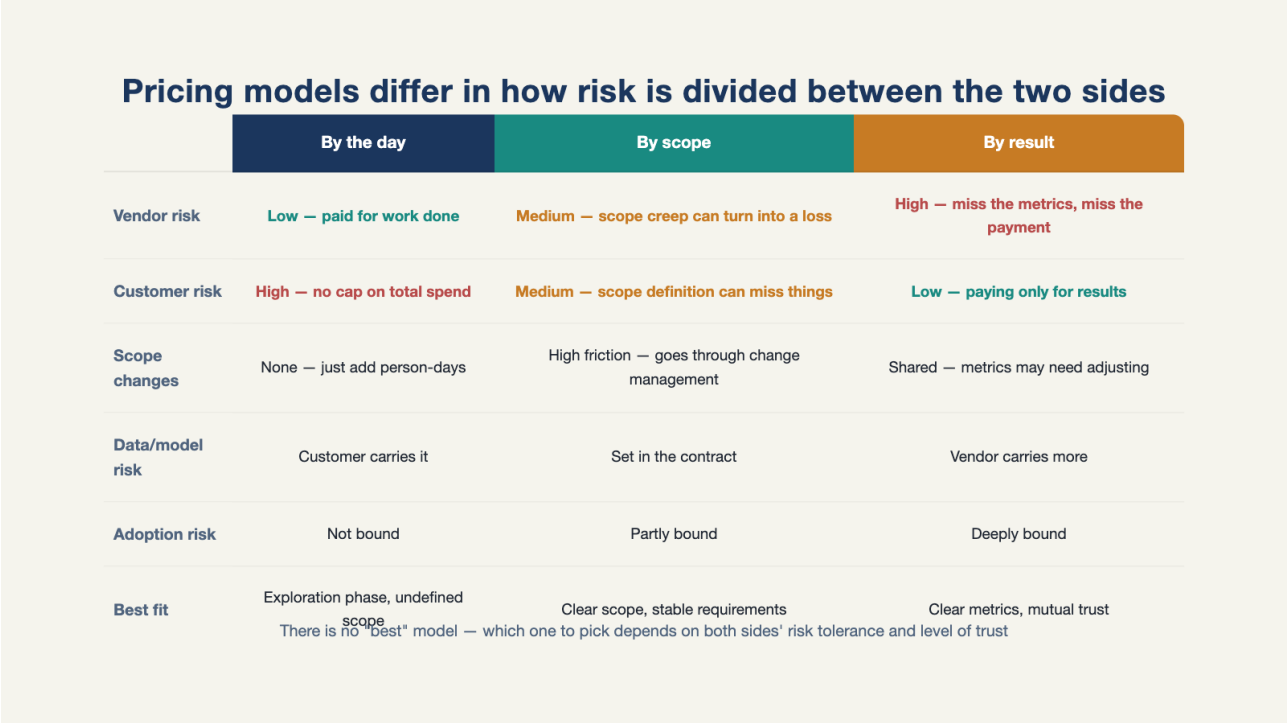

Figure 20–1: What separates pricing models, at bottom, is how scope risk and result risk get divided between the two sides.

II. Four Preconditions for Charging by Result

"No results, no charge" sounds like the vendor has tied its own fate to the customer's: if the project produces no effect, the customer doesn't have to pay for a pile of hours, meetings, and intermediate deliverables. Compared to billing by the day, this obviously reads more like a real commitment to results.

But once the contract negotiation actually starts, the questions quickly get specific: what counts as "meeting the target"? If sales go up, is that the system's doing, or the combined result of peak season, promotions, and the sales team? If the effect only shows up three months later, who fronts the delivery cost for those first few months?

Some call this arrangement "Result as a Service" (RaaS): the vendor doesn't just deliver a system — it also ties part of its billing to business results. What the customer buys is no longer just development work, but a results commitment written into the settlement rules.

That also explains both why it's tempting and why it's hard to pull off. Behind the line "no results, no charge," at least four things have to be settled first: where the effect starts being measured, who the effect should be credited to, how long you wait before settling up, and who backstops the worst case. Miss any one of them, and settlement time turns into a whole second project.

The four preconditions below decide whether charging by result is actually a workable way to collaborate, or just a bet dressed up to look elegant.

A measurable baseline. The effect needs a baseline both sides sign off on in advance — this is exactly the business meaning of the outcome acceptance sheet from Chapter 8: revenue share without a baseline turns settlement into an argument.

Credible attribution. The improvement has to be traceable to the system. If sales are up 30%, is that the agent's doing or the season's? The attribution logic has to be locked in before signing — a control period, control stores, or at minimum a conversion rule both sides accept.

A survivable time gap. Delivery cost comes first, revenue-share income comes later, and the gap in between needs to be checked against whether cash flow can actually survive it — more companies have starved to death charging by result than have simply botched the delivery.

A floor on the downside. The guaranteed base fee has to cover delivery cost. A pure bet isn't pricing — it's betting the company's life; a business with healthy cash flow has no reason to wager next quarter's payroll on a customer's business metrics.

All four preconditions share one name: measurability. The evaluation system from Chapter 12 closes its business loop right here — **evaluation isn't just a quality tool, it's the foundation pricing stands on.** A business that can't state its evaluation criteria clearly can only be sold by the day.

III. Three Pricing Anchors

A consultant who has served many enterprises grouped the pricing approaches he's seen into three categories. They aren't a universal template — just one way of thinking about it:

Formula one, cost reduction. His example: a company with fifty R&D staff, at an annual cost of RMB 200,000 per head; if headcount drops from fifty to thirty, that saves RMB 4 million, and the fee can be anchored to a share of that difference. The customer doesn't have to take the vendor's quote on faith — they just have to work out their own cost ledger first.

Formula two, replacing a fixed cost. An e-commerce company's product pages cost RMB 2–3 million a year in photography, models, retouching, and design; building the company an AI image-generation system was priced at RMB 1 million. The existing spend being replaced is the anchor for the price; the extra engineering cost needed to make that replacement work also has to go into the quote.

[3] Turning Point (破局点), Episode 31: "FDE, Chinese Style"

Formula three, revenue growth. A salesperson's monthly performance goes from RMB 100,000 to RMB 300,000, and the RMB 200,000 increase is shared at an agreed ratio. The price lands only on the incremental portion, which also makes it easy for the customer to keep it separate from existing performance. [3]

The three formulas look different on the surface, but the core is one sentence: **the price anchors to numbers the customer can already verify on their own books.** Cost reduction anchors to the headcount gap, replacement anchors to existing spend, revenue growth anchors to incremental turnover — none of these anchors requires the customer to "trust" the vendor's quote; every one of them, the customer can check with a calculator. This lines up neatly with Chapter 2's third test (what you're charging for) and the baseline measurement in Chapter 8: a quote is persuasive because the customer can recompute it on their own ledger.

Large platform vendors' price lists line up with this independent consultant's three formulas. Alibaba Lingyang's sales model runs on two tracks: one is by seat — a customer's existing 500-person customer-service team gets matched in output today for roughly 80—90% of the original cost, anchored to the labor spend being replaced (a variant of formula one); the other is by result-sharing — for marketing and ad-placement scenarios, only the portion of conversion rate above the original baseline gets shared, anchored to the increment (a variant of formula three). [4] More notable still is what they call "no results, no charge": "we don't call it a bet, we call it an evaluation" — first run a minimum-value validation on one of the customer's real business units, keep serving only once the customer signs off on it, and they say plainly that a customer whose numbers don't add up won't keep renewing. [4] Whether a "bet" can be relabeled an "evaluation" comes down exactly to this chapter's Section II preconditions: a measurable baseline and credible attribution earn it the name evaluation; without those, it can only be called a bet.

IV. A Blended Structure: A Steadier Arrangement

For projects with high delivery cost and a lagged payoff, a steadier structure is: **a fixed fee + a results bonus + a run fee.** The fixed fee covers predictable delivery cost, the results bonus shares in the upside, and the run fee corresponds to managed operation after handoff. The run phase keeps consuming engineering and support resources; if it isn't billed separately, that cost erodes delivery gross margin.

A reference skeleton for the clauses: the fixed fee splits into three milestone payments, each

[4] Silicon Valley 101, Episode 248: "China-Style FDE, with Alibaba Lingyang's Peng Xinyu"

tied to acceptance criteria (Chapter 8); the results bonus sets two thresholds — zero below the baseline, capped at an agreed ratio above the target; the run fee is billed annually, spelling out response tiers, escalation paths, and responsibility boundaries (Chapter 17); plus a settlement rule for either side terminating early.

V. Five Things a Contract Must Spell Out

Beyond the model, there are the clauses. If any of these five things isn't spelled out, no model will hold up.

First, the evaluation criteria and who judges them. Which eval set is used, what counts as meeting the target, who has the final say — the evaluation system from Chapter 12 becomes a contract exhibit right here.

Second, the customer's obligation to cooperate. Cooperation on data and permissions is a delivery prerequisite (Chapter 10) — if the customer's delay causes a slip, whose fault is it? Without this clause, every delay defaults to the vendor.

Third, acceptance milestones and rollout rules. How many stages of acceptance, how the rollout percentage climbs (Chapter 8) — written into the contract, not left in an email thread.

Fourth, the responsibility boundary after handoff. Incident severity tiers and response during the managed period after acceptance, tying into the responsibility handoff table from Chapter 17.

Fifth, IP ownership. Who owns the custom code, the eval set, the reusable modules — this clause directly collides with the product feedback loop in Chapter 22 (field assets are exactly the raw material that flows back), and the vendor has to fight for it, but how it's fought can be designed: custom code stays with the customer, reusable components stay with the vendor, the eval set travels with the handoff (Chapter 17's position).

To guard against getting used for free, one more practice comes straight from a frontline practitioner: **split the proposal in two.** The pre-sales proposal only covers the problem diagnosis, the technical approach, and the price; the actually executable staged steps, acceptance metrics, and implementation details get worked out jointly only after signing. The reason is blunt: if the customer takes an already-executable proposal to another vendor, or hands it to an internal team to copy, all the pre-sales work the vendor put in goes to waste. [3]

[3] Turning Point (破局点), Episode 31: "FDE, Chinese Style"

VI. The Buyer's View: How to Read a Quote

Last, stand on the other side of the table — the customer. Once a quote lands, ask three things first.

First, what does this money actually buy? Hours, a defined scope, or an already-defined result? Look at the billing structure before the total price — only then can you tell where the risk lands if the project goes off the rails.

Second, what does a low price leave out? Are integration, data governance, and operational support already priced in? Those costs don't disappear — they may just come back mid-delivery as change orders.

Third, who defines the metric? If the quote says "expected 40% improvement," what's the denominator, how long is the observation window, who makes the call? A commitment that isn't clearly defined is a dispute waiting to happen at settlement time.

No matter how well a single contract is priced, that's still only a victory on the small ledger. If the hardest curve in Chapter 19's big ledger — the Nth customer's customization volume — doesn't come down, no pricing model can save gross margin. The next chapter takes up how to make customization actually decline.

Chapter 21 · Why Does the Nth Customer's Customization Volume Have to Decline?

> Today's problems come from yesterday's solutions.
> — Peter Senge, The Fifth Discipline

"This customer is just like the previous nine — do we send another team of engineers?"

The tenth customer has just signed, and the project manager starts scheduling: connect the systems, configure permissions, adjust the workflow, build the evaluation, support after launch. Every item was already done on the previous nine projects, yet the schedule still reads as if this were the very first customer.

The real danger isn't that the delivery team isn't working hard enough — it's that the company never turned the previous nine deliveries into a capability the tenth one can use directly. Why does the Nth customer's customization volume have to decline? That's the question this chapter answers.

I. The Target Isn't Customization — It's Customization That Never Declines

The forward-deployed model was built on customization — this book has never been vague about that. A hundred percent customization for the first customer is inevitable: you're seeing what this industry's workflow even looks like for the first time, and there's nothing to do but customize. Criticism that treats "customization" as a dirty word is aiming at the wrong target.

What's actually fatal is something else: **customization volume that doesn't decline with customer count.** A hundred-percent-customized first customer is where the model starts; a tenth customer still 80% custom is where the model fails — because gross margin, delivery speed, and quality consistency will all degrade linearly with customer count, which the next two sections unpack.

So this chapter's question isn't "how do we do less customization" — it's "how do we make the Nth customer's customization volume significantly smaller than the first customer's." The metric

[1] Fan Bing, Forward Deployed Engineer (FDE) (XDash open-source book)

already exists: the Customization Decline Rate, introduced in Chapter 19, with the reading rule spelled out in the open-source reference book — if it doesn't drop for three customers running, check the product feedback mechanism immediately. [1]

Demand is moving the same direction too. One delivery lead at a major platform company said orders where the customer just buys person-days will keep shrinking — day-rate customization is, for both sides, an inefficient model of doing everything yourself, from building the henhouse to raising the hens to collecting the eggs. He drew the same distinction even for Databricks, known for its service revenue: what it does is "help the customer find the problem worth solving, not help them knock out the 100 feature requests on their to-do list." [2] As "buying features by the person-day" recedes, delivery organizations are forced to answer this chapter's question: if not by piling on headcount, what powers customization instead?

II. Two Cost Curves

Start by using the math to spell out what "dying slowly" actually looks like.

In a world without decline, the cost structure is a multiplication: **total customization cost = number of customers × unit customization cost.** Say each customer takes fifty person-days — ten customers is five hundred person-days — and team size has to scale linearly with customer count. This is exactly Vendor B's path from Chapter 19's three-year comparison table: gross margin locked down by labor cost, its most senior engineers burning out and quitting in year three. The gross-margin trap (Chapter 19) and this cost curve are two sides of the same ledger.

In a world with decline, the Nth customer's delivery cost gradually approaches the product's cost to replicate.

By the tenth customer, per-customer cost differs tenfold between the two worlds — and the decline world doesn't just save money: the delivery cycle shortens, and quality consistency rises at the same time. Decline isn't a cost-saving line item; it's the threshold this business has to cross to turn from a labor business into a product business.

III. When Customization Capability Only Lives in a Person

The problem isn't that a given FDE is too capable — it's that his judgment, his trade-offs, and his record of what went wrong stay locked in his own head. A ticketing integration built for the

[2] Silicon Valley 101, Episode 248: "China-Style FDE, with Alibaba Lingyang's Peng Xinyu"

first customer gets rebuilt from memory when the fourth customer arrives; that's more practiced customization, not repeatable delivery.

Individual proficiency can keep improving, and customer count will keep growing too. If every delivery leaves nothing behind that the organization can use, customers will eventually outnumber what one person's capability can absorb.

The fix has the same shape as the external handoff in Chapter 17: **deliver the system outward, accumulate knowledge inward.** Every time customization wraps up, it should leave behind a list of reusable assets: which integration capability can be componentized, which processes can be turned into configuration, which evaluation structures can migrate, and which parts belong only to this one customer.

IV. Can the Contract Still Grow Without Customization Falling?

McGrew said on a podcast: the FDE strategy has to drive contract size up, doing increasingly valuable things for the same customer, so **it's acceptable to keep a certain level of customization per customer**, and the internal yardstick is contract size, not work per customer. [3] "Customization roughly holding steady"? That's a direct clash with this chapter's title.

The key is the unit of measure: **decline measures customization's share of delivery or contract value, not absolute hours.** A single customer's customization volume can stay flat for a while; as long as product leverage keeps pushing contract size and delivered value up, customization's share still falls, and gross margin still has room to improve.

Lowering the numerator (less customization) and growing the denominator (bigger, higher-value contracts) are two readings of the same health metric. Most teams have to nail the first one first; the second only holds if the customization content is actually flowing back into product capability, continuously.

V. Three Kinds of Reusable Capability Behind the Decline

With the measure defined, the question is what actually drives the decline. Three engines, each backed by evidence from the field.

[3] Y Combinator: The FDE Playbook for AI Startups (Bob McGrew)

[4] Inside an Applied AI company (Pace long-form post)

Set the ground rule first: customer data belongs to the customer and can't be carried to the next project. What actually accumulates across deployments is judgment about the scenario, methods of evaluation and verification, and the engineering process that turns "it ran once" into stable delivery. [4]

Capability one: componentizing integration capability. What's reusable isn't customer data, and isn't just interface code — it's a configurable set of integration capabilities: authentication methods, field mapping, sync rules, exception handling, and permission boundaries. Customer differences stay at the configuration layer; the stable parts move into platform components.

Capability two: turning processes into templates. Business processes common to an industry get abstracted into configurable templates. There's a line here that's easy to blur: **configuration isn't customization — configuration changes parameters, customization changes code.** The goal of templating is to downgrade "change the code" requests into "change a parameter" requests: approval tiers, field naming, and threshold ranges go into a configuration table; code only moves when the process topology itself changes.

Capability three: migrating evaluation structure. Evaluation patterns can accumulate across customers: task types, difficulty tiers, and judging rules are reusable; case data is not. What travels is "how to test," not the customer's "exam paper."

All three capabilities share one trait: **every reuse makes the next delivery lighter.** Platform components fill in exception handling, templates accumulate parameter experience, evaluation structures expand their library of failure samples. Reuse isn't carrying the old solution over unchanged — it's letting the reusable portion keep accumulating, so each next delivery gets a little lighter (see Figure 21–1).

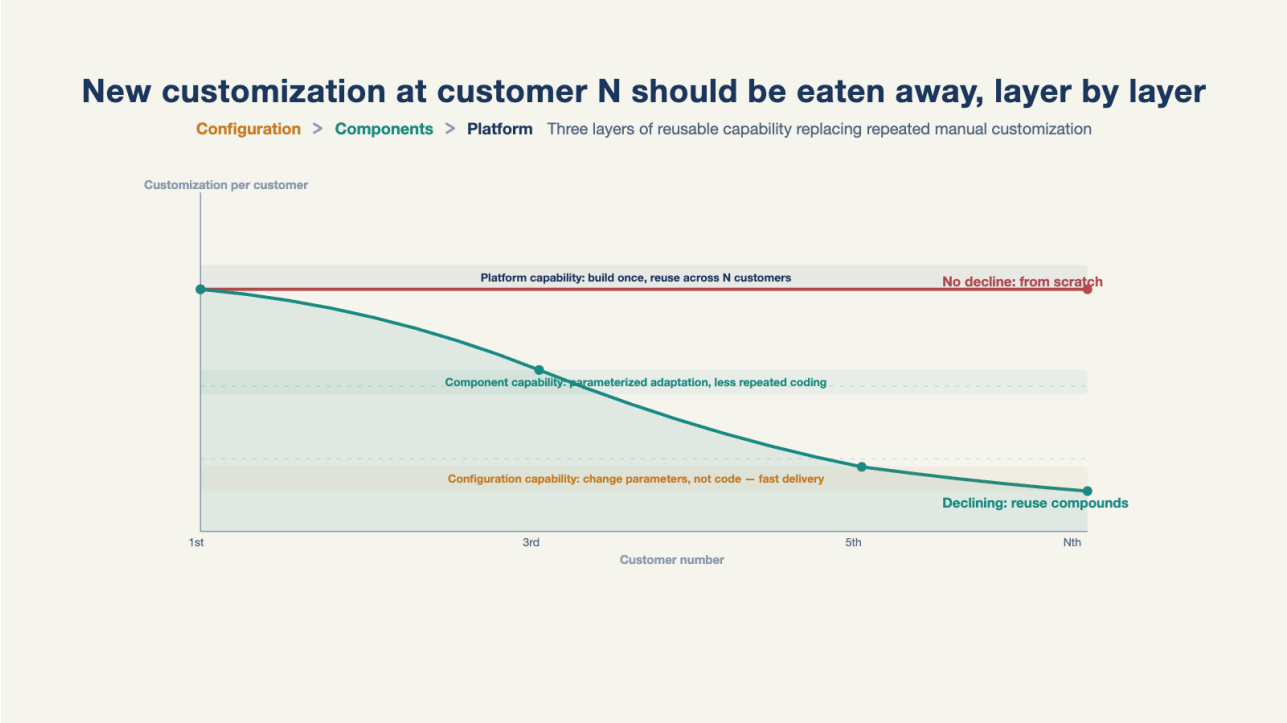

Figure 21–1: Whatever new customization the Nth customer needs should be eaten away, layer by layer, by components, configuration, and platform capability.

VI. Where Configuration Ends and Customization Begins

Before the engines start turning, every request has to clear a gate at the door. Three questions, asked in order:

Question one: can configuration solve it? Anything a parameter, a template, or a rules table can handle is barred from touching code.

Question two: could this become a default capability for the next customer? Is this need unique to this one customer, or will other customers want the same capability too — if it's the latter, even if it has to be coded today, it should be written to the standard of a "future default capability": merged into the main branch, with proper abstraction, with parameters left open.

Question three: is this genuinely customization only? Unique to this customer, with no expectation of reuse — that's fine to customize, but the customization code must be **tagged**: one-off code has to be clearly marked and physically separated from product code. The tag isn't ceremony — it's what stops customization code from disguising itself as product capability. Half of the dead code discussed in the next paragraph gets in exactly this way.

[2] Silicon Valley 101, Episode 248: "China-Style FDE, with Alibaba Lingyang's Peng Xinyu"

Behind these three questions is a more fundamental directional call, from a delivery lead at a major platform company, put as a syllogism: customization, standardization, personalization. Software delivery used to work by "using customization to solve standardized problems" — encoding a business's already-smooth-running process into the system; get the encoding right and it solves 30% of it, get it wrong and there's "a fresh 70% increment every single day," turning the project into "a long, drawn-out black-swan engineering effort." [2] The agent era's opportunity runs the other way: "using standardization to solve personalized problems" — once workflows, data flows, and enterprise context are standardized, the same product can autonomously grow its own distinct traits at each different customer; whereas "once something is customized, it's hard to make it personalized" — customization locks down precisely the capacity to change. [2] Put into practice, that's exactly the direction the three-question gate pushes toward: keep downgrading "change the code" into "change a parameter," then let "change a parameter" settle into a preset capability the product ships with out of the box.

VII. Timing Productization: When to Abstract, When to Leave It Custom

Once the engines and the gate are both in place, what's left is a timing question: when to upgrade customization into product. The first customer usually has to go through the whole thing end to end, and the product team then abstracts out a general-purpose version from it that can serve the customers that follow. [3]

There are two common misjudgments here.

Stuffing one customer's solution straight into the product. It only solves one customer's problem, yet every future customer ends up paying its maintenance cost. A correct abstraction has to sit one level more general than the problem the customer brought in. Palantir, in its early days, moved away from building separate tables for concrete objects like "people" and "funds," toward describing the business using objects, properties, relationships, and actions — this modeling approach later crystallized into what's now called **Ontology**: instead of a separate table for every concrete object, a unified set of object types, property types, relationship types, and action types describes any category of entity (see Figure 21-2). [3] Palantir's own materials position Ontology as "a digital twin of the organization," and stress specifically that it isn't a thin semantic layer on top — integrating data, logic, operations, and security together can't be done with a tagged field dictionary. [5] Customer differences stay at the modeling and configuration layer; general-purpose capability stays at the product layer. How the Ontology keeps growing

[3] Y Combinator: The FDE Playbook for AI Startups (Bob McGrew)

[5] Palantir official documentation: the Ontology architecture and permission system

across customers and feeding the next delivery is taken up in Chapter 22.

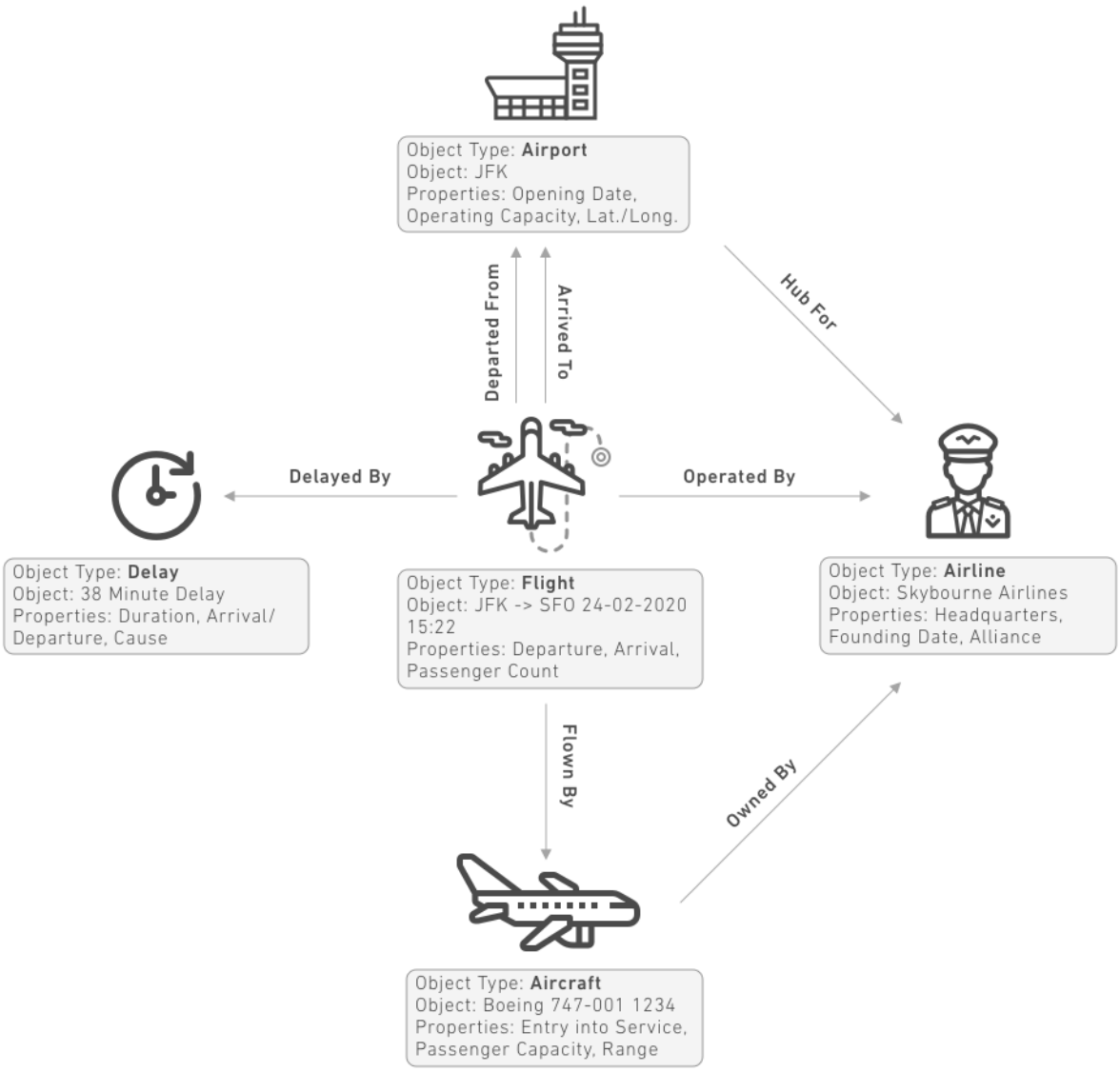

Figure 21–2: Palantir's own Ontology teaching example (aviation); the English labels in the image are from the original and are left untranslated. Image source: Palantir's official documentation.[5]

Rushing to productize before a common signal shows up. Investing for future customers when there's still only one customer risks turning guesswork into platform features. The steadier discipline: don't rush to roll it out before there's a second customer; if a third instance of the same need still hasn't been acted on, that's when to check the feedback mechanism.

[5] Palantir official documentation: the Ontology architecture and permission system

[6] Turning Point (破局点), Episode 31: "FDE, Chinese Style"

The signal can come from three places: the same process gets raised by two customers in different variants; pre-sales keeps demoing the same integration; a new customer proactively asks to reuse the previous solution. There's one more precondition: subsequent customers have to follow a similar business path. One practitioner's case: only after serving three to five customers in the same industry did the process gradually take shape as an agent product built for that industry. [6]

VIII. What Won't Decline: Book It as a Service Business

Some customization is naturally exempt from decline: a mega-customer's unique process (that's the vehicle for its competitive advantage, and it won't give ground for your reuse); differing compliance requirements across companies in a heavily regulated industry (different regulatory texts mean different obligations to rebuild for). The right way to handle this kind of need isn't to force it into the decline narrative — it's to admit it: **this is a service business, so price it as a service business** (the day-rate and blended structures from Chapter 20), book it separately (Cost to Serve from Chapter 19), and don't let it slip into the story of "we're building a product too" to inflate valuation expectations.

Honestly labeling the naturally non-declining portion as service actually protects the value of the declining portion — every point of drop on the decline curve is then real productization; lumping the two together and reporting a "product revenue share" is the most hidden version of the gross-margin trap.

Behind the falling curve is a hand carrying field experience back to the product. How that mechanism gets built, what gets carried back, who's responsible, how it's incentivized — the next chapter takes up flowing experience back into the product.

Chapter 22 · How Does Field Experience Become Product Capability?

> Good intentions don't work. Mechanisms do.
> — Jeff Bezos

"Same pitfall, again?"

The night before the second customer's launch, an FDE stares at the error log for a moment. The problem is almost identical to the last customer's: two systems used different names for the same thing, and the moment the workflow connected them, permissions and approvals fell into chaos.

He knows how to fix it. He already fixed this once before.

Except last time's rules, mappings, and tests all stayed behind in another project. So he redoes it all from scratch. The customer's problem gets solved, but the company forgets the answer a second time.

I. Why the Field's Answers Never Make It Back to the Product

When a company does forward deployment, it's actually running two things at once.

The first runs at the customer's site. The FDE uses this customer's data, permissions, and workflow to get one specific problem running; the goal is to get the project live. The second is the vendor's own product platform: a data model, integration components, process capabilities, and application tools that multiple customers can reuse; the goal is to keep the next delivery from starting over at zero.

The field and the platform aren't naturally connected. The FDE finds the problem on-site and writes a solution that works; if that solution just sits as a project deliverable in the customer's folder, the platform learns nothing from it. Over time, the platform can end up looking like it has plenty of features, while the field keeps having to customize the same things over and over — and what's actually being done is still outsourcing.

The core of the problem: demand signals are generated on the delivery team, product decisions are made on the platform team, and there's no fixed mechanism carrying information between

them. The delivery team's priority is hitting milestones; the product team's priority is protecting the roadmap. Neither side is wrong, and yet experience stalls inside the project anyway.

Flowing it back, then, isn't a matter of "write more documentation" — it's an interface that has to be designed on purpose: identify the assets that took shape in the field, carry them into product decisions, and then let the new product capability flow back into the next delivery.

II. First, Translate the Field Problem into a Shared Business Language

The problem that opened this chapter looks like field mapping on the surface, but underneath, it's two systems not speaking the same business language. One system calls it a "service request," the other calls it a "ticket"; the FDE wires the two together on the spot, and the customer can keep working. But the next customer will show up with yet another set of names, statuses, and approval flows.

To make this kind of experience persist across customers, it's not enough to save one instance of mapping rules — a shared language for describing the business has to be accumulated. This is the **Ontology**: it defines what objects exist in the business, how those objects relate to each other, and which actions can be carried out under what rules and permissions. [1]

Take a ticketing scenario: tickets, customers, devices, and service staff are objects; "which device a given ticket belongs to" and "who handles it" are relationships; dispatch, escalate, and close are actions; amount thresholds, approval tiers, and visibility scope are rules. The customer keeps its own field names and raw data; what the platform accumulates is this modeling approach.

Inside this modeling approach, there's one more dimension that's easy to lose under the three words "objects, relationships, actions": **permissions.** Who can see this ticket, who can dispatch it, who can change its fields — these boundaries have to be accumulated too, not bolted on afterward as a layer of access control. Palantir's own materials use the example of a medical manufacturing company to make this point: the production team needs global telemetry access to equipment, warehouse staff have permissions narrowed to their own region, and supply-chain analysts have permissions scoped down to specific rows and columns; once all three roles start using AI agents to operate on the same set of business objects, each agent's permissions have to inherit from the permission structure of the human user or project it represents — it can't run on a separate set of rules. [2] Objects, relationships, actions, and permissions together are

[1] Y Combinator: The FDE Playbook for AI Startups (Bob McGrew)

[2] Palantir official documentation: the Ontology architecture and permission system

what make this shared language's definition complete — but writing the definition clearly is only the starting point. How does it turn from a one-off mapping rule into an asset the next customer can pick up and use directly?

"Distillation" is the more everyday word for "flowing back." Lu Xiaopeng (co-founder of Ventus AI) put it most sharply: experience scattered across everyone's individual heads doesn't count as accumulated — it only counts once it's gathered into a concrete knowledge base or Skill. Zhong Qianjie (Cresta) offered a further-along productized form: his team packages the distilled output — maybe a Markdown file, a CLI, a script — into an agentic tool; the customer says what they want to do, and the system automatically calls the best-practice logic behind the scenes to try it. [3] The business Ontology is the shared language distilled out of the field; the six reusable assets below are everything else that can be distilled out of it and packaged into tools that feed back into the field.

III. What Flows Back Isn't "Experience" — It's the Six Reusable Assets

The business Ontology is the shared language, but it can't complete a delivery on its own. The field also leaves behind five more kinds of portable assets.

Connectors. A connector isn't just the story of "we wired up an interface" — it's a reusable integration component: authentication method, data retrieval, field mapping, sync rules, exception handling, and permission boundaries. What it solves is how data gets connected reliably, not how to carry the customer's data away.

Tools and approval templates. The ticketing system, approval chain, risk tiering, and operating interface the field builds to constrain what an agent can do — if these can be configured for use by different customers, they shouldn't stay stuck in a single project forever.

Process templates. Processes that recur across the same industry can be abstracted into parameterized templates. Configuration changes parameters, customization changes code; a template's value is turning more and more "change the code" into "change a parameter."

Evaluation structure and the failure-mode library. What's portable is task types, difficulty tiers, judging rules, and failure modes — not customer case data. It lets the next project start from "how do we verify this," instead of starting from "let's just try it and see." [4]

[3] Tencent Research Institute, AI Lens Roundtable, Episode 6: "FDE Non-Consensus Views and a Field Guide from Silicon Valley Founders"

[4] Inside an Applied AI company (Pace long-form post)

Demand signals. Which requests keep recurring across different customers, which are just one customer's exception — these records determine what's worth putting on the product roadmap. This is often the easiest one to throw away, and also the closest one to the product direction.

A former Palantir employee recalled that requests that kept recurring in the field — importing data, building visualizations, building pages — gradually accumulated into data-ingestion, visualization, and application-building tools. The product wasn't drawn up first and then accepted by customers; a lot of the time, it grows out of requests that kept showing up in the field, over and over. [5]

IV. Getting the Assets Actually Back to the Platform: Three Steps and Two Gates

Step one, the FDE builds and records the context of the problem on-site: why the customer needs this capability, why alternative approaches failed, which parts belong only to this customer.

Step two, the FDE carries this context back to the product team, and together they discuss "the correct general-purpose version." Handing over a requirements document alone isn't enough, because documents usually lose the constraints and trade-offs that shaped the requirement in the first place.

Step three, invite FDEs serving other customers to validate it together. With three workflows that are slightly different but share a common origin laid out side by side, the team can much more easily tell what's an industry commonality and what's just one customer's habit. [1]

Around these three steps sit two gates. The first gate governs intake: requests get sorted into one-off, industry-common, and general-purpose before anything else — a one-off request can still be delivered, but it doesn't go straight into the product queue. The second gate governs output: the feedback review has to give every asset a clear destination — into the product roadmap, into the component library, into the template library, or an explicit "not doing this" (see Figure 22-1).

[1] Y Combinator: The FDE Playbook for AI Startups (Bob McGrew)

[5] "Reflections on Palantir" (Nabeel Qureshi)

Figure 22–1: Not every customer request should be productized; only by passing through two gates does reusable field experience make it back to the platform.

Zhong Qianjie's team's practice is a concrete example of this output path: the distilled asset isn't left as vague as "goes into the component library" — it's packaged directly into an agentic tool: the customer says what they want to do, the tool automatically calls the accumulated best practice to try it, and if that doesn't work, it switches to the next approach and tries again. [3]

Productization also can't just take orders from whatever the customer asks for. A request has to clear at least three questions at once: does it keep recurring at similar customers? Does it serve a business goal the customer genuinely cares about? Can it be abstracted one level further than the current customer's solution? If any one of the three can't be answered, don't rush to turn it into a platform capability.

V. Making Feedback Something Both Sides Actually Want to Do

A checklist and a process can't change incentives — without that, they just stay on paper. The people doing delivery need to be able to see the payoff of feeding things back: when a component, template, or evaluation structure they built gets reused by a later project, it should count toward their delivery results.

[3] Tencent Research Institute, AI Lens Roundtable, Episode 6: "FDE Non-Consensus Views and a Field Guide from Silicon Valley Founders"

The product team also needs to reserve room for field signals. Without a clear capacity and decision-making authority, field requests will never outrank the existing roadmap; if the review only takes in requests and never disposes of them, the feedback review turns into a mailbox.

Nor should feeding back be one-directional. Every time the platform accumulates a configurable component, a process template, or an evaluation structure, it should go back to the FDE first, so that at the next customer he needs fewer people, writes less one-off code, and spends his time on harder problems. When McGrew calls the FDE another kind of key customer of the product, this is exactly the relationship he means. [1]

Whether feedback is actually running isn't measured by meeting minutes — it's measured by the FDE's choices: faced with a new request, does he reach for the platform's components to assemble a solution, or does he route around the platform and write another one-off fix.

VI. Three Ways to Make Feedback Fail

Turning the review into a blame session. The delivery team vents about the product, the product team defends the roadmap, and nothing comes out of it — no actionable asset, no clear decision. A review should only discuss actionable assets, and every one of them needs a clear destination.

Turning the review into an intake box that never outputs anything. Requests get logged as to-dos, but no one has the authority to reorder priorities; a few months later, the field naturally stops submitting. A feedback review has to carry real authority to influence product priorities.

Turning the first customer's answer straight into the product. That isn't feedback — it's just moving customization code into the platform. There's no need to rush before a second similar customer exists; only if the same kind of request keeps showing up with no action taken should you go back and check the mechanism.

Chapter 21 asked: why can't the Nth customer keep repeating the first customer's labor. This chapter's answer: the answers the field arrives at again and again have to be translated into shared language, components, templates, and evaluation, and then carried back to the platform through a mechanism that actually has the authority to decide.

[1] Y Combinator: The FDE Playbook for AI Startups (Bob McGrew)

Part Six · People and Organization

The ledger is balanced, the walls are scaled — the only variable left is people.

The six chapters unfold across three vantage points. First, **see the job clearly**: Chapter 23 strips away the job-title filter to show what a day and a week actually look like; Chapter 24 builds the three-pillar capability model and a twelve-question self-check. Next, **walk the path**: Chapter 25 covers how engineers make the transition; Chapter 26 switches to the employer's side — when to build a team, and how; Chapter 27 asks whether an individual can go it alone. Finally, Chapter 28 closes the book — does experience hold its value, does the role survive, does the whole structure disappear — landing on one sentence: AI projects stall because nobody is accountable for the last mile, and FDE is what that accountability is called right now.

Written for engineers weighing whether to step in, managers sizing up whether to build a team, practitioners considering going independent — and for the industry observers who have read the whole book and want to test their own judgment against it.

Chapter 23 · What Does an FDE Actually Do?

> Managerial activities are characterized by brevity, variety, and discontinuity.
> — Henry Mintzberg, "The Manager's Job: Folklore and Fact"

Hangzhou, 11:30 p.m. An FDE's WeChat lights up: a customer calling to say the weekly-report bot has stopped sending messages. He remotes into the system, checks everything, finds nothing wrong — and finally sends someone to the site the next day, who discovers that someone had simply switched the machine off. [1]

That same week in Silicon Valley, another FDE watches ten customer systems light up with alerts at once: an upstream Cloudflare outage, no contingency plan, an all-night firefight.

Same job, two very different nights. This chapter strips off the job-title filter and pulls the role's real rhythm, task mix, and emotional texture out of four real samples: two Silicon Valley playbooks (running many customers at once, or embedding deep with one), a brand-new hire in Japan, and an ordinary day on the Chinese side in Hangzhou.

I. Two Rhythms: Many Customers in Parallel vs. Deep On-Site

The first sample comes from Silicon Valley: an engineer with twelve years of experience who moved into a forward-deployed role in the last two years. He posted his own "week in the life": **ten active customers, roughly fifty hours a week, meetings and actual building splitting the time about evenly**, plus unplanned firefighting on top — the Cloudflare outage that opened this chapter was all ten of his customers' systems alarming at once. [2]

Worth reading line by line is the task mix: pushing forward multiple customer builds in parallel, presales calls, firefighting, weekly reports. Not one line reads "research algorithms." Someone in the comments asked him how the job differs from a traditional software engineer's. His answer was blunt — in some ways it's more satisfying, because the solution is his, start to finish. [2]

[1] Turning Point (破局点), Episode 31: "FDE, Chinese Style"

[2] Hacker News, "A Week in the Life of a Forward Deployed Engineer" (10 customers, 50 hours)

[3] Baseten, "What I Learned as a Forward-Deployed Engineer Working at an AI Startup" (Het Trivedi)

The second sample runs at a completely different rhythm, also American: the Baseten FDE from Chapter 1's time-budget breakdown (roughly three-quarters engineering and model optimization) moves into a support role once delivery wraps, staying with the same customer. [3]

Set them side by side: one is **many-customers-in-parallel** (ten customers, high-frequency switching, presales included too), the other is **deep-on-site** (a handful of customers, a bigger engineering share, staying on to run things after delivery). Same title, but the material conditions, the rhythm, and where the effort concentrates all differ — Chapter 18 already showed this is a function of market structure. The fact worth holding onto here: **before asking what "an FDE's day" looks like, first ask which kind of FDE.**

II. A New Hire's First Days Aren't Spent Writing Algorithms

The third sample comes from a new hire in Japan, two months into the job, whose account corrects the most common misconception. She had imagined this as "fighting alone on the customer's site." Two months in, the reality looks different: the project has just gone live, and the team is in the phase of waiting for customers to actually pick it up. Her days go to **sorting out requirements, sometimes writing code, and following through until what the customer actually wanted becomes real.** [4]

Two details carry the most weight. First, part of her job is **sitting with the customer's admin to configure production permissions and authentication** — a new FDE's starting point isn't algorithms, it's plumbing into the system (all of Chapter 10). Second, her working principle: **rather than write the perfect spec, build a prototype the customer can actually try, watch how they react, and correct course on the spot** — which is exactly the new-hire version of Chapter 7's Minimum Viable Deployment. [4]

III. Assembling a Week

Three samples, each weighted differently — put together, they form one composite weekly structure: **Monday**, on-site or planning meetings, aligning on this week's goals and risks; **midweek**, building and integration — writing code, wiring up permissions, running evals (the daily grind of Part Three); **Friday**, the customer's weekly report and eval review, letting the data do the talking (Chapters 12 and 13).

[4] "I Thought FDE Meant Fighting Alone at the Customer Site" — What Two Months on the Job Actually Looks Like (note.com · AI Shift)

But that ideal cycle is only half the picture. The other half is production incidents that show up whenever they want (the midnight WeChat message, the outage upstream), requests shoved in at the last minute (the demo the boss suddenly wants, a customer calling an impromptu meeting), and the relationship maintenance that holds it all together — in the Chinese sample, that last item isn't a lubricant, it's a load-bearing wall. The ratio between these two halves largely decides which version of this job you end up experiencing (see Figure 23-1).

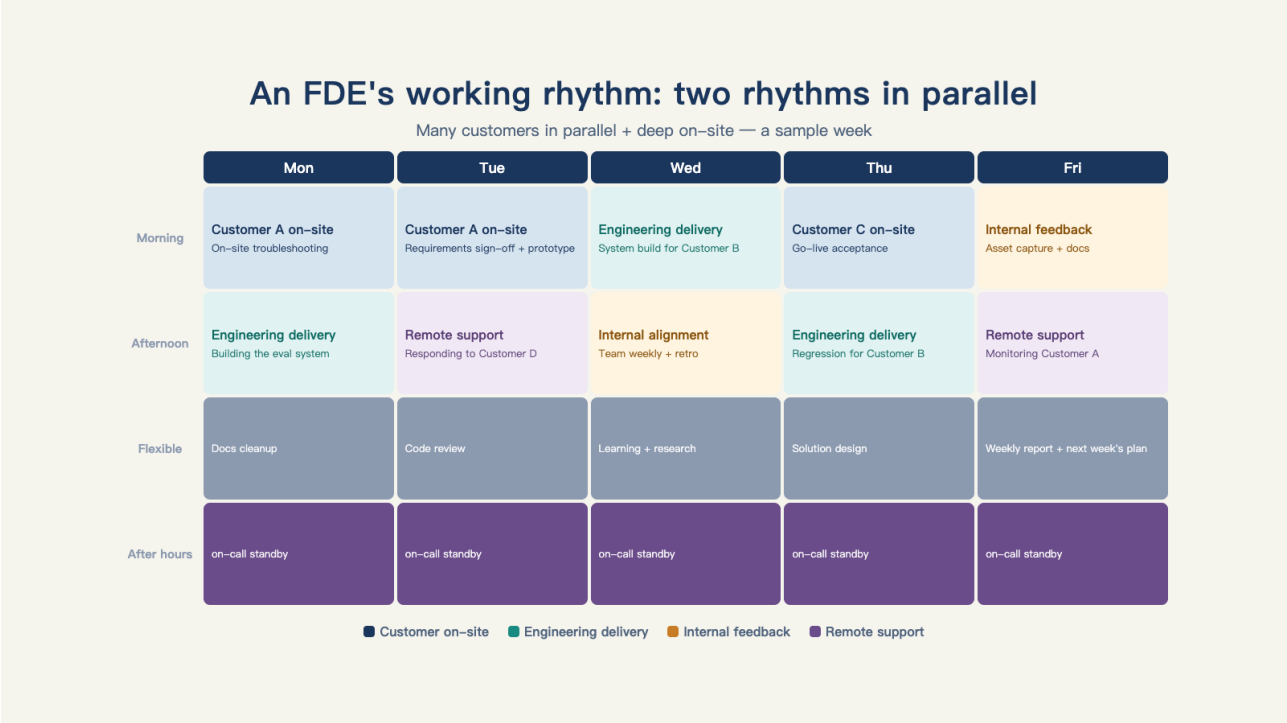

Figure 23-1: An FDE's working rhythm swings between on-site presence, engineering delivery, feeding lessons back internally, and ongoing support.

IV. A Day in China

Turn the lens back to China — the material conditions here run on an entirely different base. That midnight phone call from the opening — the weekly-report bot gone silent, traced back to someone casually switching off the computer — is a product of that base; Chapter 18's time budget of **sixty percent spent untangling the business and maintaining relationships** shows up, day to day, as exactly this rhythm. [1] The same account has an even more absurd detail: a customer showed up with a computer still running Windows XP to install the deployment package — and this was 2026. [1]

[1] Turning Point (破局点), Episode 31: "FDE, Chinese Style"

[5] "Anyone Out There Actually Doing FDE Work?" (linux.do thread)

The real fork in the road sits here: the "on-site prototyping" FDE the community describes spends most of a day anchored to one customer, building and revising a prototype — a narrow radius of responsibility, but deep and hard to detach from; the Silicon Valley many-customers type has a wide radius and a high switching cost. [5] Same word, "a day" — completely different contents packed inside it. This is exactly Chapter 18's structural difference, playing out on an individual calendar.

V. A Ceiling That's Widening

Come back to the two numbers from Section I: ten customers, fifty hours. What lets one person carry ten at once isn't just individual capability — there's a factor now shifting underneath it: **tool leverage**. One Silicon Valley practitioner offered a directional read: tools like transcription and internal enterprise search could, in the short term, push how many projects a single FDE can carry at once from two or three up to five or six; over the longer term, the role will split — one end handling the hardest problems as high-end FDEs, the other serving small and mid-sized customers, sometimes never setting foot on-site, as remote FDEs. [6]

This lines up with Chapter 5's "execution-layer devaluation" argument: the same tools pushing up the output ceiling of "a day" are also stretching the role into different tiers. **"A day" isn't a constant — it's a variable that tools are actively rewriting.**

VI. The Honest Truth About How It Feels

Start with the upside. Back to that Silicon Valley engineer from Section I: end-to-end ownership — seeing the problem yourself, solving it yourself, hearing back from the customer yourself — is, for a twelve-year engineer who just switched roles, the most direct payoff. [2]

The downside runs two layers deep. The first sits on the client side: misconceptions about what AI can do, mismatched tech stacks, code that can't be delivered end to end, no security guarantees — endless rework, all of it landing on the front-line team. [7] The second layer travels down the responsibility chain to the vendor side: the client's round-the-clock grind becomes the FDE's midnight WeChat pings, the system that's down and needs saving, the exhaustion of switching between customers all day long.

[2] Hacker News, "A Week in the Life of a Forward Deployed Engineer" (10 customers, 50 hours)

[6] Silicon Valley 101, Episode 240: "The Hottest New Job in Silicon Valley — FDE" (YouTube)

[7] "AI Job Sense: Stop the Agent Rat Race"

Same job — heaven for one person, hell for another. Only two variables draw the line: **tolerance for ambiguity, and the need for ownership.** People who can absorb interruption, live with vagueness, and genuinely need "this thing to be mine from start to finish" thrive in this work; people without those traits are worn down by it, continuously. These two variables are the last gate in next chapter's capability model.

VII. Thinking About This Path? Try Three Things First

You don't have to quit your job to find out. To test your own fit, try running these three scenarios where you already work:

Deliver something real for an internal user. Find a cross-departmental internal-tool need and carry it from requirements interviews through to running it live in production — feel what it's like to deliver when "the user is sitting right next to you."

Sit through one full customer engagement. From presales to post-sale, count how many "problems that aren't actually technical problems" show up in a single meeting.

Take over a system that's already live, and run it for a month. Even if it's just being on call — every alert after go-live, every "why doesn't this work anymore," is the daily texture of this job.

Run all three, and on both variables — tolerance for ambiguity, and the need for ownership — you'll get a more accurate answer than any personality test could give you.

The real texture of a day and a week is settled now. Who can handle it, and handle it well — the next chapter starts with the capability model.

Chapter 24 · What Kind of Person Is Cut Out for FDE?

> The fox knows many things, but the hedgehog knows one big thing.
> — Archilochus, as quoted by Isaiah Berlin in The Hedgehog and the Fox

Ask online what capability an FDE role actually needs, and you get two answers, arguing loudly with each other.

One side: business knowledge can be learned on the job, technical skill can't — technical skill is the hard bar, business is the soft one.

The other side flips it: building is already handed off to agents, so business understanding is the truly scarce thing. [1] Both sides sound absolutely certain. Which one should you believe?

Don't rush to a verdict — they're both answers the same market gave, at two different stages, from two different vantage points.

The real test only comes into view once two frameworks are in place: the three pillars, which say what's required, and the two-layer structure, which says what depreciates and what compounds. By the end, you'll see that each of those competing claims has hold of only one of the two layers.

I. The Three Pillars: An Intersection, Not a Single Specialty

FDE capability sits at the intersection of three pillars. [2]

Pillar one, engineering delivery. Not algorithm research — the engineering that gets something running: integration, permissions, evals, operations, all of Part Three. It's the first rebuttal to the "FDE talks its way through" image: in the on-site samples, engineers put three-quarters of their time into engineering and model optimization; people who only talk don't survive the first wall. [3]

[1] "FDE Engineer 101: Bridging the Gap Between AI and Business" (Chinese-language compilation video)
[2] "100 Questions About FDE" (open-source ebook)
[3] Baseten, "What I Learned as a Forward-Deployed Engineer Working at an AI Startup" (Het Trivedi)

Pillar two, business understanding. Breaking down a customer's vague goal into a process that can actually be measured. It builds in three levels: level one, understanding industry vocabulary; level two, reading the logic of a workflow — why this company's order routes the way it does; level three, the scarcest one, reading incentive structures — whose performance rating rides on this process, whose interests the new system will disturb. That third level is the mirror image of Chapter 16's "three kinds of not understanding": someone who doesn't understand the organization can't do this job at the executive level.

Pillar three, project-driving ability. Knowing who signs off, who's affected, and whose resistance needs handling. One practitioner puts "executive-level understanding" right at the front of the capability map, calling it "the first door" — fail to get through it, and none of the other capabilities have anywhere to go; his metaphor is door, process, muscle — train nothing but muscle, and you're still stuck outside the door. [2] This is the field-operator's version of Chapter 15's three cast members: supporters, influencers, and the people who stand to lose.

The three pillars are abstract; one company's actual hiring bar puts them on concrete ground.

Cresta's FDE lead has one clear "who we don't hire" rule: **no junior FDEs** — a project site usually has only one or two people co-creating with the customer, and someone without enough track record can't build trust. In his hiring language, the three pillars map onto three concrete layers: technically, you have to be "a solid developer who has built and tested AI agents"; on communication, you need "hard-earned, credible customer-facing experience" — able to talk with the other side's CTO, IT director, and security lead, and able to break a complicated problem down for someone non-technical; the rest comes down to being dependable, resilient, and able to carry uncertainty — he prefers founding engineers, people who've been through real turbulence (his term for the ideal hire is a "mini-CTO": hands-on, able to write code, able to talk to customers, thoroughly dependable). [4]

The three pillars are an intersection, not a full house on every subject: **clear the bar on all three, and your strongest one sets your style** — a strong engineering pillar leans delivery-type, a strong business pillar leans discovery-type (the two roles in Section III), a strong organizational pillar leans driver-type. "The full-stack superhuman who does everything" is a misconception — the three strengths each form their own school.

The three pillars aren't a fixed recipe either — the weighting shifts with market structure. The

[2] "100 Questions About FDE" (open-source ebook)

[3] Baseten, "What I Learned as a Forward-Deployed Engineer Working at an AI Startup" (Het Trivedi)

[4] Silicon Valley 101, Episode 240: "The Hottest New Job in Silicon Valley — FDE" (YouTube)

[5] Turning Point (破局点), Episode 31: "FDE, Chinese Style"

Silicon Valley sample puts three-quarters of its time into engineering; [3] a Chinese practitioner who has served more than thirty enterprises splits the job into consulting, sales, engineering, and coaching instead, with engineering ranked third — sixty percent of the time goes to business and relationships. [5] The difference isn't about who's right; it's about whether a product foundation exists (Chapter 18): a strong foundation absorbs the complexity for you, so engineering time naturally converts into delivery; without one, consulting and sales become the bottleneck instead. **What the reader should learn isn't which ranking to memorize — it's how to judge which ranking fits the market they're actually facing.**

One more relationship worth spelling out: the three pillars are the job-level expression of "the composite capability a build needs." The pillar model was never describing a superhuman — it's a breakdown map for a small team's capability. In team form, the three pillars can sit in different roles — Chapter 26's Chinese trio (business analyst, AI architect, business coach/embedded team) is exactly one small team with each pillar as its own long suit; in individual form, all three pillars press down on a single person.

II. Two Layers: What Resets to Zero, What Compounds

More important than the ranking is the time dimension.

The domain layer: an industry's processes, vocabulary, and data shapes. Switch industries and it resets to zero — retail's restocking logic doesn't transfer to finance's risk control.

The meta-capability layer: problem decomposition, baseline measurement, acceptance design, accountability management — the four items already native to this layer. Add one more, deeper still: the capacity to learn itself. Models, frameworks, and industry jargon all change year over year; new terms and new workflows never stop appearing. What resists depreciation most in the meta-capability layer isn't what you've memorized — it's how fast you can absorb something new, and whether you're willing to keep letting new things in. Openness itself is a meta-capability. Cross-industry compounding — Chapter 8's acceptance contract, Chapter 12's eval method — carries over unchanged when the industry changes; new tools and new jargon draw on that same "relearn from scratch" muscle (see Figure 24-1).

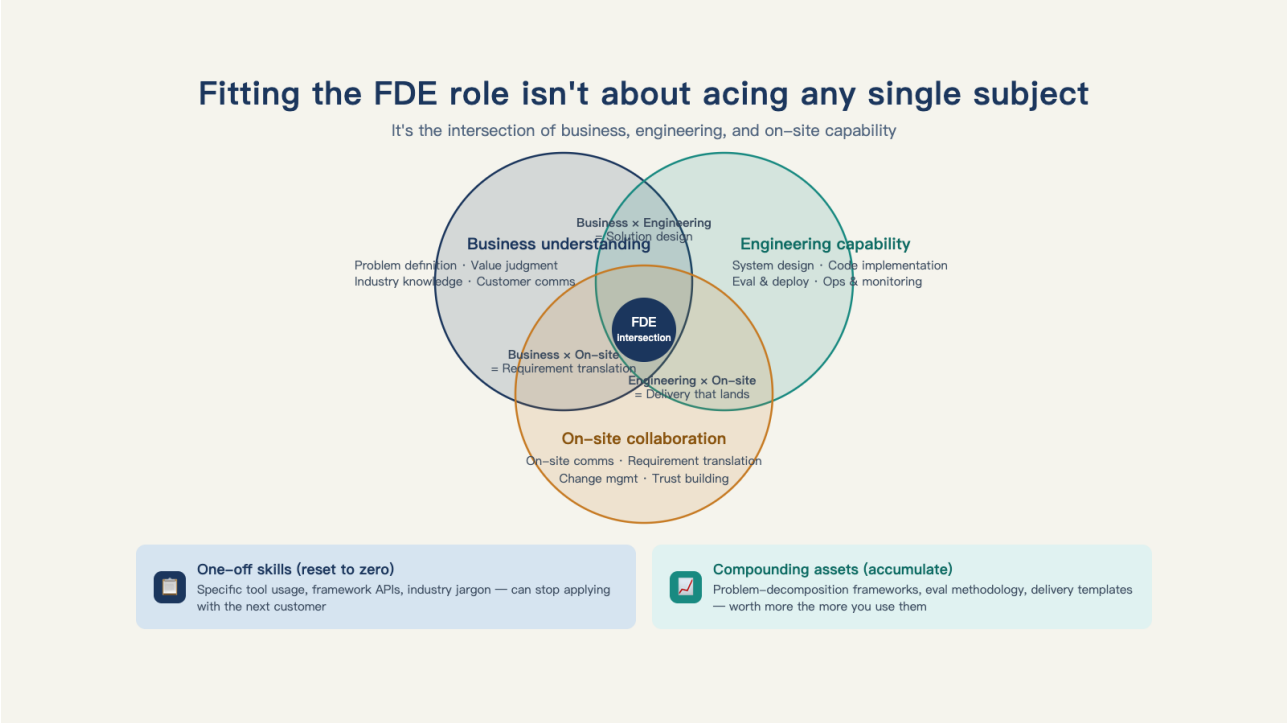

Figure 24–1: Fitting the FDE role isn't about acing any single subject — it's about the intersection of three capabilities and the assets that compound.

The samples back this up: what transferred for the twelve-year engineer who moved into the new role clearly wasn't any industry knowledge — it was engineering meta-capability: production instincts, a nose for where things break, a feel for systems. [6] The same logic holds for an FDE who moves from retail to manufacturing: the domain layer resets, the meta-capability layer travels.

The investment implication is direct: **training and hiring budgets should tilt toward meta-capability** — industry knowledge can be bought (consultants, expert interviews, a few months on-site grow it naturally); meta-capability can't. This is also the answer to the two opening claims. The first, "business can be learned," holds at the meta-capability layer. The second, "business understanding is the scarce thing," holds at the domain layer and the incentive-structure layer at once. Both layers are right — put together, they're the whole picture.

III. Discoverer and Deliverer: Two Kinds of People, One Role

Above the three pillars sits another layer of division inside the role itself — within the same FDE team, discoverers and deliverers are two different kinds of hires. McGrew has sketched both profiles — discoverer and deliverer. Chapter 1 already defined this split; here's what it means for

[6] Hacker News, "A Week in the Life of a Forward Deployed Engineer" (10 customers, 50 hours)

[7] Y Combinator: The FDE Playbook for AI Startups (Bob McGrew)

hiring: [7]

Discoverers (the Echo type): the classic profile comes from the domain itself — his examples are a former army officer, someone who spent years deep in healthcare, seasoned domain veterans. And they have to be rebels: people who understand exactly how things are done today and are convinced it isn't good enough. Why does it have to be a rebel? Because the new software has to deliver a 3x or 10x step change to be worth the effort of building at all — a domain expert who's made peace with the status quo can't see why it has to be that big a jump, and can't talk anyone else into it either.

Deliverers (the Delta type) — and the anti-profile cuts closer to home: a craftsman engineer is the wrong hire. Someone seduced by getting the abstractions exactly right, by code built to stay maintainable for a decade — the deliverer's job is to write a prototype that actually runs, fast, absorb a lot of pain doing it, and accept that **the first version is usually thrown away and rewritten** (McGrew's own words, already quoted in Chapter 7 at the point about what happens to MVD code). Temperament and background have nothing in common with the discoverer's, and the two can't be swapped.

That anti-profile will land hard for engineer readers, but read it correctly: it isn't saying craftsman engineers are bad — it's saying **the craftsman's values are mismatched to this job's delivery rhythm.** A prototype exists to test a problem, not to be handed down through generations.

IV. What a Senior Engineer Brings — and Where the Gaps Are

Put the three pillars and the two roles together, and the checkup for a senior engineer moving into FDE writes itself.

The natural strength is **system-level judgment.** One lesson from the Chinese delivery front line: when packaging a capability for a customer, the hardest part isn't the packaging itself — it's **knowing what to package**, which takes architectural judgment, and needs a company's mid-level or senior staff involved from day one, in an architect's capacity. [8] A decade of accumulated engineering instinct happens to fill exactly this gap.

The common weak spot is project-driving ability: used to answering for code, not used to answering for people; used to writing it yourself, not used to watching someone else write it; and least used to all, starting work with no clear spec — because there is no spec on-site. So **the**

[8] Yilu Tongxing (一路瞳行), Episode 134: "From AI Assistant to Digital Employee"

weight of transition training should sit on the weak spot, not the strong one: sending engineering backbone staff to sit through customer meetings, run requirements interviews, and take a complaint or two does more good than teaching them new technology ever would.

Two companies' hiring philosophies confirm each other, and also pull against each other. Jove Zhong (Cresta) won't hire any junior FDE, only people who have built and tested AI agents; Vincent Lu (Ventus AI) has hired two people still in school — one fresh out of high school on his way to Harvard, one who left NYU without finishing — but he admits himself it "falls a bit short," because at an early-stage company the FDE has to independently carry customer communication and project management, and "without having tripped over a few things yourself, it's hard to develop any sense of project management." [9] Put the two hiring lines together, and they confirm this chapter's point that the three pillars are an intersection, not a full house — but the bar can't drop: Cresta filters on track record, Ventus manages the risk by admitting the gap up front; different paths, but neither dares put someone in the seat who hasn't been tested on project-driving ability — exactly the weak spot this section names, and neither company can dodge it.

The hiring-funnel numbers are worth citing too: Jove Zhong's team has received 7,000 résumés for a single opening, and issued fewer than 20 offers. [9] That number confirms this chapter's opening point — the intersection of the three pillars isn't a framework a writer worked out after the fact; it's a real filter in active use on the hiring floor, and it filters harder than most people would guess.

V. Self-Check: Are You a Fit

Twelve questions to check yourself against — not a personality test, don't treat the result as a verdict. For each, score yourself 2 for "often," 1 for "depends," 0 for "almost never."

1. Given a vague goal ("just get this handled"), I can break out the first step on my own.
2. After getting interrupted, I can pick my train of thought back up quickly.
3. Talking with someone from an unfamiliar business, I can find the thing they're actually worried about.
4. I can walk someone with no technical background through a technical plan clearly enough for them to decide.
5. When there's no right answer, I'm willing to make a call on limited information and own the outcome.

[9] Tencent Research Institute, AI Lens Roundtable, Episode 6: "FDE Non-Consensus Views and a Field Guide from Silicon Valley Founders"

6. I want to be accountable for the result, not just the quality of the code.
7. When a customer calls me at midnight, my first reaction is to solve the problem, not to feel annoyed.
8. I can hold more than three things in my head at once without dropping any of them.
9. Seeing someone else's business process in disarray makes my hands itch to fix it.
10. I can accept that the first version of a plan may get thrown away and rebuilt later.
11. I'm willing to spend time on relationships — meals, small talk, favors — and treat it as part of the job.
12. Starting over from scratch in a new industry excites me more than it scares me.

18 or above: the groundwork for the three pillars and the psychological bar are basically in place — worth trying (Chapter 25 gives the path). **12 to 17**: some of the conditions are there; see where the points are missing and use the three trials from Chapter 23, Section VII, to fill the gap deliberately. **11 or below**: this isn't a capability shortfall, it's an open question of fit — read the second-thoughts checklist in Section VI before deciding.

VI. A Second-Thoughts Checklist

One last thing, held in reserve for balance: three kinds of people should think twice before moving into FDE. Note — this isn't a capability problem, it's a **fit problem** — a mismatch is a lose-lose.

People who prefer deep, focused work on a single problem. This job's default state is high-frequency switching (Chapter 23) — routine for some people, a constant drain for anyone who prefers deep focus.

People who need a clear spec before they can start. On-site, you write the spec yourself, and it changes once you have. Waiting for clean instructions is a luxury this job doesn't offer.

People willing to answer only for code, not for results. This job is graded on whether the customer's business actually changed — code is just the means. Anyone who treats the means as the whole boundary of their responsibility will go quiet in every retro where "the system worked but the business didn't move."

VII. The Company's View: If You Can't Buy It, Grow It

Beyond the individual view, look at FDE from the company's side.

[10] "Anyone Out There Actually Doing FDE Work?" (linux.do thread)

Composite talent is either too expensive or not up to par — most companies end up having to grow it themselves. [10] The internal training path matters more than hiring here: **engineering backbone + on-site rotation + shadow delivery** — pull people with strong meta-capability roots out of the engineering team, put them on rotation with real customers for one or two quarters, have them shadow a senior FDE through the full process first, then let them independently carry one small customer's complete loop.

The capability model is set. Next comes a very practical question: for an engineer already on the job, how do they actually walk the path from writing code to owning results — the next chapter is the transition path.

Chapter 25 · How Should an Engineer Make the Transition to FDE?

> Scars signal skin in the game.
> — Nassim Nicholas Taleb, Skin in the Game

Two thirty-day promises.

One lives on a Silicon Valley social platform: a post titled "How to Become a Million-Dollar Forward Deployed Engineer in 30 Days," 360,000 views, complete with a thirty-day roadmap and salary tiers. [1] A Chinese "FDE training" syllabus can be summed up in one phrase: hard to even describe. Lesson one teaches how to print Hello World on your computer, not a shred of hands-on delivery training anywhere in the course, a lineup of "famous instructors," and a price tag of twenty thousand yuan. [2]

The anxiety around transitioning is real, and the training market that grew up around that anxiety is the first place the bubble is inflating. This chapter won't repeat "how to prepare" — that topic has been written to death. It only covers two questions people usually skip: **how to judge it when you're the one being tapped for the move, and what employers are actually looking for when they hire.**

I. Paths of Transition

Five backgrounds, five different shortcuts, five different traps. Coming from backend: engineering delivery is already there; what's missing is reconstructing the business and driving the organization. Coming from product management: the business and organizational grounding is there; what's missing is end-to-end engineering feel. Coming from consulting: customer communication was never the problem — the hardest step is turning "hand over a proposal" into "hand over a whole system that runs." Coming from data analysis: the eval and data grounding is there; what's missing is system integration and owning accountability. Coming from algorithm engineering: the most common mistake is overrating the model and underrating integration — of Chapter 9's eight walls, the model doesn't even count as one of them; integration and data

[1] "A million dollars in 30 days" course-marketing post (@gregisenberg)

[2] Turning Point (破局点), Episode 31: "FDE, Chinese Style"

permissions are the walls you hit first.

One line applies across every background: **the ticket into this transition isn't finishing a course — it's producing one piece of delivery evidence you can actually show someone.**

II. Tapped for a Transfer: Opportunity or Trap?

Most writing on transitioning assumes "you want to move." In reality, a lot of transitions are pushed from above. One backend engineer at a foreign company wrote about it plainly: he was fluent with AI tools and a decent communicator on his R&D team, so his manager wanted to move him into FDE — but he liked the technical work, and "the moment I think about going down this path, it feels like I'm drifting further and further from the technology," a reluctance he couldn't shake. [3] Facing this, the move isn't to rush toward "accept" or "refuse" — it's to run three checks first.

Step one, run the three-way test, using the standard from Chapter 2. Ask three questions: Does my code go to production? Where does field experience flow back to? What does this role get measured on? Get clear answers to all three, and it's a real FDE role; hedge, or hear only "let's just see how it goes," and it's probably implementation wearing a new label.

Step two, test whether the organization actually backs it. Does the company have a real business problem it needs solved, or is this just "everyone else has one, so we should too"? Is there a business-side sponsor willing to invest? There's a more direct signal too: which department the role reports into, and to whom. Report into product or engineering, with revenue booked against the platform product, and FDE success equals product adoption. Report into sales or delivery, billed by the day, and the role easily degrades into a presales labor pool — a fancier name for implementation. [4] [5] In an interview, you can ask outright who the FDE role reports to — more reliable than guessing yourself. A solo FDE — the only one in the whole company, no feedback channel, no business-side interface — is a high-risk setup: succeed, and it's your personal win; fail, and there's no organizational backstop.

Step three, test whether you're actually a fit (the twelve questions from Chapter 24). The score itself isn't the point — what matters is whether that worry about "drifting from technology" holds up. A real FDE role's engineering content isn't thin — Chapter 23's three-quarters-time-on-engineering figure is the evidence. What you'd actually be drifting from

[3] Switching from R&D to FDE: A Step Up, or Just Moving Into Implementation? (linux.do thread)

[4] "100 Questions About FDE" (open-source ebook)

[5] Fan Bing, Forward Deployed Engineer (FDE) (XDash open-source book)

is deep-focus, algorithm-research-style technical work — so first figure out which kind of "technology" you actually care about.

Run all three, and there are only three outcomes: pass all three, and go all in; the role is real but organizational support is weak, so negotiate boundaries — get a business-side interface and a feedback mechanism in place before accepting; the role itself isn't real, so switch internal tracks instead — don't bet your career on a title that's just old work in new clothes.

III. What Employers Actually Look For

There's another angle people often skip: what makes a company willing to give someone changing careers a shot. That decides where the transitioning engineer should actually spend their effort.

Evidence that counts as a signal: a real delivery made for an internal user — real users, real operational traces after go-live; a track record inside a cross-departmental project — meeting notes showing you were the one who saw it through; having maintained a system after it went live — handled alerts, taken complaints, shipped iterations.

Evidence that doesn't count as a signal: course certificates; a demo reel with no real users; "I've learned framework X" — frameworks go stale fast, meta-capability is what sticks (this is Chapter 24's two-layer structure showing up in hiring: what an employer is buying isn't which framework you know, it's whether you've made an architectural call yourself when nobody handed you a spec) (see Figure 25-1).

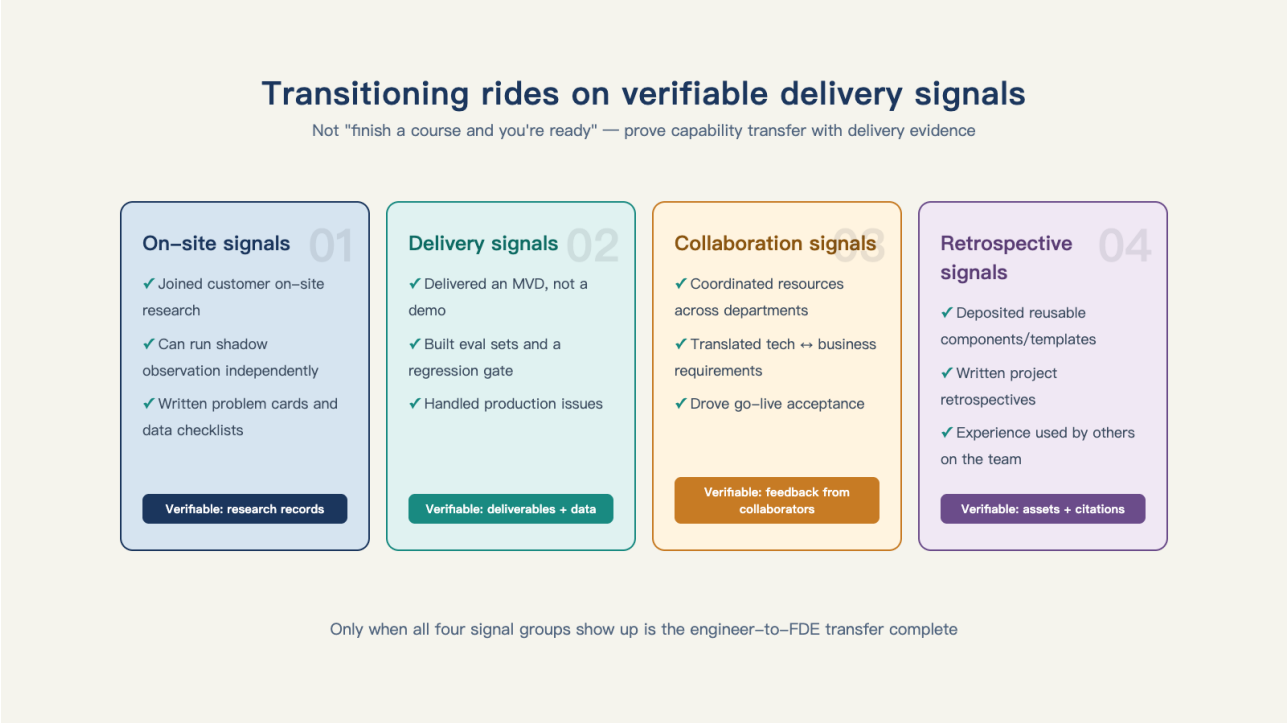

Figure 25–1: Transitioning rides on a chain of verifiable delivery signals, not on writing the job title into your résumé.

So "finish learning first, act later" is a strategic mistake: learning is necessary, but what learning produces — certificates, course assignments — is exactly the least convincing signal. **Producing one piece of delivery evidence you can show someone, in the job you already have, is worth more than any course** — which is also why the three trials from Chapter 23, Section VII (internal delivery, a customer meeting, running a live system for a month) double as a transitioning engineer's résumé-building project.

IV. The Training Market: How to Tell If It's Real

Back to the two "thirty days" from the opening. The Chinese and American samples are saying the same thing: transition training is the first link in this chain where the bubble inflates — the anxiety shows up before the demand matures, and anxiety has always been the easiest thing to sell. [1] [2]

Choosing a course or a bootcamp comes down to one hard test: **is there real-project practice.** Training built on real customers, real delivery, real acceptance is worth the price even when it's expensive; without that, the prettier the syllabus, the more suspicious you should be — "thirty days to a million dollars" and "lesson one, print Hello World" are, underneath, the same business.

[1] "A million dollars in 30 days" course-marketing post (@gregisenberg)

[2] Turning Point (破局点), Episode 31: "FDE, Chinese Style"

One more note of caution: the practitioner quoted above who criticizes the training bubble runs his own bootcamp, and its selling point is exactly "real-project practice" — his opinion lines up with his own business interest, so hear the argument on its merits, but keep a clear head. [2]

V. The Cheapest Way to Test It

You don't need to quit to test this path — the three scenarios from Chapter 23, Section VII (internal delivery, a cross-departmental project, a customer meeting), are already the cheapest way to try it out; there's no need to design a separate approach. Run them, and you'll walk away with three things at once: a piece of evidence you can show someone (Section III), an answer to whether you're a fit (Chapter 24), and an answer to whether you'd actually be drifting from technology (Section II).

VI. Pay Cuts and Expectations

One last honest question: does moving into this role mean taking a pay cut? The market can't give a trustworthy number — pricing for this transition is still in a chaotic phase; the same city, the same title, and the spread can be absurd. Instead of a number, here's a structural judgment: **in a chaotic phase, the difference between individuals matters more than the difference between titles** — how solid that piece of delivery evidence is affects your market value more than the three letters "FDE" ever will. So the anchor for any negotiation shouldn't be the title — it should be the evidence. Walking in with a record of internal delivery and walking in with the sentence "I can do FDE work" put you in two entirely different positions.

The individual side of the path is complete. Now look across the table: when does a company need its first person like this, where do they put them, and how does the team grow from there — the next chapter takes the employer's side of the question.

Chapter 26 · How Do You Build an FDE Team?

> There is nothing so useless as doing efficiently that which should not be done at all.
> — Peter Drucker

When a company's AI project can't get off the ground, the first reflex is usually the same three questions: Is the model not strong enough? Add a couple of algorithm engineers. Are the people wrong? Swap out the team. Are the tools falling short? Buy a whole platform.

The money gets spent, the people get swapped, the tools get upgraded — and the project is still stuck exactly where it started.

This chapter makes the case that what's actually missing may be a role that owns the result, start to finish. Once that's diagnosed, the answers to when to hire, how many, and how a team grows from there tend to surface on their own.

I. Diagnose First: What's Actually Missing

Behind that opening three-question reflex — model not strong enough, people not right, tools not keeping up — sit at least three different diseases, and they call for completely different medicine.

Cause one, the wrong problem was chosen. You picked a problem not worth solving (it never cleared Chapter 6's five filters) — this is a direction disease. Swapping people doesn't fix it; re-choosing the problem does.

Cause two, the engineering walls didn't get scaled. The problem and the direction are both right, but people are falling on Part Three's eight walls — data and permissions never got sorted, integration never got wired through, outcomes can't be proven, results can't be turned into trust — this is an engineering disease, and what it needs is delivery engineering capability.

Cause three, nobody owns the result. The system went live and nobody uses it — this is an organizational disease, and what it needs is a role that owns the result end to end.

Of the three causes, only the first might call for swapping people. Most of the time the right answer is **adding a role, not replacing a team.**

[1] Accenture Newsroom, "Accenture Launches Microsoft Forward Deployed Engineering Practice to Help Organizations Scale AI Across the Enterprise"

A recommended diagnostic tool: run a post-mortem on failed projects. Take failed projects one by one and attribute each against the eight walls (Chapter 9) plus the four responsibility segments (discovery, production engineering, business adoption, product feedback) — attribution clustered around "couldn't build it" points to a model-and-engineering-capability problem; clustered around "nobody uses it" points to a missing owner of the delivery loop. The same diagnosis made it into the official language of Accenture and Microsoft's 2026 announcement of a joint forward-deployed engineering practice: what's stalling enterprise AI isn't technology — it's having someone who can put engineering capability in the right place. [1]

II. Three Signals on Timing

Once the gap is diagnosed, the second question is whether to build now. Wait for three signals before hiring.

Signal one, there's a clear business problem. Not "let's explore AI a bit" — a problem you can actually state: which step, what it costs, who's under pressure about it (Chapter 6). "We want to do AI" is where a budget starts, not where a team starts.

Signal two, there's a business-side sponsor willing to invest. Executive backing, the business side contributing people, data, and time (Chapter 16's two gates) — delivery without an internal ally is a parachute drop, and parachute drops have a very low survival rate.

Signal three, you've already run one trial. Even if it was an outside vendor's Minimum Viable Deployment (Chapter 7) — the company has at least seen once what "problem to production" actually looks like, and knows what it wants.

When the three signals aren't all there yet, the right move isn't to hold your breath for headcount — it's to **buy a small service first**: an outside assessment, a single-point delivery, letting someone else's delivery help you assemble the missing signals (that market gets its own treatment next chapter). Building a team is a heavyweight commitment; don't stake it on uncertainty.

III. The First FDE: Profile, Placement, Evaluation

Once the signals are there, hire the first person. Three decisions determine whether it works.

[2] "I Thought FDE Meant Fighting Alone at the Customer Site" — What Two Months on the Job Actually Looks Like (note.com · AI Shift)

Profile: senior, not junior. The first FDE needs strong engineering–delivery capability plus strong business–reconstruction capability (the first two of Chapter 24's three pillars need to be strong; the third can be developed on the job). It can't be a junior hire — a junior needs a team around them to grow inside: the new hire from Japan in Chapter 23, still two months into her customer–confirmation phase, had a team standing behind her; [2] drop that same person straight into an unsupported solo post, and you're leaving her to thrash alone in deep water. The first FDE has to be someone who can build their own scaffolding.

Placement: the business–delivery line, not a pure technical line. Report to a business or delivery leader, not a research line — the reporting line decides what gets responded to first. At the same time, define the interface with the product team from day one, even if it's only a once–a–month feedback review (the minimum version of Chapter 22's mechanism). Without that interface, the first hire slowly turns into an outsourced team. [3]

Evaluation: on results, not hours. Acceptance metrics plus adoption metrics (Chapters 8 and 15), not lines of code or day–rate utilization — misaligned evaluation turns an FDE into an ordinary developer. This is the organizational version of Chapter 2's third test: someone who sells time isn't accountable for a result; whatever you measure is what the person becomes.

IV. Outside Reference Points (I): Two Roles, Three Functions

There's no standard answer for building a team, but there are a few highly credible reference points. The first is two modern shapes for splitting the roles.

The discoverer/deliverer pair (the profiles quoted in Chapters 1 and 24): the Echo–type on–site analyst owns discovery and relationships; the Delta–type deployment engineer owns delivery — hire the two roles separately, by profile, and don't expect them to swap. [4]

Three functions and an account model: one applied–AI company organizes itself into three blocks — engineering (platform and infrastructure), go–to–market (the acquisition funnel), and applied AI (the bridge: catch a warm opportunity, define proof of value, get the agent into production). Applied AI then splits into two roles of its own — the **Agent PM**, who carries the account from proof of value all the way through production and beyond, technical enough to build an agent themselves without having to write production code; and the **forward deployed**

[3] @deployengineer (Bhaulik Patel), essay series

[4] Y Combinator: The FDE Playbook for AI Startups (Bob McGrew)

[5] Inside an Applied AI company (Pace long–form post)

[6] Silicon Valley 101, Episode 240: "The Hottest New Job in Silicon Valley — FDE" (YouTube)

engineer, who writes code, feeds lessons back into the product, and needs enough commercial sense to sit directly in sales conversations — because "the person who's actually going to build the thing is the most credible voice in the room." [5] Another company's variant (Cresta) gives a staffing ratio: one forward deployed product manager runs two to three FDEs, and a PM can manage multiple projects at once; once the team scales, they also deliberately grow specialists by industry (healthcare insurance, say) and by technical domain (payments, search) instead of expecting everyone to be a generalist. [6]

Put the two samples together, and the lesson is: **"the one who understands the customer" and "the one who can deliver" should be two clearly defined roles from day one** — either two separate positions, or one position wearing two explicit hats. Pile all those requirements onto one person and expect them to be great at everything, and you'll struggle to hire anyone at all.

V. Outside Reference Points (II): The Chinese Version — the Trio, and the Extreme Case

The second set of reference points comes from the Chinese market and from the giants. The Chinese version gives a more complete answer for building a team: not splitting the work in two, but fusing three specialties into one small unit.

The deepest version comes from Lingyang — Alibaba's enterprise AI unit — and its "organizational trio" (background and case details in Chapter 18). Their view: "our approach isn't one person, it's an organization" — finding one person with all that talent is "extremely hard." [7] The organization splits into three roles: the **business analyst owns direction**, driving toward business outcomes — orchestrating a 260-plus-step customer-service workflow, finding AI's best working pattern, setting eval standards; these people come from industries like beauty, appliances, and automotive. The **AI architect owns the technology**, translating business problems into model selection and system design — what size of model to use, which step needs a human in the loop, which step the machine takes over. The **customer-side coaching team owns the business**, an embedded coaching function — the best customer-service reps and top sales performers are pulled from the customer's own staff to give continuous feedback and coach the AI, with the vendor deliberately stepping back here. [7]

Set this beside Palantir's Echo/Delta split, and the structure is the same underneath, cut differently:

	Palantir Echo/Delta (U.S. version)[^c06]	Lingyang's Trio (Chinese version)[^b19]
Direction: what to do	Echo — on-site analysts and account managers	Business analyst: outcomes, process orchestration, eval standards
Technology: how to build it	Delta — deployment engineers	AI architect: model, data, system design
Business: how to use it well	Implicit in Echo's relationship work	A formal third role: customer-side coaching team, continuous feedback

The difference concentrates in the third row: the U.S. version splits in two (discovery/delivery), and "business coaching" is an implicit part of Echo's job; the Chinese version promotes it to a formal, third role — because the benchmark business knowledge is scattered on the customer's side (Chapter 18's "the benchmark performer as the default"), and without folding the customer's own people into the build, the model never gets calibrated right.

A more elastic version of this also comes from the Chinese market. One practitioner has said that what a Chinese enterprise doing AI transformation actually needs is three roles — project manager, engineer, product manager — and those three roles can be one person, two people, or a whole team; there's no requirement to formalize headcount. [8] At bottom it's the same point: you need someone who understands the business and can translate requirements for the AI engineers. How the three roles get combined depends on company size and how strong the signals are (the three signals from Section II) — with the signals just barely in place, one person wearing all three hats can still get things started; once the signals are solid, split it into three positions. The trio is the standard form; the elastic combination is its cut-down version.

The extreme case: Accenture and Microsoft's joint practice — a scale of thousands of AI-skilled engineers, official language calling it taking AI "from idea to production in days, not months," acting as "the gateway for enterprise AI transformation." [1] A reference point, not a template to copy — thousands of people is the shape delivery takes after it's been industrialized, not the shape it starts in.

Set the two sets of reference points side by side, and two shared points stand out. First, **the complete form of FDE isn't one person, it's a built team** — special-forces style: small, multiple specialties fused together, embedded on the front line, accountable for the mission's result. A solo FDE is that team's cut-down form; Section III's "the first FDE" is exactly about which pieces to keep when you cut a team down to one. Second, **an FDE has to have an institutionalized**

[1] Accenture Newsroom, "Accenture Launches Microsoft Forward Deployed Engineering Practice to Help Organizations Scale AI Across the Enterprise"

[3] @deployengineer (Bhaulik Patel), essay series

[8] Turning Point (破局点), Episode 31: "FDE, Chinese Style"

feedback channel into the product team — without one, delivery quietly turns into outsourcing, a warning that holds across every market. [3]

VI. How a Team Grows From One Person

Once the first hire is standing on solid ground, how does the team grow?

Step one, get the smallest possible loop running. One FDE, one product-side point of contact, and half a platform-support person (a share of a backend/infrastructure engineer's time) — run two or three complete deliveries through it. What this step validates isn't the person, it's the loop itself: problem definition → delivery → adoption → feedback, the full chain walked through once, inside the company.

Step two, scale against the bottleneck, not against a plan. Deliveries are queuing up — add an FDE. The same kind of customization keeps recurring (Chapter 21's trigger signal is lit) — add product-side headcount and fold the customization into the platform. So many customers that service quality is slipping — first ask whether it's a people bottleneck or a feedback-loop bottleneck (if it's the latter, adding people only adds chaos). Review every step of scaling against Chapter 19's six metrics — scale by watching the direction of the implementation-leverage ratio and the customization-decline rate, not by feel (see Figure 26-1).

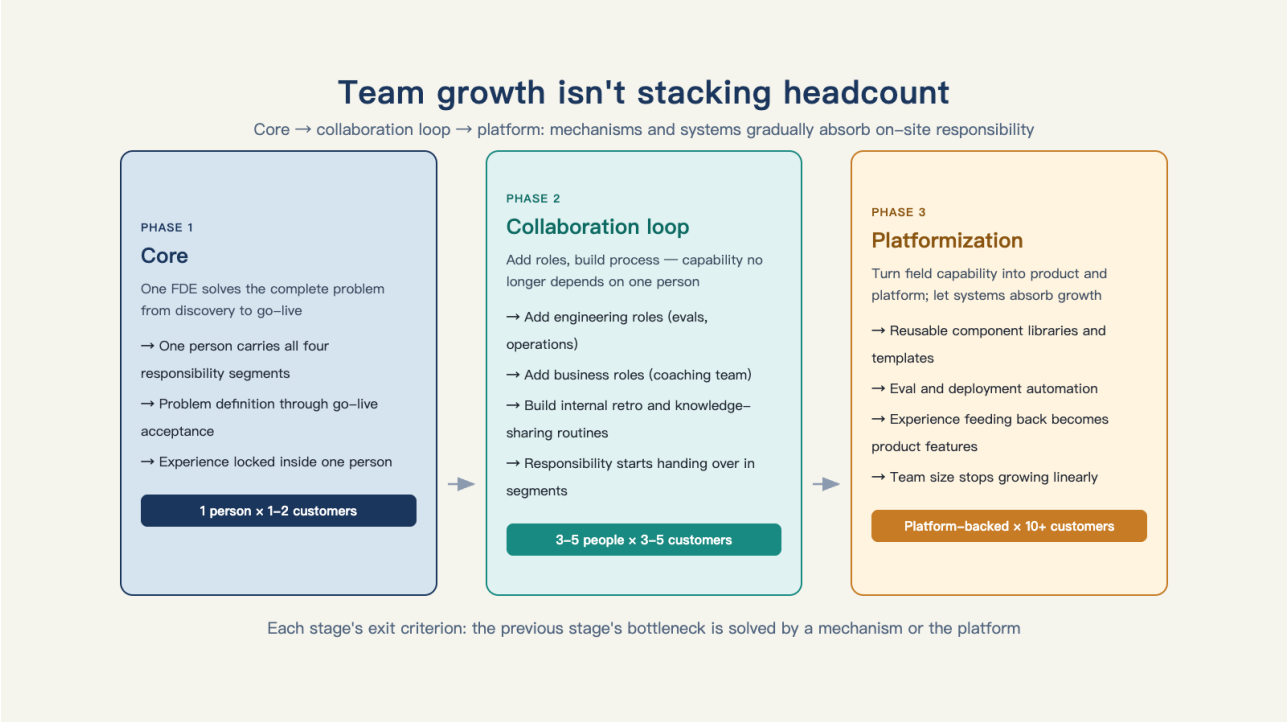

Figure 26-1: Team growth isn't stacking headcount — it's letting collaborative mechanisms and platform capability gradually absorb what used to be on-site responsibility.

The reverse of this path is "hire a twenty-person delivery department in one shot" — headcount arrives before the loop does, and the organization hardens into a headcount-heavy shape before it ever develops the habit of feeding lessons back. Chapter 19's hero-culture trap and consulting-style trap both grow out of exactly this shape.

VII. Three Anti-Patterns

The close, as always, is a mirror.

Chasing the trend into a headcount line. A competitor has an FDE, so we need one too — the organizational version of Chapter 3's "outsourcing in a new coat, a passing fad." A role created without the signals in place always ends up as old implementation wearing a new title.

Hiring and then neglecting. You hire the person, hand them a customer, but give them no business-side interface, no feedback channel, no results-based evaluation — miss two of those three, and you're training talent for your competitors. The resignation letters from people like this are usually written politely; the real reason never makes it onto the page.

Managing FDE the way you'd manage a consulting firm. Evaluate by the day, schedule by utilization rate, treat feedback time as "not real work" — every management move borrowed from a consulting firm, while still expecting a productized result. Whatever you measure is what you get; manage by hours, and hours are what you'll get.

The company's side of the loop closes here. One group is left: people who don't join a company at all, who work on their own — can an individual go independent, part-time, or found a business doing FDE work? Next chapter.

Chapter 27 · Can an Individual Do FDE Alone?

> Knowing the technical work of a craft is not the same as knowing how to run a business that profits from it.
> — Michael Gerber, The E-Myth Revisited

"FDE work — go independent and you have no one to vouch for you; stay employed and it's grinding."

One independent practitioner left that as his own summary on a forum. He calls himself an OPC — One Person Company. With no backing, the only path he can see is to stay humble and first accumulate enough experience, solution patterns, and infrastructure; the post ends with him looking for partners: to explore client sources, methodology, and infrastructure together. [1]

That sentence is exactly this chapter's question: can one person actually take on enterprise customers as an FDE? Which engagements can you accept, and which ones become traps the moment you do?

I. Splitting One Question into Two

The question "can an individual do FDE" is actually two completely different questions bundled together.

Question one: can one person complete a narrowly scoped prototype or evaluation? Yes — and AI has significantly amplified that ability. A skilled person using off-the-shelf models, tools, and frameworks can produce an evaluable prototype for a single customer within a few weeks; that is entirely feasible today (Chapter 24, on how the pieces assemble).

Question two: can one person sustainably carry production-grade delivery for multiple customers? Extremely hard. Production delivery isn't one skill — it's **parallel accountability**: business discovery, system integration, security and compliance, engineering delivery, customer communication, training and adoption, live support, product feedback — eight responsibilities

[1] "Doing FDE Work in Wuhan, Looking for a Partner" (linux.do thread)
[2] Crossroads × Rolling AI, "A Conversation on FDE" (podcast)

hung on a single person, and if any one of them breaks inside a customer's production system, someone has to stand behind it. Live support means being there when the customer's system alerts; running multiple customers in parallel means being there when three customers alert at the same time. A 7×24 responsibility structure is something one person mathematically cannot carry — delivery goes far beyond writing code; see the three exit conditions: business integration, knowledge governance, and system integration, [2] every one of them a continuous responsibility.

Seen from another angle, it's clearer still: these eight responsibilities were never designed for one person in the first place. The complete form of FDE is a built team (Chapter 26's trio is one way to staff it) — and the root reason the OPC doesn't work is precisely that it compresses a team into a single person.

So the answer isn't "yes" or "no." It's this: **an OPC is not logically impossible, but as a business proposition it cannot be run as the mainstream organizational form.** Whether it can be made to work depends on the model and conditions you choose — which is exactly what the two tables below are for.

II. The Father of the FDE Model, With a Warning: "Don't Try This at Home"

Asked what people thinking about trying this playbook should do, McGrew's advice is one line — don't try this at home: "it's probably bad for you — you're probably going to end up doing services. Only if you really try hard not to, and fail — then maybe it actually is a moat for you, if it's the only thing that can possibly work in your market." [3]

As a warning, this lands even harder on the individual than on the company: **the default outcome of doing FDE alone is becoming a one-person services shop** — selling yourself by the day, carrying unlimited liability, with no product leverage and no reusable assets.

But note the structure of the sentence: it doesn't say "impossible." It says "the default outcome is bad — unless specific conditions are met."

What are the conditions? Exactly what we're about to cover.

III. Four Models: The Conditions Table

Model	Preconditions for it to work	What it must not be confused with
Fractional FDE / consultant (fixed-cadence commitment, e.g., two days a week)	The customer has a clear decision-maker, the system boundary is limited, the customer side has an assigned maintenance owner, and the goals are quantifiable	Round-the-clock backstopping of results
Fixed-scope evaluation / go-live readiness audit	Deliverables, data access, and the boundary of responsibility spelled out in the contract	A full delivery project
Vertical FDE studio	Reusable assets (connectors / evals / templates) accreting out of deliveries for a handful of customers	Day-rate, on-site outsourcing
FDE-led product startup	Validate the workflow first with a small set of design partners, then extract the repeating parts into a product	An endless pile of infinitely customized projects

The four models are ordered from lightest to heaviest responsibility. The first two **carry no production responsibility** — the deliverable is a report or a prototype, and the engagement ends at handover. The last two carry partial production responsibility, but hedge the single-person limit with assets (a reuse library) or organization (partners).

The most valuable column in every row is the right-hand one — **the thing it must not be confused with is precisely the standard way an individual engagement goes off the rails**: you negotiate a model-one price and end up doing backstop work; you sign an evaluation contract, and a few weeks in the customer says "while you're at it, why not run the go-live too." The model's boundary has to be spelled out in the first email — not discovered to have dissolved in month three.

Read the practitioner from the opening back through the conditions table: three years of engagements, a self-built case library, reusable modules — **the asset column already has the embryo of model three (the vertical studio)**, which is exactly what he means by "accumulating infrastructure." The blockers are just as clear: backing (why would a customer trust one person) and channel (where does the work come from) — hence staying humble, looking for partners, posting in the community. [1] The path he's instinctively chosen lines up exactly with the conditions table: one person can't carry all the responsibility, so use asset reuse plus a partner network to split the responsibility out. One community sample isn't a conclusion, but it genuinely

[1] "Doing FDE Work in Wuhan, Looking for a Partner" (linux.do thread)

exists: the model table isn't theory — it's a map of a road someone is walking right now.

Ventus AI is a ten-person, Series A company, and co-founder Vincent Lu is himself an FDE. The early shape he describes is almost a hybrid of model three (the vertical studio) and model four (the product startup): the FDE follows the work from presales through post-sale and carries commercial targets directly — "save the customer two million, or help them earn two million more." That is this chapter's "eight responsibilities hung on one person" argument with a name attached. What's even more worth recording is his on-site strategy: Ventus's FDEs mostly don't embed on-site, and the reason is that they are "very clear about what they're providing the customer and how much they can charge for it." He distinguishes two motives for embedding: the good motive is giving a large customer a sense of security, with a clear delivery goal; the bad motive is "I don't know what I'd have you do, but you're a big customer — let's grab the seat first." The second kind he judges outright unhealthy — it produces no value. [4] The community practitioner from this chapter's opening proves that someone really is walking the path the conditions table describes; Ventus AI proves what the path looks like once it works — a control group that can quote specific numbers, specific customer relationships, and a specific business model.

IV. Which Customers Are Yours: Market Segmentation

Choosing customers starts with customer size. [5] Large enterprises (several hundred employees and up, revenue in the hundreds of millions) are the territory of the big platform vendors' multi-million deals; a small team can at most pick up the accompanying training. Micro-businesses have budget for one round of training or consulting at best. **The best entry point for an individual is the mid-sized enterprise** — a few dozen to a few hundred people, revenue starting in the tens of millions: the decision chain is short (the boss can sign off), and a single problem carries high value (his own reported numbers: a problem worth ten million easily justifies a fee of one million). The selection framework is a quadrant: high or low value × fast or slow decisions — **take the "high value × fast decision" cell**; of the other three, either they can't sustain you or they'll drag you down (see Figure 27-1). [5]

[4] Tencent Research Institute, AI Lens Roundtable, Episode 6: "FDE Non-Consensus Views and a Field Guide from Silicon Valley Founders"

[5] Turning Point (破局点), Episode 31: "FDE, Chinese Style"

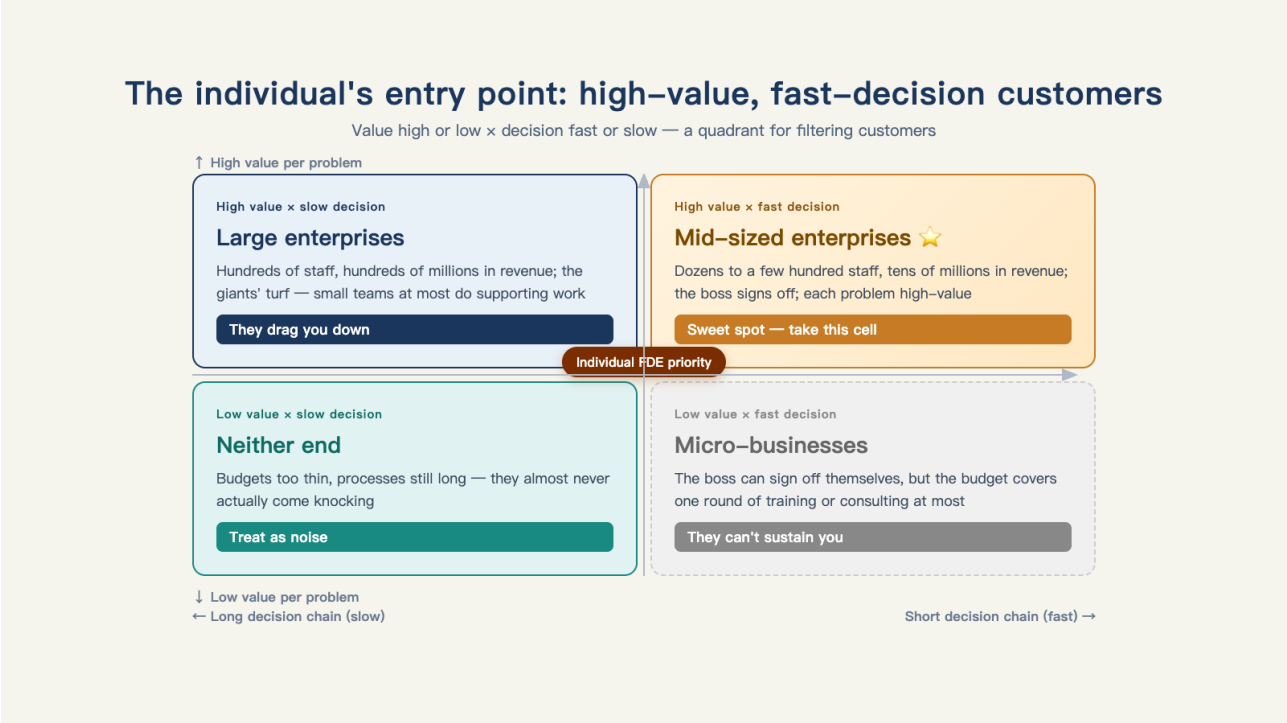

Figure 27–1: The individual's priority entry point is the high–value, fast–decision customer — not the most famous one.

V. The Audit: An Individual's First Sellable Product

If the quadrant has picked your customer, what do you sell first? One practitioner gave an answer specific down to the motion: **the process audit.** The method: get yourself seated next to the person in the customer's company with the most repetitive job, and watch for an hour; then, following Chapter 6's approach (record and probe → filter → quantify), produce an audit report — the real workflow of one link in this company, which steps are suited to handing to AI, and what the economic value is. [6]

The audit is the service **enterprises genuinely pay for**; it is **the first phase before any project gets built, and the biggest bottleneck**; and the report is itself your showcase — when applying for an FDE job or pitching a business, it proves you have seen real workflows. [6] Check it against the conditions table: the audit is the standard product form of model two (fixed–scope evaluation) — **no production responsibility, and it naturally sidesteps the responsibilities among the eight that a single person can't carry**. The complete entry path for an individual thus takes shape: audit (see the real workflow) → narrow prototype (model two, deepened) → join a partner network (Section VIII) or settle into a studio (model three).

[6] "FDE Engineer 101: Bridging the Gap Between AI and Business" (Chinese–language compilation video)

VI. A Stop List Before Taking the Work

Turn it around: which engagements should you not take. Six warning signals; hit two or more, and either decline or restructure (fix a fixed scope, bring in a partner to share the responsibility):

First, the customer has no clear decision-maker. Three months in and still "let's align internally" — neither the ability to pay nor the ability to decide exists.

Second, the system sits on a critical production path and demands uninterrupted operation. A 7×24 responsibility collides head-on with the single-person reality (Section I).

Third, you're asked to sign off on security and compliance alone. Signing is an organizational act; an individual's signature means individual backstopping — unlimited liability.

Fourth, the customer has no internal maintenance owner. No point of contact for post-launch maintenance means that person defaults to you — free of charge, forever.

Fifth, "let's just start and see" — requirements with no boundary. Where no scope exists, price is meaningless (Chapter 20).

Sixth, an on-site invitation with no articulable goal. A good embedding happens when the customer has already worked out what is to be delivered and wants to give a large customer a sense of security; the bad kind is "I don't know what I'd have you do, but you're a big customer — let's grab a seat first." [4] If you can't tell which one it is, ask before you accept.

VII. Part-Time: Does It Count, and Which Kind Are You Selling

One more thing to settle while we're here: part-time. **Part-time prototyping and evaluation is workable** — models one and two are bounded in scope and duration anyway. **Part-time production delivery is not being responsible to the customer** — this isn't conservatism, it's an inference from the responsibility structure: live support and part-time are incompatible in time, and when a customer buys production delivery, part of what they buy is "you'll be there when something breaks." Decide which kind you're selling before the price conversation, not after.

VIII. Three Ways Out for Those Who Don't Go It Alone

[4] Tencent Research Institute, AI Lens Roundtable, Episode 6: "FDE Non-Consensus Views and a Field Guide from Silicon Valley Founders"

Last, the structured options for people who don't want to go it alone.

First, join a partner network. The system run by the practitioner quoted in Chapter 26 is a living sample: a core team owns client acquisition and acceptance, and outside consultants collaborate project by project. [5] For an individual, this solves the two most fatal gaps of going alone — backing (the network's brand vouches for you) and channel (the work comes from the network); the price is giving up part of your pricing power.

Second, join an established organization. The giants are building delivery organizations in the thousands (Chapter 26's Accenture × Microsoft shape) — for an individual, joining one gives more leverage than going alone: the same capability, standing on someone else's platform, brand, and feedback channel. [7]

Third, make the studio vertical. Anchor to one industry, use three to five customers to cut all the way through the workflow (Chapter 21), and let the product compound.

Going alone, joining an organization, or working through a network — walk this road five, ten years, and where does it lead? The final chapter: the endgame.

---

[5] Turning Point (破局点), Episode 31: "FDE, Chinese Style"

[7] Accenture Newsroom, "Accenture Launches Microsoft Forward Deployed Engineering Practice to Help Organizations Scale AI Across the Enterprise"

Chapter 28 · What Is the Endgame for FDE?

> The gentleman is not an implement.
> — Confucius, The Analects, "Wei Zheng"

Look back five years from now, and part of any FDE's working inventory will already be worthless: fluency in a particular framework, the feel of a particular prompting style, the click-paths of a particular product. Tools update too fast — yesterday's new technology is tomorrow's default feature.

But some things don't expire with the version: knowing what to evaluate, how to design acceptance, who should be accountable for the result; knowing that a solution that runs in a demo and a solution that survives on the customer's site are two different things.

Perhaps by then the title "FDE" itself will have become uncommon — folded into product, engineering, industry solutions, or delivery teams, renamed to whatever sounds more fashionable.

But when a customer's system breaks, someone will still have to answer: can this solution actually reach the result — and if it can't, who is accountable?

So whether the profession disappears isn't the first endgame question. Ask two things first: will the experience you accumulate today still be worth anything in five years? And who will be carrying, by then, the responsibility for hauling new technology to a production result? This chapter answers both on three levels — the individual, the role, and the industry.

I. The Individual Level: A Balance Sheet for Experience

The most personal question first: is the experience this line of work builds up worth anything five years from now? Sort what the job gives you into three columns.

Depreciating: tricks for using a specific framework, the feel of a prompting style, the operational details of a particular product — tool-layer experience depreciates fastest. Every round of iteration in models, tools, and workflows revalues this column.

Holding value: evaluation method, acceptance design, accountability management, organizational drive — the meta-capability layer, compounding across industries, and

appreciating as cases accumulate.

Appreciating: industry depth plus reusable assets — connectors, templates, eval sets. What makes this column special is that it can be **traded**: it can travel with the person and become your market price, or stay inside the organization and become part of the platform and your bargaining power.

The career advice in one line: **rebalance the depreciating column into the holding-value column on a schedule.** Whenever you notice yourself getting fluent in something about to become obsolete, ask one question: what method underneath it should I be extracting? Rebalance diligently, and your shelf life extends indefinitely.

II. Three Paths, and Each One's Bar

These assets won't stay on a résumé. As experience deepens, they carry a person toward three different positions: turning delivery experience into team capability, into product capability, or into a company's capacity for sustained delivery.

The management path: turning experience into team capability. Moving from delivery lead toward bigger management roles, the real turn isn't carrying more projects — it's going from "carrying the result myself" to "building a mechanism that keeps carrying the result." Whether you can grow people, assign responsibility, and handle conflict sets the ceiling of this path.

The product path: turning experience into product capability. Problems that recur in the field, eval structures, and delivery templates can all be deposited into the platform. Retool's FDE team has incubated more than a dozen startups — proof that front-line experience can be the starting point for products and companies. [1] The bar on this path is recognizing what's worth standardizing, and being willing to shift from solving one customer's problem to solving a class of customers' problems.

The startup path: turning experience into a capacity for sustained delivery. The hard part of a vertical studio or a product startup isn't landing the first deal — it's organizing acquisition, solutioning, delivery, support, and productization into a system that can run over and over. This capability is especially scarce while an industry's AI demand is just moving from novelty to real budgets: too early, and you have to educate the market; too late, and the advantage gets swallowed by homogenization.

This is also why end-to-end field experience becomes a training ground for founders: it forces a

[1] @jamiecuffe's own account of the product line incubated inside Retool

person to understand, at the same time, why the customer pays, how the solution lands, and who carries the result after delivery.

Nabeel Qureshi, a former Palantir employee, observed that although Google has roughly 50 times Palantir's headcount, the number of companies founded by ex-Palantir employees in each YC batch is usually larger. [2] The comparison can't separate the training effect from the self-selection of the people involved, but it points at something: people used to carrying things from start to finish in the field find it easier to see problems and businesses that can stand on their own.

III. The Title Changes, the Responsibility Stays

On this there is no consensus — two front-line practitioners give opposite predictions. Jove Zhong (Cresta) believes FDE will remain the backbone for a long time: "when a company has good salespeople, sales is very hard to replace either"; the FDE belongs to the category of "very grounded people with engineering capability who can also use AI well," and that part is hard to hand to AI. Vincent Lu (Ventus AI) believes FDE will drift increasingly toward customer success: "the technical bar keeps getting lower, so how do you out-compete other companies? By providing emotional value, better service, a better understanding of what's needed." [3] Both predictions come from people in the field, and they point in two directions — which is exactly this chapter's opening stance confirmed: the answer to the endgame shouldn't be prophecy; what to watch are signals reality can check (see Figure 28-1).

[2] "Reflections on Palantir" (Nabeel Qureshi)

[3] Tencent Research Institute, AI Lens Roundtable, Episode 6: "FDE Non-Consensus Views and a Field Guide from Silicon Valley Founders"

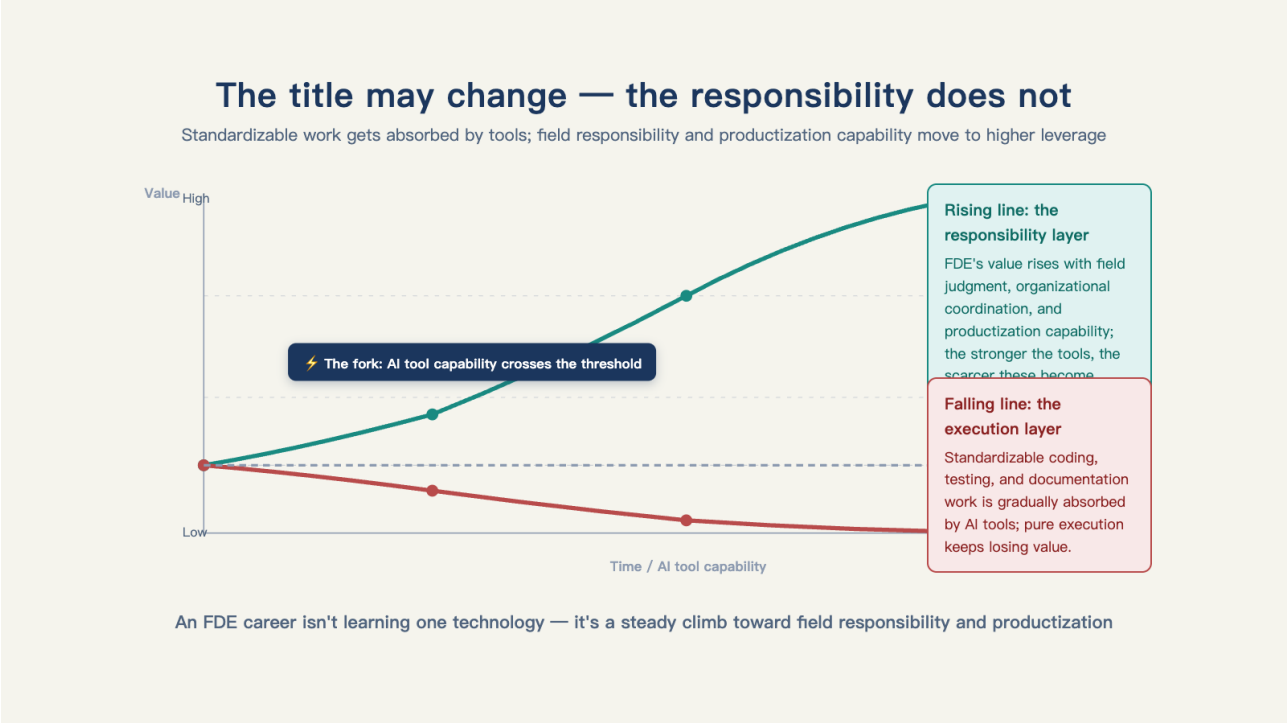

Figure 28–1: The title may change; the responsibility for getting real problems into production and feeding assets back does not disappear.

Will the FDE title still exist in five years? Maybe not. But that doesn't mean this kind of work disappears — it means the work is being redistributed.

Part of the execution gets taken by AI and platforms. Palantir has written deployment, configuration, and troubleshooting into the AI FDE's product capabilities; that's a product claim, but it points clearly in one direction: repetitive, standard execution will depend less and less on being done by hand. [4]

Another part gets taken by productization. When one kind of problem keeps recurring and the solution gradually stabilizes, it moves from being delivered on-site, time after time, to being a default capability on the platform.

Once both ends have absorbed their share, what's left in the middle? **Judgment and guarantee** — the judgment of "will this solution work in this customer's production environment," and the guarantee of "if it works, I own it; if it doesn't, I backstop it." The eternal need in B2B procurement isn't code — it's **risk transfer**: what the customer buys has always been "someone to carry this new technology to a result on my behalf." That core doesn't get absorbed — AI can substitute for execution, but it cannot take on the customer's accountability for the result.

IV. The Stronger the Technology, the More the Front Line Persists

[4] Palantir Foundry, "AI FDE" product documentation

Widen the view to the whole industry: between model companies, application companies, and enterprise customers sits the same gap — the capability arriving upstream runs ahead, and the part actually usable downstream lags behind. Filling that gap takes putting new capability into concrete workflows, permissions, and organizations; this is not a defect waiting to be eliminated — it's commercial space that keeps reappearing.

One Chinese practitioner sees the drag of old processes: bosses readily overestimate the side of AI that can do anything, and underestimate the side that gets trapped by existing institutions and ways of working together. He expects the market to converge in the end on two forms — standardized products, or standardized services. [5]

An American practitioner sees the time lag between capability and adoption. Models upgrade generation by generation, but how fast people actually put them to use doesn't necessarily keep up; AI needs people to explore and push it forward, and needs people to absorb the friction of landing it. [6]

The two observations don't contradict. The first explains why new technology gets dragged down by old organizations; the second explains why capability, however strong, doesn't happen by itself. Model companies are like a headquarters product team, while application companies and customer sites keep turning new capability into usable things; each layer does the front-line work for the layer below. [6]

This rhythm isn't unique to AI. The economic historian Carlota Perez described how technological revolutions move from being built and validated by a few early movers to spreading at scale. [7] The last mile between capability and adoption will persist for the long run — it just keeps moving up: today on the enterprise's front line, tomorrow inside industry processes and organizational collaboration.

V. What Three Kinds of Readers Should Do Now

The answer to the endgame shouldn't be prophecy. What follows are trackable actions and signals: if the judgment is wrong, readers will find out early.

Individuals: which column of the balance sheet to invest in — carry fewer depreciating items, accumulate more holding-value and appreciating items; run a rebalancing review once every three years.

[5] Turning Point (破局点), Episode 31: "FDE, Chinese Style"

[6] Y Combinator: The FDE Playbook for AI Startups (Bob McGrew)

[7] Carlota Perez, Technological Revolutions and Financial Capital: The Dynamics of Bubbles and Golden Ages

Companies: confirm three things first — is there a clear business problem, is the business side willing to commit people, data, and time, and has one small-scale trial already been run. Until all three are in place, buy a small service first; don't rush to create the role.

Industry observers: watch three signals — how fast the automation boundary of AI-FDE-type products is expanding, whether the share of outcome-based pricing in enterprise contracts is rising, and whether the "FDE" title disappears while the function survives under another name. They will tell you how this chapter's judgments are being confirmed or falsified.

VI. To Close

The answer fits in one sentence: **AI projects stall not because they can't be built, but because nobody is accountable for the last mile.** The model lives in the cloud and capability keeps growing; the real difficulty sits in "can be put to use, can stay in use, and can be accounted for."

FDE is what this responsibility is called right now. The name will change — into product capability, into a platform function, into some other title; the responsibility won't — as long as what enterprises procure is still "carrying new technology to a production result," some role will always have to own it end to end. This book teaches the full technical detail of that responsibility; and you, having read this book — whether you stand in delivery, product, or organization — are already looking at the problem through that same lens.

The road is laid to here. The rest — go explore and discover it in the field.

Appendix A · FDE Delivery Checklists

How to Use This Checklist

- Each item is marked: ☐ met / ◐ partial / ☐ not met / N/A not applicable
- The rule: if any phase's critical item (marked ★) is unmet, do not move on to the next phase
- **Consequence notes**: what typically happens when an item goes unmet (red-line risks)
- **Chapter references**: each item lists the chapter it comes from, for looking up the details

Phase I: Discovery (Chapter 6)

> **Phase goal**: locate a high-frequency, high-value, verifiable problem inside the customer's workflow; define the baseline and the success metric first.

Critical items (★)

- ★ **The problem is clearly defined**
- Elements: a one-sentence problem statement + current baseline data + a target metric
- See: Chapter 6
- If unmet:
- A vague problem definition → acceptance never reconciles with the numbers later
- No baseline data → no way to prove "better than before" after launch
- No target → the project stays forever at "pretty good, let's try a bit more"
- Checklist: [ ] defined clearly in one sentence [ ] baseline numbers have a source [ ] target value and time window are frozen

- ★ **The baseline has been measured**
- Elements: hard numbers (not estimates) for the manual process's time / error rate / cost
- See: Chapter 6
- If unmet:
- Only qualitative assessments ("slow", "error-prone") → no quantified before/after comparison

- Test sample too small or unrepresentative → the data isn't credible and the customer won't accept it
- Measurement method not kept consistent → before and after get measured on different scales, and the numbers explode
- Checklist: [ ] sample size ⩾ 30 [ ] measurement method documented [ ] measured at least twice

- **★ Users have tried an interactive prototype and given real feedback**
- Elements: one end-to-end interaction on real data with real users, collecting genuine feedback (not just "they nodded")
- See: Chapters 6 and 7
- If unmet:
- The prototype only ever ran on demo data → at launch, the data's distribution and format turn out completely different
- Users nodded but never really used it → the understanding of the requirement never got past the customer's one sentence
- No real feedback collected (how customers use it, what they gripe about) → version two is guesswork again
- Checklist: [ ] real data used (real distribution, real volume) [ ] at least 1 real front-line user has tried it [ ] user feedback / questions / points of confusion recorded

Non-critical items (optional but recommended)

- **Data-reachability checklist completed**
- Contents: where the data is / who owns it / who can see it / masking rules / the exception process
- See: Chapter 10
- Why do it early: missing data, missing permissions, and legal problems get caught while there's still time to fix them

- **Tolerance for change confirmed**
- Contents: the process owner knows, the affected people are identified, and the blast radius of the change has been calculated
- See: Chapter 15
- Why do it early: avoids "a flawless launch and broken users"

Phase II: Minimum Viable Deployment, MVD (Chapters 7 and 12)

Phase goal: use real data and real users to test quickly whether the solution actually works; use the smallest possible scope to keep the cost of acceptance under control.

Critical items (★)

- **★ Real data (real distribution, not demo data)**
- Elements: volume, distribution, and format all match the production environment; not "we picked our best 100 records"
- See: Chapter 7
- If unmet:
- Demo data with a simple distribution → 95% in the PoC, 65% in production, and a customer complaint about "shrinkage"
- Demo data volume too small → second-level responses in the PoC, hundred-scale latency at launch
- Demo never covered edge cases → one class of data goes in and the system falls over
- Checklist: [ ] data volume representative of the production environment [ ] edge cases at least 5% of the set [ ] sources traceable

- **★ Real users (at least one front-line user genuinely using it every week)**
- Elements: not "the boss took a look" — the people who actually do the work, using it weekly, able to give feedback
- See: Chapter 7
- If unmet:
- Only management signed off → the people who execute don't cooperate, and the feature design drifts completely away from the use case
- The users aren't the people doing the work → the workflow's real bottlenecks never surface
- The users don't use it often → sporadic use never exposes stability problems
- Checklist: [ ] user list has names / departments [ ] users are the actual operators (not the buyers) [ ] weekly usage records exist

- **★ Golden Set prepared (at least 50 representative examples, with a designated judge)**
- Elements: collect 50 examples representing real front-line scenarios ("this is how the problem should be solved"); explicitly designate one person responsible for judging correctness against them
- See: Chapter 12
- If unmet:
- Too few samples (<20) → the real diversity isn't represented and the evaluation results are void

- No designated judge → labeling disputes can't be resolved and the work gets redone
- Biased samples → only the scenarios the customer cares most about, missing 80% of the actual problems
- Checklist: [ ] at least 50 samples [ ] samples cover at least 3 major scenarios [ ] one judge designated [ ] version history kept for the samples and the final eval set

Non-critical items (optional)

- **Scope boundaries made explicit**
- Contents: draw the boundary clearly — one process, one user group, and the agent takes only the measurable segment
- See: Chapter 7
- Recommended early: keeps the MVD from becoming "let's try every feature"

Phase III: Production (Chapters 8–14)

> **Phase goal**: build the PoC out into a system that runs on its own in a real enterprise environment. Take down the "eight walls" from Chapter 9 one at a time.

Critical items (★) — the walls that must come down

Foundation (if these don't go through, nothing else is worth discussing)

- **★ Wall One: data and permissions → sign the Data Contract**
- Contents: scope / visibility / retention / audit — four items on one sheet
- Elements: which tables can be read, who can see the results, how masking works, what process handles exception permissions
- See: Chapters 9 and 10
- If unmet:
- Only after launch does it turn out a table can't be read → the process jams
- No masking agreement → the customer complains about sensitive information leaking
- Result visibility never defined → some stakeholders can't see the output, and the project fails
- Checklist: [ ] all data sources to be read listed on one sheet [ ] permission owner marked for each source [ ] masking rules attached as a table [ ] audit-log plan confirmed

- **★ Wall Two: integration → an integration path that actually connects the old system to**

the new

- Contents: how to connect to the system the agent writes back to (API / database / RPA / other)
- Elements: interface protocol confirmed, data-format mapping, retry-on-failure policy, idempotency guarantees
- See: Chapters 9 and 10
- If unmet:
- API availability never verified up front → after launch, the interface turns out never to have actually worked
- No retry on failure → a transient network blip loses data
- No idempotency check → the retry processes the customer's data twice
- Checklist: [ ] integration approach PoC'd (connected for real at least once) [ ] network errors, timeouts, and malformed responses all tested [ ] idempotency design documented [ ] rollback-on-failure plan explicit

Risk (it runs, but something can go wrong at any moment)

- **★ Wall Three: tool boundaries → a Tool Contract**
- Contents: which tools the agent can call, who validates the parameters, when a call gets refused
- Elements: tool inventory / parameter-validation rules / safeguards on high-risk tools (sending email, modifying data)
- See: Chapters 9 and 11
- If unmet:
- No parameter validation → the agent sends the wrong email or deletes the wrong record
- No tool whitelist → the agent calls tools it should have no access to
- No rate limits → the agent fires DDoS-level request volumes at an external API
- Checklist: [ ] tool inventory whitelisted [ ] parameter-validation rules for every tool written into code [ ] approval tickets required for high-risk tools (write operations, outbound sends) [ ] call-rate and concurrency limits confirmed

- **★ Wall Four: controllable execution → a Run State Machine + Checkpoint & Resume**
- Contents: what happens when a task fails, whether a run stuck on human approval can resume, whether a mid-run crash can pick up where it left off
- Elements: state-transition diagram / checkpoints / recovery policy
- See: Chapters 9 and 11
- If unmet:
- No checkpoints → a failed task reruns from scratch and customer data gets processed twice

- No suspension point for human approval → anything needing human confirmation jams and blocks the whole chain
- No degradation path → a system failure is a business failure, with no manual fallback
- Checklist: [ ] state-transition diagram drawn [ ] at least 3 critical checkpoints implemented [ ] the human-approval node can suspend and resume [ ] the fallback-to-manual/Excel path rehearsed

Trust (without these, the system stays stuck at trial forever)

- **★ Wall Five: provable results → an eval set + a release gate for changes**
- Contents: after a prompt / model / tool change, the data says whether results went up or down
- Elements: a Regression Set / a benchmark version / before-and-after comparison on every change
- See: Chapters 9 and 12
- If unmet:
- No Regression Set → every change is a gamble, and a week of changes breaks the system
- No benchmark version → nobody can tell improvement from regression
- No release gate → anyone can change anything, and when it breaks there's no one to hold accountable
- Checklist: [ ] Regression Set at least 200 cases [ ] baseline version frozen and recorded [ ] automated before/after comparison on every change [ ] change log traceable

- **★ Wall Six: trustworthy results → a chain of evidence**
- Contents: when the customer asks "where did this number come from" or "why should I believe this conclusion," you can produce the evidence
- Elements: execution records per call / original inputs / intermediate reasoning steps / the basis for the final decision
- See: Chapters 9 and 13
- If unmet:
- No execution records → a challenged result can only be rerun, never traced
- Incomplete execution records → the customer sees the result but not the reasoning behind it
- No archived raw data → once the data changes, a post-mortem becomes impossible
- Checklist: [ ] every call's input logged [ ] intermediate steps (retrieval results, model output) inspectable [ ] the basis for the final decision showable to the customer [ ] log retention period defined

- **★ Wall Seven: cost and stability → cost attribution + a budget circuit breaker**

- Contents: how the bill gets computed, what happens when it's exceeded, how the system meets its Service Level Objectives (SLOs)
- Elements: per-call cost attribution / cost alerts / automatic degradation when the budget is exceeded
- See: Chapters 9 and 14
- If unmet:
- No cost attribution → the month-end bill explodes and the customer is furious
- No alerts → by the time anyone notices, spend is running at several times the budget
- No automatic degradation → expensive calls keep running while costs are already high, a vicious cycle
- Checklist: [ ] cost model documented [ ] average cost per 1,000 calls measured [ ] budget-alert thresholds set [ ] automatic degradation on budget overrun implemented

- **★ Wall Eight: transferable responsibility → a responsibility handoff table + a runbook**
- Contents: who is responsible when something breaks, and who the system gets handed to once the FDE leaves
- Elements: incident classification / first responsible party / escalation path / operations documentation
- See: Chapters 8 and 17
- If unmet:
- No responsibility table → when something breaks there's nowhere to point a finger, and the relationship breaks
- The runbook lives only on the vendor's side → the customer can't take over
- No escalation path → an alert fires in the middle of the night and there's no one to reach
- Checklist: [ ] responsibility matrix (incident type × responsible party) drawn up [ ] escalation timing and escalation target explicit [ ] the customer has reviewed the runbook [ ] at least one incident drill run

Non-critical items (check them; phasing is acceptable)

- **Approval tickets for high-risk actions**
- Contents: a human approves before any write operation; run_id / approver_id / action details recorded
- See: Chapter 11

- **Service Level Objective (SLO) commitments on three dimensions (availability / latency / quality) + a degradation path**
- Contents: the metrics written into the Service Level Agreement (SLA) contract, defined across the three dimensions
- See: Chapter 14

- **Risk-tier table (read-only / reversible / irreversible / outbound)**
- Contents: classify every operation by risk tier
- See: Chapter 11

Phase IV: Launch and Adoption (Chapters 8, 15, and 16)

> **Phase goal**: results proven, system stable. The last gate from trial to production: the customer has to dare to take it, and employees have to want to use it.

Critical items (★)

- **★ The outcome acceptance sheet (metric + baseline + window + denominator definition + who decides, frozen in advance)**
- Contents: all five elements complete — print one copy and pin it in the review room
- Elements:
- Baseline value (the manual process's measured level, from Chapter 6)
- Target value (the level to be reached)
- Measurement window (weekly / monthly)
- Denominator definition (who counts, who doesn't)
- Who decides (one person's name, not "the business side")
- See: Chapter 8
- If unmet:
- Metric not frozen in advance → after launch the metric gets changed because "this month's data is a special case," and the bickering never ends
- Murky denominator → the customer says "the target wasn't met," the vendor says "you're counting wrong"
- No named judge → nobody can persuade anybody, and the project hangs unresolved
- Checklist: [ ] all five elements written into the contract [ ] the judge has signed [ ] rollback lines for each stage of the gradual rollout quantified

- **★ The operational readiness checklist (someone watches the alerts / degradation works / someone has actually read the runbook)**
- Contents: four checks — not one can be skipped
- Elements:
- Someone watches the alerts: when one fires, it maps to a name on the on-call roster
- A working degradation path: including the fallback to Excel and manual work, and it has

been rehearsed

- A runbook: the customer holds a copy and has read it through
- An on-call schedule: who covers the first week after launch and the committed response time, in black and white
- See: Chapter 8
- If unmet:
- Alerts fire with no one watching → the customer finds out about the outage first
- No degradation path → every system failure is a business outage
- The runbook lives only on the vendor's side → a middle-of-the-night failure with no one to take the customer's call
- A blank on-call sheet → who handles the first alert?
- Checklist: [ ] the on-call roster has full coverage [ ] the degradation procedure rehearsed once [ ] the customer has confirmed receipt of the runbook and read it [ ] the on-call Service Level Agreement (SLA) on response time signed

- ★ **The responsibility handoff table (a first responsible party for every incident scenario)**
- Contents: list every way the system could die, one by one, writing in the first responsible party and the escalation path for each
- Example:

What Went Wrong	First Responsible Party	Escalation Path
Model output error causing business loss	Who judges it, who blocks it, who pays for it	To both sides' leads within 1 hour
Data source failure / upstream outage	The data-side point of contact	To vendor on-call within 4 hours
Incident caused by user error	Customer-side system administrator	To a joint post-mortem, same day
Cost overrun	The platform provider	To the project lead, in the weekly report

- See: Chapter 8
- If unmet:
- No responsibility table → when something breaks, "this isn't our problem" ←→ "how could this possibly be our problem," and the relationship breaks
- No escalation path → a middle-of-the-night failure with no one answering, and the incident escalates into a customer complaint
- Legal liability written down but no handling procedure → names on paper, no one picks up in reality
- Checklist: [ ] at least 5 plausible failure scenarios listed [ ] a first responsible party named for each (a person's name, not a department) [ ] escalation timing and escalation target explicit [ ] signed by both sides

Non-critical items (recommended, phasing acceptable)

- **Gradual rollout plan (staged traffic increase + a rollback line at each stage)**
- Contents: first 10% of users → 20% → 50% → 100%, with explicit rollback conditions at every stage
- See: Chapter 8

- ★ **The three incentive questions answered** (quick reference: did the scorecard change / who keeps the time saved / whose problem is a mistake)
- Contents: why users would use this system at all
- See: Chapter 15

- **Champion network (one champion per team)**
- Contents: in every affected department, one person who understands the system and can answer colleagues' questions
- See: Chapter 15

- **Activation metric definitions (activation rate / diversion rate, with the denominator and window pinned down)**
- Contents: how to count a user as genuinely using the system
- See: Chapter 15

Phase V: Handoff and Flowing Back (Chapters 16, 17, 21, and 22)

> **Phase goal**: pass the baton cleanly from the FDE to the customer; distill what this delivery produced into reusable product capability.

Critical items (★)

- ★ **Handover of the three assets**
- Contents: three things the customer must catch (miss one, and the handoff doesn't yet count as successful)
- Item one: **the operations manual**
- Includes: a fault-troubleshooting tree, a Q&A for common questions, the on-call procedure, and the manual-fallback steps
- Customer acceptance standard: at least one person can work through one real incident using the manual

- See: Chapter 17
- Item two: **the eval set + maintenance responsibility**
- Includes: the eval set (for ongoing verification) + a plan for replenishing it on a schedule (and who owns that)
- Customer acceptance standard: they can run the evaluation themselves and know how to add new samples
- See: Chapter 17
- Item three: **the incident playbook**
- Includes: diagnosis and handling steps for the common failures, each with a worked example
- Customer acceptance standard: they have seen the scenarios in the playbook and know how to handle them
- See: Chapter 17
- If unmet:
- The manual never rehearsed → the first real incident sends them straight back to the FDE
- The customer doesn't know how to maintain the eval set → the system drifts and nobody notices
- No incident playbook → every failure becomes an emergency call for help
- Checklist: [ ] all three documents received and reviewed by the customer [ ] each one rehearsed once (not "seen" — "done") [ ] the customer can independently handle the 3 most common failures

- ★ **Every customization request tagged (one-off / industry-common / general-purpose)**
- Contents: of this project's requirements, which belong only to this customer, which are industry-common, and which can be distilled into product
- See: Chapters 21 and 22
- If unmet:
- Everything treated as a "one-off requirement" → the next customer starts from zero again
- Everything promised as "industry-common" → the promise is too heavy and later projects' costs explode
- Nothing tagged → at the feedback review, nobody knows what to distill
- Checklist: [ ] every customization requirement labeled with a category [ ] one-off requirements carry an explicit sunset date [ ] first-pass code for the general-purpose capability committed

Non-critical items (recommended)

- **The handoff staircase executed (run alongside → shadow → exit, with a time frame and**

exit criterion at each step)

- Contents: stage one, the FDE runs alongside; stage two, the FDE shadows; stage three, the customer runs it alone
- See: Chapter 17

- **Customer-side operators pass a hands-on check (not just handed documents)**
- Contents: the customer's point of contact has actually operated the system, hit problems, and solved some on their own
- See: Chapter 17

- **At least one feedback review (delivery + product, both seats filled)**
- Contents: after the project closes, the delivery side and the product side sit down together and sort out what capability from this engagement is worth keeping
- See: Chapter 22

- **Customization trend tracked (customer N vs customer 1)**
- Contents: whether the customization load is rising or falling — the product-market-fit indicator
- See: Chapter 21

Usage Recommendations

For FDE teams

- **Phases 1–3**: score the project against the checklist at kickoff, marking each item //
- **Phases 4–5**: two weeks before launch, bring the customer in and walk the list item by item, closing out every ★-marked must-have
- **Ongoing maintenance**: after each project, run a retrospective and add the pitfalls this project surfaced

For customer-side procurement

- Write the "critical items" section of this checklist into your procurement requirements
- Cross-reference Chapter 8's customer self-check list against the critical items here, and ask whether your procurement process itself manufactures the "look but don't touch" trap

For project managers

- **Early assessment**: use the checklist to gauge quickly how far a "PoC complete" project

still is from production

- **Risk management**: turn the checklist into a Gantt chart, and the effort and dependencies behind each of the eight walls become visible at a glance

---

Appendix C · Sources and Citation Notes

Every citation in this book points to an entry in this list. Footnotes in the main text carry only the source ID (e.g. `[c06]`); the full bibliographic record lives here — **one source, defined exactly once for the whole book**.

To trace the basis for any given claim, note the ID in the main text and turn to the matching entry here. There are 38 entries in all — the book's entire evidence base; material never cited in the main text is not listed.

Source List

1. enterprise_agent_platform (open-source enterprise agent platform engineering project)

- **Source**: GitHub · datagallery-lab/enterprise_agent_platform, Apache-2.0
- **Cited in**: Chapters 9—14

2. Anthropic, "Building a New Enterprise AI Services Company with Blackstone, Hellman & Friedman, and Goldman Sachs"

- **Source**: official announcement on anthropic.com, 2026-05-04
- **Cited in**: Chapter 4

3. Accenture Newsroom, "Accenture Launches Microsoft Forward Deployed Engineering Practice to Help Organizations Scale AI Across the Enterprise"

- **Source**: official Accenture press release, 2026-03-18
- **Cited in**: Chapters 4, 26, and 27

4. Palantir Foundry, "AI FDE" product documentation

- **Source**: palantir.com/docs/foundry/ai-fde
- **Cited in**: Chapters 4, 5, 9, 10, and 28

5. Crossroads × Rolling AI, "A Conversation on FDE" (podcast)

- **Source**: Xiaoyuzhou podcast, 2026–06–03
- **Cited in**: Chapters 2, 15—17, 20, and 27

6. "AI Job Sense: Stop the Agent Rat Race"

- **Source**: Xiaoyuzhou podcast, 2026–06–21
- **Cited in**: Chapters 6, 10, 11, 15, 16, 18, and 23

7. Yilu Tongxing (一路瞳行), Episode 134: "From AI Assistant to Digital Employee"

- **Source**: Xiaoyuzhou podcast, 2026–04–10
- **Cited in**: Chapters 6, 7, 15, 17, and 24

8. Baseten, "What I Learned as a Forward-Deployed Engineer Working at an AI Startup" (Het Trivedi)

- **Source**: Baseten blog, baseten.co
- **Cited in**: Chapters 1, 18, 23, and 24

9. Hacker News, "A Week in the Life of a Forward Deployed Engineer" (10 customers, 50 hours)

- **Source**: Hacker News thread
- **Cited in**: Chapters 8, 23, and 24

10. "I Thought FDE Meant Fighting Alone at the Customer Site" — What Two Months on the Job Actually Looks Like

- **Source**: note.com · AI Shift (Japanese)
- **Cited in**: Chapters 6, 10, 23, and 26

11. Inside an Applied AI company (Pace long-form post)

- **Source**: long-form post on X (formerly Twitter)
- **Cited in**: Chapters 7—9, 11—14, 21, 22, and 26

12. Turning Point (破局点), Episode 31: "FDE, Chinese Style"

- **Source**: Xiaoyuzhou podcast, ca. 2026–08
- **Cited in**: Chapters 3, 5, 6, 15, 18, 20, 21, and 23—28

13. "Reflections on Palantir" (Nabeel Qureshi)

- **Source**: nabeelqu.co, 2024–10
- **Cited in**: Chapters 3, 6, 22, and 28

14. Silicon Valley 101, Episode 248: "China-Style FDE, with Alibaba Lingyang's Peng Xinyu"

- **Source**: Xiaoyuzhou podcast, 2026–08
- **Cited in**: Chapters 16, 18, 20, 21, and 26

15. Tencent Research Institute, AI Lens Roundtable, Episode 6: "FDE Non-Consensus Views and a Field Guide from Silicon Valley Founders"

- **Source**: WeChat official account, 2026–07–08
- **Cited in**: Chapters 2, 9, 12, 16, 18, 22, 24, 27, and 28

16. a16z: Box CEO on AI agents and why enterprises can't keep up

- **Source**: YouTube, 2026–04–28
- **Cited in**: Chapters 3, 5, 9, 10, and 16

17. Y Combinator: The FDE Playbook for AI Startups (Bob McGrew)

- **Source**: YouTube, 2025–09–08, about 50 minutes
- **Cited in**: Chapters 1, 5—7, 19, 21, 22, 24, and 26—28

18. Every engineer should do a stint in consulting (Hacker News thread)

- **Source**: Hacker News, 2021
- **Cited in**: Preface

19. @deployengineer (Bhaulik Patel), essay series

- **Source**: X (formerly Twitter)
- **Cited in**: Chapters 2, 3, 9, 17, 19, and 26

20. European remote FDE job description (reposted by @afahmy_dev)

- **Source**: X (formerly Twitter)

- **Cited in**: Chapter 1

21. "A million dollars in 30 days" course-marketing post (@gregisenberg)

- **Source**: X (formerly Twitter)
- **Cited in**: Chapter 25

22. "Doing FDE Work in Wuhan, Looking for a Partner" (linux.do thread)

- **Source**: the linux.do community
- **Cited in**: Chapters 18 and 27

23. @jamiecuffe's own account of the product line incubated inside Retool

- **Source**: X (formerly Twitter)
- **Cited in**: Chapter 28

24. "Anyone Out There Actually Doing FDE Work?" (linux.do thread)

- **Source**: the linux.do community
- **Cited in**: Chapters 1, 2, 3, 16, 18, 23, and 24

25. Switching from R&D to FDE: A Step Up, or Just Moving Into Implementation? (linux.do thread)

- **Source**: the linux.do community
- **Cited in**: Chapter 25

26. "FDE Engineer 101: Bridging the Gap Between AI and Business" (Chinese-language compilation video)

- **Source**: Xiaohongshu, unattributed, saved as a local export
- **Cited in**: Chapters 6, 8, 15, 24, and 27

27. linux.do post #2580046

- **Source**: the linux.do community, 2026-07-14
- **Cited in**: Chapters 1 and 3

28. Silicon Valley 101, Episode 240: "The Hottest New Job in Silicon Valley — FDE"

- **Source**: YouTube; the guests are Cresta's FDE lead and the VP of enterprise business at Invisible
- **Cited in**: Chapters 5, 6, 10, 11, 20, 23, 24, and 26

29. Palantir official documentation: the Ontology architecture and permission system

- **Source**: palantir.com/docs/foundry/architecture-center/ontology-system
- **Cited in**: Chapters 21 and 22

30. The Dynamo and the Computer: An Historical Perspective on the Modern Productivity Paradox

- **Source**: Paul A. David, American Economic Review 80(2), 1990-05, pp. 355—361
- **Cited in**: Chapter 4

31. Ali Ghodsi's guest lecture in Stanford's MS&E 435 course

- **Source**: the course website's materials page, Week 4 (2026-04-24)
- **Cited in**: Chapter 9

32. Tencent Research Institute, "FDE Model: Industry Observations and Practice"

- **Source**: publicly shared Chinese-language reference material
- **Cited in**: Chapters 1 and 3

33. "100 Questions About FDE" (open-source ebook)

- **Source**: openly shared open-source ebook, 92 pages
- **Author's page**: https://my.feishu.cn/wiki/QWQHwA9uYibmtzkZLJBccaEhnNd
- **Cited in**: Chapters 6, 7, 12, 15, 16, 18, 24, and 25

34. Fan Bing, Forward Deployed Engineer (FDE) (XDash open-source book)

- **Source**: open-source ebook, v1.0.23, 2026-08
- **Cited in**: Chapters 6—8, 12, 15—19, 21, and 25

35. Carlota Perez, Technological Revolutions and Financial Capital: The Dynamics of Bubbles and Golden Ages

- **Source**: published book (cited at second hand, via local reading notes)
- **Cited in**: Chapter 28

36. MIT Project NANDA, The GenAI Divide: State of AI in Business 2025

- **Source**: research report from MIT Media Lab's Project NANDA, 2025–07
- **Cited in**: Preface

37. Forward deployed engineer is AI's hottest job as OpenAI and Google race to hire

- **Source**: reporting by The New Stack
- **Cited in**: Chapter 4

38. We'd Better Watch Out (book review)

- **Source**: Robert M. Solow, New York Times Book Review, 1987–07–12, p. 36
- **Cited in**: Chapter 4